

**PDFlib GmbH München, Germany**

**[www.pdflib.com](http://www.pdflib.com)**



**General Edition for  
Cobol, C, C++, Java, Perl,  
PHP, Python, RPG, and Tcl**

Copyright © 1997–2004 PDFlib GmbH and Thomas Merz. All rights reserved.

PDFlib GmbH  
Tal 40, 80331 München, Germany  
[www.pdflib.com](http://www.pdflib.com)

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

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

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

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

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

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

PANTONE® colors displayed in the software application or in the user documentation may not match PANTONE-identified standards. Consult current PANTONE Color Publications for accurate color. PANTONE® and other Pantone, Inc. trademarks are the property of Pantone, Inc. © Pantone, Inc., 2003. Pantone, Inc. is the copyright owner of color data and/or software which are licensed to PDFlib GmbH to distribute for use only in combination with PDFlib Software. PANTONE Color Data and/or Software shall not be copied onto another disk or into memory unless as part of the execution of PDFlib Software.

PDFlib contains modified parts of the following third-party software:

ICCLib, Copyright © 1997-2002 Graeme W. Gill  
PNG image reference library (libpng), Copyright © 1998-2002 Glenn Randers-Pehrson  
Zlib compression library, Copyright © 1995-2002 Jean-loup Gailly and Mark Adler  
TIFFlib image library, Copyright © 1988-1997 Sam Leffler, Copyright © 1991-1997 Silicon Graphics, Inc.  
Cryptographic software written by Eric Young, Copyright © 1995-1998 Eric Young ([ey@cryptsoft.com](mailto:ey@cryptsoft.com))  
Independent JPEG Group's JPEG software, Copyright © 1991-1998, Thomas G. Lane

PDFlib contains the RSA Security, Inc. MD5 message digest algorithm.  
Viva Software GmbH contributed improvements to the font handling for Mac OS.



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

# Contents

## o Applying the PDFlib License Key 9

### 1 Introduction 11

- 1.1 PDFlib Programming 11
- 1.2 Major new Features in PDFlib 6 13
- 1.3 PDFlib Features 15
- 1.4 Availability of Features in different Products 17

### 2 PDFlib Language Bindings 19

- 2.1 Overview 19
- 2.2 Cobol Binding 20
  - 2.2.1 Special Considerations for Cobol 20
  - 2.2.2 The »Hello world« Example in Cobol 20
- 2.3 COM Binding 24
- 2.4 C Binding 24
  - 2.4.1 Availability and Special Considerations for C 24
  - 2.4.2 The »Hello world« Example in C 24
  - 2.4.3 Using PDFlib as a DLL loaded at Runtime 25
  - 2.4.4 Error Handling in C 26
  - 2.4.5 Memory Management in C 27
  - 2.4.6 Unicode in the C language binding 28
- 2.5 C++ Binding 28
  - 2.5.1 Availability and Special Considerations for C++ 28
  - 2.5.2 The »Hello world« Example in C++ 28
  - 2.5.3 Error Handling in C++ 29
  - 2.5.4 Memory Management in C++ 29
  - 2.5.5 Unicode in the C++ language binding 29
- 2.6 Java Binding 30
  - 2.6.1 Installing the PDFlib Java Edition 30
  - 2.6.2 The »Hello world« Example in Java 31
  - 2.6.3 Error Handling in Java 32
- 2.7 .NET Binding 33
- 2.8 Perl Binding 33
  - 2.8.1 Installing the PDFlib Perl Edition 33
  - 2.8.2 The »Hello world« Example in Perl 33
  - 2.8.3 Error Handling in Perl 34
- 2.9 PHP Binding 34
  - 2.9.1 Installing the PDFlib PHP Edition 34
  - 2.9.2 The »Hello world« Example in PHP 35

- 2.9.3 Error Handling in PHP 37
- 2.10 Python Binding 38
  - 2.10.1 Installing the PDFlib Python Edition 38
  - 2.10.2 The »Hello world« Example in Python 38
  - 2.10.3 Error Handling in Python 38
- 2.11 REALbasic Binding 39
- 2.12 RPG Binding 39
  - 2.12.1 Compiling and Binding RPG Programs for PDFlib 39
  - 2.12.2 The »Hello world« Example in RPG 39
  - 2.12.3 Error Handling in RPG 41
- 2.13 Tcl Binding 43
  - 2.13.1 Installing the PDFlib Tcl Edition 43
  - 2.13.2 The »Hello world« Example in Tcl 43
  - 2.13.3 Error Handling in Tcl 44

## 3 PDFlib Programming 45

- 3.1 General Programming 45
  - 3.1.1 PDFlib Program Structure and Function Scopes 45
  - 3.1.2 Parameters 45
  - 3.1.3 Exception Handling 46
  - 3.1.4 Option Lists 48
  - 3.1.5 The PDFlib Virtual File System (PVF) 50
  - 3.1.6 Resource Configuration and File Searching 51
  - 3.1.7 Generating PDF Documents in Memory 54
  - 3.1.8 Using PDFlib on EBCDIC-based Platforms 55
- 3.2 Page Descriptions 57
  - 3.2.1 Coordinate Systems 57
  - 3.2.2 Page Sizes and Coordinate Limits 59
  - 3.2.3 Paths 60
  - 3.2.4 Templates 61
- 3.3 Working with Color 63
  - 3.3.1 Color and Color Spaces 63
  - 3.3.2 Patterns and Smooth Shadings 63
  - 3.3.3 Spot Colors 64
  - 3.3.4 Color Management and ICC Profiles 67

## 4 Text Handling 71

- 4.1 Overview of Fonts and Encodings 71
  - 4.1.1 Supported Font Formats 71
  - 4.1.2 Encodings 72
  - 4.1.3 Support for the Unicode Standard 73
- 4.2 Font Format Details 74
  - 4.2.1 PostScript Fonts 74
  - 4.2.2 TrueType and OpenType Fonts 75

- 4.2.3 User-Defined (Type 3) Fonts 77
- 4.3 Font Embedding and Subsetting 78**
  - 4.3.1 How PDFlib Searches for Fonts 78
  - 4.3.2 Font Embedding 79
  - 4.3.3 Font Subsetting 81
- 4.4 Encoding Details 83**
  - 4.4.1 8-Bit Encodings 83
  - 4.4.2 Symbol Fonts and Font-specific Encodings 86
  - 4.4.3 Glyph ID Addressing for TrueType and OpenType Fonts 87
  - 4.4.4 The Euro Glyph 87
- 4.5 Unicode Support 89**
  - 4.5.1 Unicode for Page Content and Hypertext 89
  - 4.5.2 Content Strings, Hypertext Strings, and Name Strings 90
  - 4.5.3 String Handling in Unicode-capable Languages 91
  - 4.5.4 String Handling in non-Unicode-capable Languages 91
  - 4.5.5 Character References 94
  - 4.5.6 Unicode-compatible Fonts 95
- 4.6 Text Metrics and Text Variations 97**
  - 4.6.1 Font and Character Metrics 97
  - 4.6.2 Kerning 98
  - 4.6.3 Text Variations 99
- 4.7 Chinese, Japanese, and Korean Text 101**
  - 4.7.1 CJK support in Acrobat and PDF 101
  - 4.7.2 Standard CJK Fonts and CMaps 101
  - 4.7.3 Custom CJK Fonts 105
  - 4.7.4 Forcing monospaced Fonts 107
- 4.8 Placing and Fitting Single-Line Text 108**
  - 4.8.1 Simple Text Placement 108
  - 4.8.2 Placing Text in a Box 109
  - 4.8.3 Aligning Text 110
- 4.9 Multi-Line Textflows 111**
  - 4.9.1 Placing Textflows in the Fitbox 112
  - 4.9.2 Paragraph Formatting Options 113
  - 4.9.3 Inline Option Lists and Macros 114
  - 4.9.4 Tab Stops 116
  - 4.9.5 Numbered Lists 117
  - 4.9.6 Control Characters, Character Mapping, and Symbol Fonts 118
  - 4.9.7 Hyphenation 121
  - 4.9.8 Controlling the Linebreak Algorithm 122

## **5 Importing and Placing Objects 125**

- 5.1 Importing Raster Images 125**
  - 5.1.1 Basic Image Handling 125
  - 5.1.2 Supported Image File Formats 126
  - 5.1.3 Image Masks and Transparency 128

5.1.4	Colorizing Images	130
5.1.5	Multi-Page Image Files	131
5.1.6	OPI Support	131
5.2	Importing PDF Pages with PDI (PDF Import Library)	133
5.2.1	PDI Features and Applications	133
5.2.2	Using PDI Functions with PDFlib	133
5.2.3	Acceptable PDF Documents	135
5.3	Placing Images and Imported PDF Pages	136
5.3.1	Scaling, Orientation, and Rotation	136
5.3.2	Adjusting the Page Size	138
<b>6</b>	<b>Variable Data and Blocks</b>	<b>141</b>
6.1	Installing the PDFlib Block Plugin	141
6.2	Overview of the PDFlib Block Concept	142
6.2.1	Complete Separation of Document Design and Program Code	142
6.2.2	Block Properties	143
6.2.3	Why not use PDF Form Fields?	144
6.3	Creating PDFlib Blocks	146
6.3.1	Creating Blocks interactively with the PDFlib Block Plugin	146
6.3.2	Editing Block Properties	148
6.3.3	Copying Blocks between Pages and Documents	149
6.3.4	Converting PDF Form Fields to PDFlib Blocks	150
6.4	Standard Properties for Automated Processing	153
6.4.1	General Properties	153
6.4.2	Text Properties	155
6.4.3	Image Properties	158
6.4.4	PDF Properties	158
6.4.5	Custom Properties	159
6.5	Querying Block Names and Properties	160
6.6	PDFlib Block Specification	162
6.6.1	PDF Object Structure for PDFlib Blocks	162
6.6.2	Generating PDFlib Blocks with pdfmarks	164

## **7 Generating various PDF Flavors 167**

7.1	Acrobat and PDF Versions	167
7.2	Encrypted PDF	168
7.2.1	Strengths and Weaknesses of PDF Security	168
7.2.2	Protecting Documents with PDFlib	169
7.3	Web-Optimized (Linearized) PDF	170
7.4	PDF/X	171
7.4.1	The PDF/X Family of Standards	171
7.4.2	Generating PDF/X-conforming Output	172
7.4.3	Importing PDF/X Documents with PDI	174

- 7.5 Tagged PDF 176**
  - 7.5.1 Generating Tagged PDF with PDFlib 176
  - 7.5.2 Creating Tagged PDF with direct Text Output and Textflows 178
  - 7.5.3 Activating Items for complex Layouts 179
  - 7.5.1 Using Tagged PDF in Acrobat 182

## **8 API Reference for PDFlib, PDI, and PPS 185**

- 8.1 Data Types and Naming Conventions 185**
- 8.2 General Functions 187**
  - 8.2.1 Setup 187
  - 8.2.2 Document and Page 189
  - 8.2.3 Parameter Handling 198
  - 8.2.4 PDFlib Virtual File System (PVF) Functions 199
  - 8.2.5 Exception Handling 201
  - 8.2.6 Utility Functions 202
- 8.3 Text Functions 204**
  - 8.3.1 Font Handling 204
  - 8.3.2 User-defined (Type 3) Fonts 207
  - 8.3.3 Encoding Definition 209
  - 8.3.4 Simple Text Output 209
  - 8.3.5 Multi-Line Text Output with Textflows 216
- 8.4 Graphics Functions 224**
  - 8.4.1 Graphics State Functions 224
  - 8.4.2 Saving and Restoring Graphics States 226
  - 8.4.3 Coordinate System Transformation Functions 227
  - 8.4.4 Explicit Graphics States 229
  - 8.4.5 Path Construction 230
  - 8.4.6 Path Painting and Clipping 233
  - 8.4.7 Layer Functions 234
- 8.5 Color Functions 237**
  - 8.5.1 Setting Color and Color Space 237
  - 8.5.2 Patterns and Shadings 240
- 8.6 Image and Template Functions 243**
  - 8.6.1 Images 243
  - 8.6.2 Templates 249
  - 8.6.3 Thumbnails 249
- 8.7 PDF Import (PDI) Functions 251**
  - 8.7.1 Document and Page 251
  - 8.7.2 Other PDI Processing 255
  - 8.7.3 PDI Parameter Handling 255
- 8.8 Block Filling Functions (PPS) 258**
- 8.9 Hypertext Functions 261**
  - 8.9.1 Actions 261
  - 8.9.2 Named Destinations 264

- 8.9.3 Annotations **265**
- 8.9.4 Form Fields **269**
- 8.9.5 Bookmarks **274**
- 8.9.6 Document Information Fields **275**
- 8.9.7 Deprecated Hypertext Parameters and Functions **276**

**8.10 Structure Functions for Tagged PDF 278**

**A Literature 281**

**B PDFlib Quick Reference 283**

**C Revision History 288**

**Index 289**



# o Applying the PDFlib License Key

All binary versions of PDFlib, PDFlib+PDI, and PPS supplied by PDFlib GmbH can be used as fully functional evaluation versions regardless of whether or not you obtained a commercial license. However, unlicensed versions will display a *www.pdflib.com* demo stamp (the »nagger«) cross all generated pages. Companies which are seriously interested in PDFlib licensing and wish to get rid of the nagger during the evaluation phase or for prototype demos can submit their company and project details with a brief explanation to *sales@pdflib.com*, and apply for a temporary license key (we reserve the right to refuse evaluation keys, e.g. for anonymous requests).

Once you purchased a license key you must apply it in order to get rid of the demo stamp. There are several methods available:

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

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

The *license* parameter must be set only once, immediately after instantiating the PDFlib object (i.e., after *PDF\_new()* or equivalent call).

- Enter the license key in a text file according to the following format:

```
PDFlib license file 1.0
# Licensing information for PDFlib GmbH products
PDFlib 6.0.0 ...your license key...
```

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

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

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

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

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

**Accumulating individual CPU keys.** If you purchased multiple CPU licenses with more than one orders (as opposed to a single order for all of these CPU licenses), you can accumulate all keys in the license file by entering one after the other. The function *PDF\_set\_parameter()* also be called multiply for individual license keys. However, the Windows registry cannot be used to accumulate license keys.

**Evaluating features which are not yet licensed.** You can fully evaluate all feature by using the software without any license key applied. However, once you applied a valid license key for a particular product using features of a higher category will no longer be

available. For example, if you installed a valid PDFlib license key the PDI functionality will no longer be available for testing. Similarly, after installing a PDFlib+PDI license key the personalization features (block functions) will no longer be available.

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

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

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

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

PDFlib GmbH, Licensing Department

Tal 40, 80331 München, Germany

[www.pdflib.com](http://www.pdflib.com)

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

Licensing contact: [sales@pdflib.com](mailto:sales@pdflib.com)

Support for PDFlib licensees: [support@pdflib.com](mailto:support@pdflib.com)

# 1 Introduction

## 1.1 PDFlib Programming

**What is PDFlib?** PDFlib is a library which allows you to generate files in Adobe's Portable Document Format (PDF). PDFlib acts as a backend to your own programs. While you (the programmer) are responsible for retrieving the data to be processed, PDFlib takes over the task of generating the PDF code which graphically represents your data. While you must still format and arrange your text and graphical objects, PDFlib frees you from the internal details of PDF. Our binary packages contain different products in a single library:

- ▶ PDFlib contains all functions required to create PDF output containing text, vector graphics and images plus hypertext elements.
- ▶ PDFlib+PDI includes all PDFlib functions, plus the PDF Import Library (PDI) for including pages from existing PDF documents in the generated output.
- ▶ PDFlib Personalization Server (PPS) includes PDFlib+PDI, plus additional functions for automatically filling PDFlib blocks. Blocks are placeholders on the page which can be filled with text, images, or PDF pages. They can be created interactively with the PDFlib Block Plugin for Adobe Acrobat (Mac or Windows), and will be filled automatically with PPS. The plugin is included in PPS.

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

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

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

reflecting user input or some other dynamic data, e.g. data retrieved from the Web server's database. The PDFlib approach offers several advantages:

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

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

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

## 1.2 Major new Features in PDFlib 6

The following list discusses the most important new or improved features in PDFlib 6.

**Programming improvements.** Many restrictions in previous versions have been lifted. For example, pages can be created in arbitrary order, new pages can be inserted between existing ones, and more content can be added later to an existing page.

**Layers.** PDF's layer functionality (introduced in Acrobat 6) is important for CAD and engineering applications, but can also be used for impressive interactive documents, multi-lingual documentation, etc. PDFlib supports all layer control features available in PDF 1.5, including various controls which are not accessible in Acrobat.

**Unicode.** PDFlib 6 improves support for the Unicode standard by allowing Unicode strings in all relevant areas, such as file names, page content, hypertext, form fields, etc. This is especially important for users outside of Europe and North America.

**Text formatting.** The new textflow formatter offers a powerful, yet simple to use facility for formatting text according to a variety of options. Unicode text, ragged or justified text, arbitrary font changes, multi-line body text or large tables in an invoice – the new textflow features handles all common formatting tasks.

**Image handling.** TIFF image processing has been extended to cover TIFF flavors which were previously not supported, such as JPEG-compressed TIFFs or Lab and YCbCr color spaces. Since PDF 1.5 supports 16 bit color depth TIFF and PNG images with 16 bit per color component can now be converted to 16-bit color in PDF.

**Tagged PDF.** Tagged PDF is the key for accessible PDF according to section 508 in the USA and similar regulations in other countries. PDFlib is the first PDF library for general use which supports Tagged PDF generation. Using the new features it is very easy to create Tagged PDF from dynamic data. The generated output can leverage all Acrobat features for Tagged PDF, such as page reflow, read aloud, and improved export to other formats such as RTF, HTML, or XML. In combination with the new textflow formatter large amounts of text can quickly be transformed to Tagged PDF. For the first time ever, PDF generated dynamically on the Web server can satisfy accessibility regulations.

**PDF/X for Prepress.** PDFlib 6 is the first software on the market to support generating and processing PDF documents according to the latest 2003 editions of the PDF/X standards for prepress (PDF/X-1a:2003, PDF/X-2:2003, and PDF/X-3:2003). PDF/X plays an important role for file exchange in the prepress world. More and more publishers worldwide standardize on PDF/X for data exchange in order to implement reliable data exchange in the graphics arts industry. The new 2003 editions update, enhance, and unify the PDF/X family of standards.

**OPI for Prepress.** Some workflows in the graphics arts industry still rely on the OPI standard from the PostScript age, and use OPI information embedded in PDF documents. PDFlib 6 supports this by offering options for adding OPI information to imported images.

**Linearized PDF.** PDFlib 6 generates linearized PDF, also known as web-optimized PDF. This enables page-at-a-time download (also known as byteserving) when viewing PDFs in the Web browser, and significantly enhances the user experience.

**PDFlib Blocks for variable data processing.** The user interface of the PDFlib block plugin for creating PDF templates has been extended and streamlined. Blocks can now be filled with multi-line text, using the new textflow formatter. As a result, the PDFlib Personalization Server (PPS) is no longer restricted to simple mail-merge pieces with small amounts of text, but can also be used for complex applications with advanced text formatting requirements.

**Form fields.** All types of PDF form fields can be generated and enhanced with JavaScript and other actions. This can be used to create PDF forms dynamically subject to user input or database information.

**Hypertext.** PDFlib's hypertext features have been extended to fully support all PDF options for bookmarks, actions, and annotations. Page labels can be created to attach a symbolic name or roman numerals to a page, such as *i*, *ii*, *iii...* or *A-1*, *A-2*, etc.

**REALbasic.** As a new member in the large family of supported programming environments PDFlib 6 introduces a new language binding for REALbasic on Mac and Windows. REALbasic is a language for developing applications for multiple platforms. PDFlib 6 for REALbasic smoothly integrates into RB's object model, supports Unicode strings, and gives the developer access to all PDFlib features from within REALbasic.

# 1.3 PDFlib Features

Table 1.1 lists the major PDFlib features for generating and importing PDF. New or improved features in PDFlib 6 are marked.

Table 1.1 Feature list for PDFlib, PDFlib+PDI, and the PDFlib Personalization Server (PPS)

topic	features
PDF output	PDF documents of arbitrary length, directly in memory (for Web servers) or on disk file compression for text, vector graphics, image data, and file attachments suspend/resume <sup>1</sup> and insert page <sup>3</sup> features to create pages out of order
PDF flavors	PDF 1.3, 1.4, and 1.5 (Acrobat 4, 5, and 6) Linearized (web-optimized) PDF for byteserving over the Web <sup>1</sup>
PDF input	import pages from existing PDF documents (only PDFlib+PDI and PPS)
Blocks	PDF personalization with PDFlib blocks for text, image, and PDF data (only PPS) PDFlib Block plugin for Acrobat to create PDFlib blocks (only PPS), redesigned user interface <sup>1</sup>
Graphics	common vector graphics primitives: lines, curves, arcs, rectangles, etc. smooth shadings (color blends), pattern fills and strokes efficiently re-use text or vector graphics with templates explicit graphics state parameters for text knockout, overprinting etc. transparency (opacity) and blend modes layers <sup>1</sup> : optional page content which can selectively be enabled or disabled
Fonts	TrueType (ttf and ttc) and PostScript Type 1 fonts (pfb and pfa, plus lwfn on the Mac) OpenType fonts (ttf, otf) with PostScript or TrueType outlines AFM and PFM PostScript font metrics files font embedding directly use fonts which are installed on the Windows or Mac host system subsetting for TrueType and OpenType fonts user-defined (Type 3) fonts for bitmap fonts or custom logos
Text output	text output in different fonts; underlined, overlined, and strikeout text kerning for PostScript, TrueType, and OpenType fonts TrueType and OpenType glyph id addressing for advanced typesetting applications proportional widths for standard CJK fonts
Internationalization	Unicode for page content, hypertext <sup>1</sup> , and file names <sup>1</sup> ; UTF-8 and UCS-2 formats, little- and big-endian fully integrated handling of Unicode strings in COM, Java, .NET, REALbasic, Tcl support for a variety of encodings (international standards and vendor-specific code pages) fetch code pages from the system (Windows, IBM eServer iSeries and zSeries) standard CJK font and CMap support for Chinese, Japanese, and Korean text custom CJK fonts in the TrueType and OpenType formats with Unicode encoding embed Unicode information in PDF for correct text extraction in Acrobat
Images	embed BMP, GIF, PNG, TIFF <sup>1</sup> , JPEG, and CCITT raster images automatic detection of image file formats (file format sniffing) transparent (masked) images including soft masks image masks (transparent images with a color applied) colorize images with a spot color image interpolation (smooth images with low resolution)

Table 1.1 Feature list for PDFlib, PDFlib+PDI, and the PDFlib Personalization Server (PPS)

<b>topic</b>	<b>features</b>
Color	grayscale, RGB, CMYK, CIE L*a*b* color
	built-in PANTONE® and HKS® spot color tables
	user-defined spot colors
Color management	ICC-based color with ICC color profiles: honor embedded profiles in images, or apply external profiles to images
	rendering intent for text, graphics, and raster images
	default gray, RGB, and CMYK color spaces to remap device-dependent colors
Prepress	generate output conforming to PDF/X-1, PDF/X-1a, PDF/X-2 <sup>1</sup> , and PDF/X-3, including 2003 flavors <sup>1</sup>
	embed output intent ICC profile or reference standard output intent
	copy output intent from imported PDF documents (only PDFlib+PDI and PPS)
	create OPI 1.3 and OPI 2.0 information for imported images <sup>1</sup>
Formatting	separation information (PlateColor) <sup>1</sup>
	Textflow formatting <sup>1</sup> : format arbitrary amounts of text into one or more rectangular areas, using hyphenation, font and color changes, various justification methods, control commands
	text line placement and formatting
Security	flexible image placement and formatting
	generate output with 40-bit or 128-bit encryption
	generate output with permission settings
Hypertext	import encrypted documents (master password required; only PDFlib+PDI and PPS)
	create form fields <sup>1</sup> with all field options and JavaScript <sup>1</sup>
	create actions <sup>1</sup> for bookmarks, annotations, page open/close and other events
	create bookmarks <sup>1</sup> with a variety of options and controls
	page transition effects, such as shades and mosaic
	create all PDF annotation types <sup>1</sup> , such as PDF links, launch links (other document types), Web links
	document information: standard fields (Title, Subject, Author, Keywords) plus unlimited number of user-defined info fields
	named destinations for links, bookmarks, and document open action
Tagged PDF	viewer preferences (hide menu bar, etc.) <sup>1</sup>
	create page labels (symbolic names for pages) <sup>1</sup>
	create Tagged PDF <sup>1</sup> and structure information for accessibility, page reflow, and improved content repurposing
Programming	easily format large amounts of text for Tagged PDF <sup>1</sup>
	language bindings for Cobol, COM, C, C++, Java, .NET, Perl, PHP <sup>1</sup> , Python, REALbasic <sup>1</sup> , RPG, Tcl
	thread-safe and robust for deployment in multi-threaded server applications
	virtual file system for supplying data in memory, e.g., images from a database

1. New or considerably improved in PDFlib 6



# 1.4 Availability of Features in different Products

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

Table 1.2 Availability of features in different products

Feature	API functions and parameters	PDFlib Lite (open source)	PDFlib	PDFlib+PDI	PDFlib Personalization Server (PPS)
basic PDF generation	(all except those listed below)	X	X	X	X
language bindings	C, C++, Java, Perl, Tcl, PHP, Python	X	X	X	X
language bindings	Cobol, COM, .NET, REALbasic, RPG	–	X	X	X
works on EBCDIC systems		–	X	X	X
password protection and permission settings	PDF_begin_document() with userpassword, masterpassword, permissions options	–	X	X	X
linearized PDF	PDF_begin_document() with linearize option	–	X	X	X
font subsetting	PDF_load_font() with subsetting option	–	X	X	X
Kerning	PDF_load_font() with kerning option	–	X	X	X
access Mac and Windows host fonts	PDF_load_font()	–	X	X	X
access system encodings on Windows, iSeries, zSeries	PDF_load_font()	–	X	X	X
Unicode encoding and ToUnicode CMaps	PDF_load_font() with encoding = unicode, autocidfont, unicodemap parameters	–	X	X	X
numeric and character entity references	charref option in PDF_fit_textline() and PDF_create_textflow(), charref parameter	–	X	X	X
proportional glyph widths for standard CJK fonts with Unicode CMaps	PDF_load_font() with a UCS2-compatible CMap	–	X	X	X
glyph ID addressing	PDF_load_font() with encoding = glyphid	–	X	X	X
extended encoding for PostScript-based OpenType fonts	PDF_load_font()	–	X	X	X
Textflow	PDF_create_textflow(), PDF_delete_textflow(), PDF_fit_textflow(), PDF_info_textflow()	–	X	X	X
spot color	PDF_makespotcolor()	–	X	X	X
color separations	PDF_begin_page_ext() with separationinfo option	–	X	X	X
form fields	PDF_create_field(), PDF_create_fieldgroup(), PDF_create_action() with type=SetOCGState	–	X	X	X
JavaScript actions	PDF_create_action() with type=JavaScript	–	X	X	X
layers	PDF_define_layer(), PDF_begin_layer(), PDF_end_layer(), PDF_set_layer_dependency(), PDF_create_action() with type=SetOCGState	–	X	X	X
Tagged PDF	PDF_begin_item(), PDF_end_item(), PDF_activate_item(), PDF_begin_document() with tagged and lang options	–	X	X	X

Table 1.2 Availability of features in different products

Feature	API functions and parameters	PDFlib Lite (open source)	PDFlib	PDFlib+PDI	PDFlib Personalization Server (PPS)
PDF/X support	PDF_process_pdi(), PDF_begin_document() with pdfx option	–	X	X	X
ICC profile support	PDF_load_iccprofile(), PDF_setcolor() with icc-basedgray/rgb/cmyk, PDF_load_image() with honoriccprofile option, honoriccprofile parameter, PDF_begin/end_page_ext() with defaultgray/rgb/cmyk option	–	X	X	X
CIE L*a*b* color	PDF_setcolor() with type = lab; Lab TIFF images	–	X	X	X
OPI support	PDF_load_image() with OPI-1.3/OPI-2.0 options	–	X	X	X
PDF import (PDI)	PDF_open_pdi(), PDF_open_pdi_callback(), PDF_open_pdi_page(), PDF_fit_pdi_page(), PDF_process_pdi()	–	–	X	X
Query information from existing PDF	PDF_get_pdi_value(), PDF_get_pdi_parameter()	–	–	X	X
variable data processing and personalization with blocks	PDF_fill_textblock(), PDF_fill_imageblock(), PDF_fill_pdfblock()	–	–	–	X
query standard and custom block properties	PDF_get_pdi_value(), PDF_get_pdi_parameter() with vdp/Blocks keys	–	–	–	X
PDFlib Block plugin for Acrobat	interactively create PDFlib blocks for use with PPS	–	–	–	X

# 2 PDFlib Language Bindings

## 2.1 Overview

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

Table 2.1 Tested language and platform combinations

language	Unix (Linux, Solaris, HP-UX, Mac OS X, AIX, IRIX a.o.)	Windows NT4SP2 or above	IBM eServer iSeries and zSeries
Cobol	–	–	ILE Cobol
COM	–	ASP (PWS, IIS 4, 5, 6) WSH (VBScript 5, JScript 5) Visual Basic 6.0, Borland Delphi 5–7	–
ISO/ANSI C	gcc 2/3, HP C, IBM C 6, Sun Workshop 6, and other ISO C compilers	Microsoft Visual C++ 6, VS .NET Metrowerks CodeWarrior 8 Borland C++ Builder 5	IBM c89 SAS C for MVS
ISO C++	gcc 2/3 and other ISO C++ compilers	Microsoft Visual C++ 6, VS .NET Metrowerks CodeWarrior 8	IBM c89
Java	JDK 1.1.8, 1.2.2, 1.3, 1.4	Sun JDK 1.1.8, 1.2.2, 1.3, 1.4 ColdFusion MX	JDK 1.3.1
.NET	–	.NET Framework 1.0, 1.1: C#, VB.NET, ASP.NET	–
Perl	Perl 5.6–5.8	Perl 5.6–5.8	–
PHP	PHP 4.3.x, 5.0	PHP 4.3.x, 5.0	–
Python	Python 1.6, 2.0–2.3	Python 1.6, 2.0–2.3	–
REALbasic	REALbasic 5.5 or above for Mac OS Classic, Mac OS X, and Windows		–
RPG	–	–	ILE RPG
Tcl	Tcl 8.3.2 and 8.4.4	Tcl 8.3.2 and 8.4.4	–

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

## 2.2 Cobol Binding

### 2.2.1 Special Considerations for Cobol

The PDFlib API functions for Cobol are not available under the names documented in Chapter 8, but use abbreviated function names instead. The short function names are not documented here, but can be found in a separate cross-reference listing (*xref.txt*). For example, instead of using *PDF\_load\_font()* the short form *PDLODFNT* must be used.

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

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

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

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

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

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

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

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

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

### 2.2.2 The »Hello world« Example in Cobol

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

IDENTIFICATION DIVISION.  
PROGRAM-ID. HELLO.

ENVIRONMENT DIVISION.

DATA DIVISION.  
WORKING-STORAGE SECTION.

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

01 PDFLIB-CALL-AREA.  
05 P PIC S9(9) BINARY.  
05 RC PIC S9(9) BINARY.  
05 PDFLIB-RETURN-LONG PIC S9(9) BINARY.  
05 PDFLIB-RETURN-CHAR PIC X(64) VALUE SPACES.  
05 PDFLIB-ERR-STRING PIC X(128).

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

PROCEDURE DIVISION.

```
* * * * *
* CREATE A PDF OBJECT *
* * * * *
CALL "PDNEW" USING P,
                  RC.
* * * * *
* OPEN NEW PDF DOCUMENT *
* * * * *
MOVE 0 TO WS-INT.

STRING Z'HELLO.PDF'
      DELIMITED BY SIZE INTO WS-STRING.

CALL "PDBEGDOC" USING P,
                    WS-STRING,
                    WS-INT,
                    WS-NULL,
                    PDFLIB-RETURN-LONG,
                    RC.

IF PDFLIB-RETURN-LONG = -1
    CALL "PDERRMSG" USING P,
                        PDFLIB-ERR-STRING,
                        RC
    DISPLAY PDFLIB-ERR-STRING
    MOVE +8 TO RETURN-CODE
    GOBACK.
* * * * *
* SET PDF INFORMATION *
```

```

* * * * *
STRING Z'Creator'
    DELIMITED BY SIZE INTO WS-STRING.
STRING Z'Hello.cbl'
    DELIMITED BY SIZE INTO WS-STRING2.

CALL "PDSETINF" USING      P,
                           WS-STRING,
                           WS-STRING2,
                           RC.

STRING Z'Author'
    DELIMITED BY SIZE INTO WS-STRING.
STRING Z'Thomas Merz'
    DELIMITED BY SIZE INTO WS-STRING2.

CALL "PDSETINF" USING      P,
                           WS-STRING
                           WS-STRING2,
                           RC.

STRING Z'Title'
    DELIMITED BY SIZE INTO WS-STRING.
STRING Z'Hello, world (COBOL)!'
    DELIMITED BY SIZE INTO WS-STRING2.

CALL "PDSETINF" USING      P,
                           WS-STRING
                           WS-STRING2,
                           RC.
* * * * *
*   BEGIN A NEW PAGE                                           *
* * * * *
CALL "PDBEGPAG" USING      P,
                           PDFLIB-A4-WIDTH,
                           PDFLIB-A4-HEIGHT,
                           WS-NULL,
                           RC.
* * * * *
*   LOAD & SET THE CURRENT FONT                               *
* * * * *
MOVE 0 TO WS-INT.

STRING Z'Helvetica-Bold'
    DELIMITED BY SIZE INTO WS-STRING.
STRING Z'ebcdic'
    DELIMITED BY SIZE INTO WS-STRING2.

CALL "PDLODFNT" USING      P,
                           WS-STRING
                           WS-INT,
                           WS-STRING2,
                           WS-NULL,
                           PDFLIB-RETURN-LONG,
                           RC.

MOVE PDFLIB-RETURN-LONG    TO WS-FONT.

```

```

MOVE 24 TO WS-FLOAT.

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

CALL "PDSETTP" USING      P,
                          WS-FLOAT,
                          WS-FLOAT2,
                          RC.

STRING Z'Hello, World!'
      DELIMITED BY SIZE INTO WS-STRING.

CALL "PDSHOW" USING      P,
                          WS-STRING,
                          RC.

STRING Z'(says COBOL)'
      DELIMITED BY SIZE INTO WS-STRING.

CALL "PDCONT" USING      P,
                          WS-STRING,
                          RC.
* * * * *
* END THIS PAGE *
* * * * *
CALL "PDENDPAG" USING      P,
                          WS-NULL,
                          RC.
* * * * *
* END THE PDF DOCUMENT *
* * * * *
CALL "PDENDDOC" USING      P,
                          WS-NULL,
                          RC.
* * * * *
* DELETE THE PDF OBJECT *
* * * * *
CALL "PDDELETE" USING      P,
                          RC.
* * * * *
* END THE PROGRAM *
* * * * *
MOVE ZERO                      TO RETURN-CODE.
GOBACK.

END PROGRAM HELLO.

```

## 2.3 COM Binding

(This section is only included in the COM/.NET/REALbasic edition of the PDFlib manual.)

## 2.4 C Binding

### 2.4.1 Availability and Special Considerations for C

PDFlib itself is written in the ANSI C language. In order to use the PDFlib C binding, you can use a static or shared library (DLL on Windows and MVS), and you need the central PDFlib include file *pdflib.h* for inclusion in your PDFlib client source modules. Alternatively, *pdflibdl.h* can be used for dynamically loading the PDFlib DLL at runtime (see Section 2.4.3, »Using PDFlib as a DLL loaded at Runtime«, page 25).

### 2.4.2 The »Hello world« Example in C

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

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

#include "pdflib.h"

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

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

    PDF_TRY(p) {
        if (PDF_begin_document(p, "hello.pdf", 0, "") == -1) {
            printf("Error: %s\n", PDF_get_errmsg(p));
            return(2);
        }

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

        PDF_begin_page_ext(p, a4_width, a4_height, "");

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

        PDF_setfont(p, font, 24);
        PDF_set_text_pos(p, 50, 700);
        PDF_show(p, "Hello, world!");
        PDF_continue_text(p, "(says C)");
        PDF_end_page_ext(p, "");
    }
```



```

        PDF_end_document(p, "");
    }

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

    PDF_delete(p);

    return 0;
}

```

### 2.4.3 Using PDFlib as a DLL loaded at Runtime

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

- ▶ Include *pdflibdl.h* instead of *pdflib.h*.
- ▶ Use *PDF\_new\_dl()* and *PDF\_delete\_dl()* instead of *PDF\_new()* and *PDF\_delete()*.
- ▶ Use *PDF\_TRY\_DL()* and *PDF\_CATCH\_DL()* instead of *PDF\_TRY()* and *PDF\_CATCH()*.
- ▶ Use function pointers for all other PDFlib calls.
- ▶ *PDF\_get\_opaque()* must not be used.
- ▶ Compile the auxiliary module *pdflibdl.c* and link your application against it.

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

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

```

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

#include "pdflibdl.h"

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

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

    PDF_TRY_DL(PDFlib, p) {
        if (PDFlib->PDF_begin_document(p, "hellodl.pdf", 0, "") == -1) {

```

```

        printf("Error: %s\n", PDFlib->PDF_get_errmsg(p));
        return(2);
    }

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

    PDFlib->PDF_begin_page_ext(p, a4_width, a4_height, "");

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

    PDFlib->PDF_setfont(p, font, 24);
    PDFlib->PDF_set_text_pos(p, 50, 700);
    PDFlib->PDF_show(p, "Hello, world!");
    PDFlib->PDF_continue_text(p, "(says C DLL)");
    PDFlib->PDF_end_page_ext(p, "");

    PDFlib->PDF_end_document(p, "");
}

PDF_CATCH_DL(PDFlib, p) {
    printf("PDFlib exception occurred in hellodl sample:\n");
    printf("[%d] %s: %s\n",
        PDFlib->PDF_get_errnum(p), PDFlib->PDF_get_apiname(p),
        PDFlib->PDF_get_errmsg(p));
    PDF_delete_dl(PDFlib, p);
    return(2);
}

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

return 0;
}

```

#### 2.4.4 Error Handling in C

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

```

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

```

```
PDF_delete(p);
```

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

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

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

For C and C++ clients which do not catch exceptions, the default action upon exceptions is to issue an appropriate message on the standard error channel, and exit on fatal errors. The PDF output file will be left in an incomplete state! Since this may not be adequate for a library routine, for serious PDFlib projects it is strongly advised to leverage PDFlib's exception handling facilities. A user-defined catch clause may, for example, present the error message in a GUI dialog box, and take other measures instead of aborting.

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

## 2.4.5 Memory Management in C

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

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

The signatures of the memory routines can be found in Section 8.2, »General Functions«, page 187. These routines must adhere to the standard C `malloc/free/realloc` semantics, but may choose an arbitrary implementation. All routines will be supplied with a pointer to the calling PDFlib object. The only exception to this rule is that the very first call to the allocation routine will supply a PDF pointer of NULL. Client-provided memory allocation routines must therefore be prepared to deal with a NULL PDF pointer.

Using the *PDF\_get\_opaque()* function, an opaque application specific pointer can be retrieved from the PDFlib object. The opaque pointer itself is supplied by the client in the *PDF\_new2()* call. The opaque pointer is useful for multi-threaded applications which may want to keep a pointer to thread- or class specific data inside the PDFlib object, for use in memory management or error handling.

## 2.4.6 Unicode in the C language binding

Clients of the C language binding must take care not to use the standard text output functions (*PDF\_show()*, *PDF\_show\_xy()*, and *PDF\_continue\_text()*) when the text may contain embedded null characters. In such cases the alternate functions *PDF\_show2()* etc. must be used, and the length of the string must be supplied separately. This is not a concern for all other language bindings since the PDFlib language wrappers internally call *PDF\_show2()* etc. in the first place.

## 2.5 C++ Binding

### 2.5.1 Availability and Special Considerations for C++

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

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

### 2.5.2 The »Hello world« Example in C++

```
#include <iostream>

#include "pdflib.hpp"

int
main(void)
{
    try {
        int font;
        PDFlib p;

        if (p.begin_document("hello.pdf", "") == -1) {
            cerr << "Error: " << p.get_errmsg() << endl;
            return 2;
        }

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

        p.begin_page_ext((float) a4_width, (float) a4_height, "");

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

```

        p.setfont(font, 24);
        p.set_text_pos(50, 700);
        p.show("Hello, world!");
        p.continue_text("(says C++)");
        p.end_page_ext("");

        p.end_document("");
    }

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

    return 0;
}

```

### 2.5.3 Error Handling in C++

PDFlib API functions will throw a C++ exception in case of an error. These exceptions must be caught in the client code by using C++ *try/catch* clauses. In order to provide extended error information the PDFlib class provides a public *PDFlib::Exception* class which exposes methods for retrieving the detailed error message, the exception number, and the name of the PDFlib API function which threw the exception.

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

```

try {
    ...some PDFlib instructions...
} catch (PDFlib::Exception &ex) {
    cerr << "PDFlib exception occurred in hello sample: " << endl;
    cerr << "[" << ex.get_errnum() << "]" " << ex.get_apiname()
        << ": " << ex.get_errmsg() << endl;
    return 2;
}

```

### 2.5.4 Memory Management in C++

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

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

### 2.5.5 Unicode in the C++ language binding

C++ users must be aware of a pitfall related to the compiler automatically converting literal strings to the C++ string type which is expected by the PDFlib API functions: this conversion supports embedded null characters only if an explicit length parameter is

supplied. For example, the following will not work since the string will be truncated at the first null character:

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

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

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

## 2.6 Java Binding

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

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

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

### 2.6.1 Installing the PDFlib Java Edition

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

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

```
com.pdflib.pdflib
```

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

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

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

You can check the value of this property as follows:

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

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

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

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

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

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

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

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

## 2.6.2 The »Hello world« Example in Java

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

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

        try{
            p = new pdflib();

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

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

            p.begin_page_ext(595, 842, "");

            font = p.load_font("Helvetica-Bold", "unicode", "");
```

```

        p.setFont(font, 18);

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

        p.end_document("");

    } catch (PDFlibException e) {
        System.err.print("PDFlib exception occurred in hello sample:\n");
        System.err.print("[ " + e.get_errnum() + " ] " + e.get_apiname() +
            ": " + e.get_errmsg() + "\n");
    } catch (Exception e) {
        System.err.println(e.getMessage());
    } finally {
        if (p != null) {
            p.delete();
        }
    }
}
}

```

### 2.6.3 Error Handling in Java

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

PDFlibException

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

```

try {

...some PDFlib instructions...

} catch (PDFlibException e) {
    System.err.print("PDFlib exception occurred in hello sample:\n");
    System.err.print("[ " + e.get_errnum() + " ] " + e.get_apiname() +
        ": " + e.get_errmsg() + "\n");
} catch (Exception e) {
    System.err.println(e.getMessage());
} finally {
    if (p != null) {
        p.delete();                /* delete the PDFlib object */
    }
}

```

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



## 2.7 .NET Binding

(This section is only included in the COM/.NET/REALbasic edition of the PDFlib manual.)

## 2.8 Perl Binding

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

### 2.8.1 Installing the PDFlib Perl Edition

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

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

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

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

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

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

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

**Windows.** PDFlib supports the ActiveState port of Perl 5 to Windows, also known as ActivePerl.<sup>2</sup> Both *pdflib\_pl.dll* and *pdflib\_pl.pm* will be searched in the current directory, or the directory printed by the following Perl command:

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

Typical output of the above command looks like

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

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

```
use pdflib_pl 6.0;
```

```
$p = PDF_new();
```

```
eval {
```

<sup>1</sup> See [www.perl.com](http://www.perl.com)

<sup>2</sup> See [www.activestate.com](http://www.activestate.com)

```

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

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

PDF_begin_page_ext($p, 595, 842, "");

$font = PDF_load_font($p, "Helvetica-Bold", "winansi", "");

PDF_setfont($p, $font, 24.0);
PDF_set_text_pos($p, 50, 700);
PDF_show($p, "Hello world!");
PDF_continue_text($p, "(says Perl)");
PDF_end_page_ext($p, "");

PDF_end_document($p, "");
};

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

PDF_delete($p);

```

### 2.8.3 Error Handling in Perl

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

```

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

```

## 2.9 PHP Binding

### 2.9.1 Installing the PDFlib PHP Edition

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

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

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

1. See [www.php.net](http://www.php.net)

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

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

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

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

This will display a long info page about your current PHP configuration. On this page check the section titled *pdf*. If this section contains *PDFlib GmbH Version* (and the PDFlib version number) you are using the supported new PDFlib wrapper. The unsupported old wrapper will display *PDFlib Version* instead.

- Load PDFlib at runtime with one of the following lines at the start of your script:

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

**PHP 5 features.** PDFlib takes advantage of the following new features in PHP 5:

- New object model: the PDFlib functions are encapsulated within a PDFlib object.
- Exceptions: PDFlib exceptions will be propagated as PHP 5 exceptions, and can be caught with the usual try/catch technique. New-style exception handling can be used with both the new object-oriented approach and the old API functions.

See below for more details on these PHP 5 features.

**Modified error return for PDFlib functions in PHP.** Since PHP uses the convention of returning the value 0 (FALSE) when an error occurs within a function, all PDFlib functions have been adjusted to return 0 instead of -1 in case of an error. This difference is noted in the function descriptions in Chapter 8. However, take care when reading the code fragment examples in Section 3, »PDFlib Programming«, page 45 since these use the usual PDFlib convention of returning -1 in case of an error.

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

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

In order to provide platform-independent file name handling use of PDFlib's Search-Path facility (see Section 3.1.6, »Resource Configuration and File Searching«, page 51) is strongly recommended.

## 2.9.2 The »Hello world« Example in PHP

**Example for PHP 4.** The following sample works with PHP 4:

```

<?php
$p = PDF_new();

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

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

PDF_begin_page_ext($p, 595, 842, "");

$font = PDF_load_font($p, "Helvetica-Bold", "winansi", "");

PDF_setfont($p, $font, 24.0);
PDF_set_text_pos($p, 50, 700);
PDF_show($p, "Hello world!");
PDF_continue_text($p, "(says PHP)");
PDF_end_page_ext($p, "");

PDF_end_document($p, "");

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

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

PDF_delete($p);
?>

```

**Example for PHP 5.** The following sample uses the new exception handling and object encapsulation features available in PHP 5:

```

<?php

try {
    $p = new PDFlib();

    /* open new PDF file; insert a file name to create the PDF on disk */
    if ($p->begin_document("", "") == 0) {
        die("Error: " . $p->get_errmsg());
    }

    $p->set_info("Creator", "hello.php");
    $p->set_info("Author", "Rainer Schaaf");
    $p->set_info("Title", "Hello world (PHP)!");

    $p->begin_page_ext(595, 842, "");

    $font = $p->load_font("Helvetica-Bold", "winansi", "");

    $p->setfont($font, 24.0);
    $p->set_text_pos(50, 700);

```

```

    $p->show("Hello world!");
    $p->continue_text("(says PHP)");
    $p->end_page_ext("");

    $p->end_document("");

    $buf = $p->get_buffer();
    $len = strlen($buf);

    header("Content-type: application/pdf");
    header("Content-Length: $len");
    header("Content-Disposition: inline; filename=hello.pdf");
    print $buf;
}
catch (PDFlibException $e) {
    die("PDFlib exception occurred in hello sample:\n" .
        "[" . $e->get_errnum() . "] " . $e->get_apiname() . ": " .
        $e->get_errmsg() . "\n");
}
catch (Exception $e) {
    die($e);
}
$p = 0;
?>

```

### 2.9.3 Error Handling in PHP

**Error handling in PHP 4.** When a PDFlib exception occurs, a PHP exception is thrown. Since PHP 4 does not support structured exception handling there is no way to catch exceptions and act appropriately. Do not disable PHP warnings when using PDFlib, or you will run into serious trouble.

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

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

**Exception handling in PHP 5.** Since PHP 5 supports structured exception handling, PDFlib exceptions will be propagated as PHP exceptions. You can use the standard try/catch technique to deal with PDFlib exceptions:

```

try {
    ...some PDFlib instructions...
} catch (PDFlibException $e) {
    print "PDFlib exception occurred:\n";
    print "[" . $e->get_errnum() . "] " . $e->get_apiname() . ": " .
        $e->get_errmsg() . "\n";
}
catch (Exception $e) {
    print $e;
}

```

Note that you can use PHP 5-style exception handling regardless of whether you work with the old function-based PDFlib interface, or the new object-oriented one.

## 2.10 Python Binding

### 2.10.1 Installing the PDFlib Python Edition

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

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

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

### 2.10.2 The »Hello world« Example in Python

```
from sys import *
from pdflib_py import *

p = PDF_new()

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

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

PDF_begin_page_ext(p, 595, 842, "")

font = PDF_load_font(p, "Helvetica-Bold", "winansi", "")

PDF_setfont(p, font, 24)
PDF_set_text_pos(p, 50, 700)
PDF_show(p, "Hello world!")
PDF_continue_text(p, "(says Python)")
PDF_end_page_ext(p, "")

PDF_end_document(p, "")

PDF_delete(p)
```

### 2.10.3 Error Handling in Python

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

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

<sup>1</sup> See [www.python.org](http://www.python.org)

# 2.11 REALbasic Binding<sup>1</sup>

(This section is only included in the COM/.NET/REALbasic edition of the PDFlib manual.)

## 2.12 RPG Binding

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

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

### 2.12.1 Compiling and Binding RPG Programs for PDFlib

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

```
d/copy QRPGLSRC,PDFLIB
```

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

```
d/copy PDFsrcLib/QRPGLSRC,PDFLIB
```

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

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

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

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

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

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

### 2.12.2 The »Hello world« Example in RPG

```
*****
d/copy QRPGLSRC,PDFLIB
*****

d p                S                *
d font             S                10i 0
*
d error            S                50
d errmsg_p         S                *
```

1. See [www.realbasic.com](http://www.realbasic.com)

```

d errmsg      s      200    based(errmsg_p)
*
d filename    s      256
d fontname    s      50
d fontenc     s      50
d infokey     s      50
d infoval     s      200
d text        s      200
d n           s      1      inz(x'00')
d empty       s      1      inz(x'00')
*****
c              clear          error
*
*   Init on PDFlib
c              eval          p=pdf_new
c              if            p=NULL
c              eval          error='Couldn't create PDFlib object '+
c                          '(out of memory)!'
c              exsr          exit
c              endif
*
*   Open new pdf file
c              eval          filename='hello.pdf'+x'00'
c              if            PDF_begin_document(p:filename:0:empty) = -1
c              exsr          geterrmsg
c              exsr          exit
c              endif
*
*   This is required to avoid problems on Japanese systems
c              eval          infokey='hypertextencoding'+x'00'
c              eval          infoval='ebcdic'+x'00'
c              callp          PDF_set_parameter(p:infokey:infoval)
*
*   Set info "Creator"
c              eval          infokey='Creator'+x'00'
c              eval          infoval='hello.rpg'+x'00'
c              callp          PDF_set_info(p:infokey:infoval)
*
*   Set info "Author"
c              eval          infokey='Author'+x'00'
c              eval          infoval='Thomas Merz'+x'00'
c              callp          PDF_set_info(p:infokey:infoval)
*
*   Set info "Title"
c              eval          infokey='Title'+x'00'
c              eval          infoval='Hello, world (RPG)'+x'00'
c              callp          PDF_set_info(p:infokey:infoval)
c              callp          PDF_begin_page_ext(p:a4_width:a4_height:
c                                          empty)
*
c              eval          fontname='Helvetica-Bold'+x'00'
c              eval          fontenc='ebcdic'+x'00'
c              eval          font=PDF_load_font(p:fontname:0:fontenc:n)
*
c              callp          PDF_setfont(p:font:24)
c              callp          PDF_set_text_pos(p:50:700)
*
c              eval          text='Hello world!'+x'00'
c              callp          PDF_show(p:text)
c              eval          text='(says ILE RPG)'+x'00'
c              callp          PDF_continue_text(p:text)
c              callp          PDF_end_page_ext(p:empty)

```



```

c          callp      PDF_end_document(p:empty)
c          callp      PDF_delete(p)
*
c          exsr       exit
*****
c      geterrmsg      begsr
c                      eval      errmsg_p=PDF_get_errmsg(p)
c                      if        errmsg_p<>*NULL
c                      eval      error=%subst(errmsg:1:%scan(x'00':errmsg)-1)
c                      endif
c                      endsr
*****
c      exit           begsr
c                      if        error<>*blanks
c                      eval      error='Error: ' +error
c      error          dsply
c                      endif
c                      seton
c                      return
c                      endsr

```

You can compile this program as follows:

```

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

```

### 2.12.3 Error Handling in RPG

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

```

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

```

```

c          eval      p=pdf_new2(errhdl:*null:*null:*null:*null)

...PDFlib instructions...

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

```

## 2.13 Tcl Binding

### 2.13.1 Installing the PDFlib Tcl Edition

The Tcl<sup>1</sup> extension mechanism works by loading shared libraries at runtime. For the PDFlib binding to work, the Tcl shell must have access to the PDFlib Tcl wrapper shared library and the package index file *pkgIndex.tcl*. You can use the following idiom in your script to make the library available from a certain directory (this may be useful if you want to deploy PDFlib on a machine where you don't have root privilege for installing PDFlib):

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

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

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

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

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

### 2.13.2 The »Hello world« Example in Tcl

```
package require pdflib 6.0

set p [PDF_new]

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

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

PDF_begin_page_ext $p 595 842 ""

set font [PDF_load_font $p "Helvetica-Bold" "unicode" ""]

PDF_setfont $p $font 24.0
PDF_set_text_pos $p 50 700
PDF_show $p "Hello world!"
PDF_continue_text $p "(says Tcl)"
PDF_end_page_ext $p ""

PDF_end_document $p ""

PDF_delete $p
```

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

### 2.13.3 Error Handling in Tcl

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

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

# 3 PDFlib Programming

## 3.1 General Programming

### 3.1.1 PDFlib Program Structure and Function Scopes

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

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

The function descriptions in Chapter 8 reference these scopes; the scope definitions can be found in Table 3.1. Figure 3.1 depicts the nesting of scopes. PDFlib will throw an exception if a function is called outside the allowed scope. You can query the current scope with the *scope* parameter.

Table 3.1 Function scope definitions

scope name	definition
path	started by one of <code>PDF_moveto()</code> , <code>PDF_circle()</code> , <code>PDF_arc()</code> , <code>PDF_arcn()</code> , or <code>PDF_rect()</code> ; terminated by any of the functions in Section 8.4.6, »Path Painting and Clipping«, page 233
page	between <code>PDF_begin_page()</code> and <code>PDF_end_page()</code> , but outside of path scope
template	between <code>PDF_begin_template()</code> and <code>PDF_end_template()</code> , but outside of path scope
pattern	between <code>PDF_begin_pattern()</code> and <code>PDF_end_pattern()</code> , but outside of path scope
font	between <code>PDF_begin_font()</code> and <code>PDF_end_font()</code> , but outside of glyph scope
glyph	between <code>PDF_begin_glyph()</code> and <code>PDF_end_glyph()</code> , but outside of path scope
document	between <code>PDF_begin_document()</code> and <code>PDF_end_document()</code> , but outside of page, template, pattern, and font scope
object	in Java: the lifetime of the pdflib object, but outside of document scope; in other bindings between <code>PDF_new()</code> and <code>PDF_delete()</code> , but outside of document scope
null	outside of object scope
any	when a function description mentions »any« scope it actually means any except null, since a PDFlib object doesn't even exist in null scope.

### 3.1.2 Parameters

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

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

Details of parameter names and possible values can be found in Chapter 8.

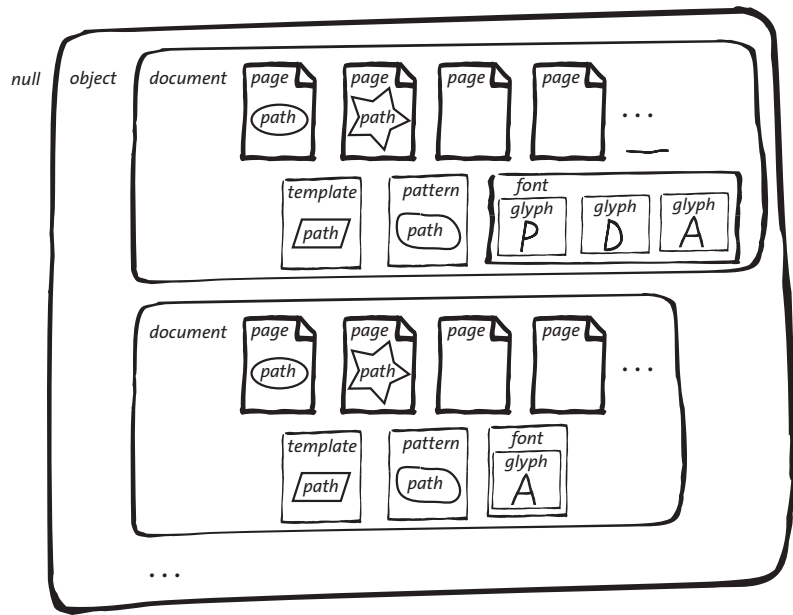


Fig. 3.1  
Nesting of scopes

### 3.1.3 Exception Handling

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

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

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

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

When PDFlib detects such a situation, an exception will be thrown instead of passing a special error return value to the caller. In the C programming language, which doesn't natively support exceptions, the client can install a custom routine (called an error handler) which will be called in case of an exception. However, the recommended method is to make use of `PDF_TRY()/PDF_CATCH()` blocks as detailed in Section 2.4.4, »Error Handling in C«, page 26.

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

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

Table 3.2 Ranges of PDFlib exception numbers

error ranges	reasons
1000 – 1999	(PDCORE library): memory, I/O, arguments, parameters/values, options
2000 – 3999	(PDFlib library): configuration, scoping, graphics and text, color, images, fonts, encodings, PDF/X, hypertext, Tagged PDF, layers
4000 – 4999	(PDF import library PDI): configuration and parameter, corrupt PDF (file, object, or stream level)

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

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

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

In addition to the global *warning* parameter, some functions also support dedicated options for enabling or disabling warnings for individual function calls. The suggested strategy is to enable warnings during the development cycle (and closely examine possible warnings), and disable warnings in a production system.

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

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

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

after a failed function call, i.e., a function call which returned with a -1 (in PHP: o) error value.

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

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

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

If the *fontwarning* parameter or option is *false* the return value must be checked for validity. If it indicates failure, the reason of the failure can be queried in order to properly deal with the situation:

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

### 3.1.4 Option Lists

Option lists are a powerful yet easy method to control PDFlib operations. Instead of requiring a multitude of function parameters, many PDFlib API methods support option lists, or optlists for short. These are strings which may contain an arbitrary number of options. Since option lists will be evaluated from left to right an option can be supplied multiply within the same list; in this case the last occurrence will overwrite earlier ones. Optlists support various data types and composite data like arrays. In most languages optlists can easily be constructed by concatenating the required keywords and values. C programmers may want to use the *sprintf()* function in order to construct optlists.

An optlist is a string containing one or more pairs of the form

name value(s)

Names and values, as well as multiple name/value pairs can be separated by arbitrary whitespace characters (space, tab, carriage return, newline). The value may be a list of multiple values. You can also use an equal sign '=' between name and value:

name=value

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

- ▶ Boolean: *true* or *false*; if the value of a boolean option is omitted, the value *true* is assumed. As a shorthand notation *none* can be used instead of *name=false*.
- ▶ String: these are plain ASCII strings which are generally used for non-localizable keywords. Strings containing whitespace must be bracketed with { and }. An empty



string can be constructed with `{}`. The characters `{`, `}`, and `\` must be preceded by an additional `\` character if they are supposed to be part of the string.

- ▶ Content strings, hypertext strings and name strings: these can hold Unicode content in various formats; for details on these string types see Section 4.5, »Unicode Support«, page 89).
- ▶ Keyword: one of a predefined list of fixed keywords
- ▶ Float and integer: decimal floating point or integer numbers; point and comma can be used as decimal separators for floating point values. Integer values can start with `x`, `X`, `0x`, or `0X` to specify hexadecimal values. Some options (this is stated in the respective documentation) support percentages by adding a `%` character directly after the value.
- ▶ Handle: several PDFlib-internal object handles, e.g., font, image, or action handles. Technically these are integer values.

Depending on the type and interpretation of an option additional restrictions may apply. For example, integer or float options may be restricted to a certain range of values; handles must be valid for the corresponding type of object, etc. Conditions for options are documented in their respective function descriptions in Chapter 8. Some examples for simple values (the first line shows a password string containing a blank character):

```
PDF_open_pdi():      password {secret string}
PDF_create_gstate(): linewidth 0.5 blendmode overlay opacityfill 0.75
PDF_load_font():     embedding=true subsetting=true subsetlimit=50 kerning=false
PDF_load_font():     embedding subsetting subsetlimit=50 nokerning
PDF_create_textflow() leading=150%
PDF_create_textflow() charmapping={ 0x0A 0x20 }
```

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

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

**Rectangles.** A rectangle is a list of four float values specifying the coordinates of the lower left and upper right corners of a rectangle. The coordinate system for interpreting the rectangle coordinates (standard or user coordinate system) is specified separately for the respective option. Example:

```
PDF_begin_document(): cropbox {0 0 500 600}
```

**Action lists.** An action list specifies one or more actions. Each entry in the list consists of an event keyword (trigger) and a list of action handles which must have been created with `PDF_create_action()`. Actions will be performed in the listed order. The set of allowed events (e.g. `docopen`) and the type of actions (e.g. JavaScript) are specified separately for the respective option. Example (assuming the values 0, 1, and 2 have been returned by earlier calls to `PDF_create_action()`):

```
PDF_begin_document():  action {open 0}
PDF_create_bookmark(): action {activate {0 1 2}}
```

**Color values.** Color values are lists consisting of a color space keyword and a list with a variable number of float values depending on the particular color space. The color space

keywords and semantics are the same as in *PDF\_setcolor()* (see Section 8.5.1, »Setting Color and Color Space«, page 237):

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

As detailed in the respective function descriptions in Chapter 8, a particular option list may only supply a subset of the keywords presented above. Some examples for color values:

```
PDF_fill_textblock(): strokecolor={ rgb 1 0 0 }  
PDF_fill_textblock(): bordercolor=none  
PDF_fill_textblock(): fillcolor={ spotname {PANTONE 281 U} 0.5 }
```

### 3.1.5 The PDFlib Virtual File System (PVF)

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

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

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

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

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

longer locked by PDFlib. PDFlib will automatically delete all virtual files in *PDF\_delete()*. However, the actual file contents (the data comprising a virtual file) must always be freed by the client.

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

- ▶ Minimize memory usage: it is recommended to call *PDF\_delete\_pvf()* immediately after the API call which accepted the virtual file name, and another time after *PDF\_close()*. The second call is required because PDFlib may still need access to the data so that the first call refuses to unlock the virtual file. However, in some cases the first call will already free the data, and the second call doesn't do any harm. The client may free the file contents only when *PDF\_delete\_pvf()* succeeded.
- ▶ Optimize performance by reusing virtual files: some clients may wish to reuse some data (e.g., font definitions) within various output documents, and avoid multiple create/delete cycles for the same file contents. In this case it is recommended not to call *PDF\_delete\_pvf()* as long as more PDF output documents using the virtual file will be generated.
- ▶ Lazy programming: if memory usage is not a concern the client may elect not to call *PDF\_delete\_pvf()* at all. In this case PDFlib will internally delete all pending virtual files in *PDF\_delete()*.

In all cases the client may free the corresponding data only when *PDF\_delete\_pvf()* returned successfully, or after *PDF\_delete()*.

### 3.1.6 Resource Configuration and File Searching

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

**Resource categories.** The resource categories supported by PDFlib are listed in Table 3.3. Other resource categories may be present in the UPR file for compatibility with Display PostScript installations, but they will silently be ignored. Redundant resource entries should be avoided. For example, do not include multiple entries for a certain font's metrics data. Also, the font name as configured in the UPR file

Table 3.3 Resource categories supported in PDFlib

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

should exactly match the actual font name in order to avoid confusion (although PDFlib does not enforce this restriction).

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

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

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

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

UPR files consist of the following components:

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

```
PS-Resources-1.0
```

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

**File searching and the SearchPath resource category.** PDFlib reads a variety of data items, such as raster images, font outline and metrics information, encoding definitions, PDF documents, and ICC color profiles from disk files. In addition to relative or ab-

solute path names you can also use file names without any path specification. The *SearchPath* resource category can be used to specify a list of path names for directories containing the required data files. When PDFlib must open a file it will first use the file name exactly as supplied and try to open the file. If this attempt fails PDFlib will try to open the file in the directories specified in the *SearchPath* resource category one after another until it succeeds. *SearchPath* entries can be accumulated, and will be searched in reverse order (paths set at a later point in time will be searched before earlier ones). This feature can be used to separate the PDFlib application from platform-specific file system schemes. In order to disable the search you can use a fully specified path name in the PDFlib functions.

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

```
HKLM\SOFTWARE\PDFlib\PDFlib\6.0.0\SearchPath
```

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

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

```
/pdflib/6.0.0/fonts  
/pdflib/6.0.0/bind/data
```

On MVS the *SearchPath* feature is not supported.

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

```
PS-Resources-1.0  
SearchPath  
FontAFM  
FontPFM  
FontOutline  
Encoding  
ICCPProfile  
.  
SearchPath  
/usr/local/lib/fonts  
Classic:Data:Fonts  
C:/psfonts/pfm  
C:/psfonts  
/users/kurt/my_images  
.  
FontAFM  
Code-128=Code_128.afm  
.  
FontPFM  
Foobar-Bold=foobb____.pfm  
  
Mistral=c:/psfonts/pfm/mist____.pfm  
.  
FontOutline  
Code-128=Code_128.pfa  
ArialMT=Arial.ttf  
.
```

```
Encoding
myencoding=myencoding.enc
.
ICCPProfile
highspeedprinter=cmkyhighspeed.icc
.
```

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

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

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

```
upr (on MVS; a dataset is expected)
pdflib/<version>/fonts/pdflib.upr (on IBM eServer iSeries)
pdflib.upr (Windows, Unix, and all other systems)
```

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

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

```
HKLM\SOFTWARE\PDFlib\PDFlib\6.0.0\resourcefile
```

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

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

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

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

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

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

### 3.1.7 Generating PDF Documents in Memory

In addition to generating PDF documents on a file, PDFlib can also be instructed to generate the PDF directly in memory (*in-core*). This technique offers performance benefits since no disk-based I/O is involved, and the PDF document can, for example, directly be

streamed via HTTP. Webmasters will be especially happy to hear that their server will not be cluttered with temporary PDF files.

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

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

```
PDF_begin_document(p, "", 0, "")
...create document...
PDF_end_document(p);

buf = PDF_get_buffer(p, &size);
... use the PDF data contained in the buffer ...
PDF_delete(p);
```

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

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

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

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

### 3.1.8 Using PDFlib on EBCDIC-based Platforms

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

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

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

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

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

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



## 3.2 Page Descriptions

### 3.2.1 Coordinate Systems

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Rotating objects.** It is important to understand that objects cannot be modified once they have been drawn on the page. Although there are PDFlib functions for rotating, translating, scaling, and skewing the coordinate system, these do not affect existing objects on the page but only subsequently drawn objects. Rotating text, images, and imported PDF pages by multiples of 90 degrees is easily accomplished with the *orientate* option in *PDF\_fit\_textline()*, *PDF\_fit\_image()*, and *PDF\_fit\_pdi\_page()* functions.

Arbitrary rotation angles can be achieved by applying the general coordinate transformation functions. The following example generates some horizontal text, and rotates the coordinate system in order to show rotated text. The save/restore nesting makes it easy to continue with horizontal text in the original coordinate system after the vertical text is done:

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

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

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

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

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

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

A different coordinate system can be established for each page, but the *topdown* parameter must not be set within a page description (but only between pages). The *topdown*

feature has been designed to make it quite natural for PDFlib users to work in a top-down coordinate system. For the sake of completeness we'll list the detailed consequences of establishing a top-down coordinate system below.

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

- ▶ All function parameters which are designated as »coordinates« in the function descriptions. Some examples: *x, y* in *PDF\_moveto()*; *x, y* in *PDF\_circle()*, *x, y* (but not *width* and *height*!) in *PDF\_rect()*; *llx, lly, urx, ury* in *PDF\_create\_annotation()*.

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

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

### 3.2.2 Page Sizes and Coordinate Limits

**Standard page formats.** For the convenience of PDFlib users, Table 3.4 lists common standard page sizes<sup>1</sup>. Symbolic page size names may be used for the *width* and *height* options in *PDF\_begin/end\_page\_ext()*. They are called *<format>.width* and *<format>.height*, where *<format>* is one of the formats in Table 3.4 (in lowercase, e.g. *a4.width*).

Table 3.4 Common standard page size dimensions in points

<i>format</i>	<i>width</i>	<i>height</i>	<i>format</i>	<i>width</i>	<i>height</i>	<i>format</i>	<i>width</i>	<i>height</i>
<i>a0</i>	2380	3368	<i>a4</i>	595	842	<i>letter</i>	612	792
<i>a1</i>	1684	2380	<i>a5</i>	421	595	<i>legal</i>	612	1008
<i>a2</i>	1190	1684	<i>a6</i>	297	421	<i>ledger</i>	1224	792
<i>a3</i>	842	1190	<i>a5</i>	501	709	<i>11x17</i>	792	1224

**Page size limits.** Although PDF and PDFlib don't impose any restrictions on the usable page size, Acrobat implementations suffer from architectural limits regarding the page size. Note that other PDF interpreters may well be able to deal with larger or smaller document formats. PDFlib will throw a non-fatal exception if Acrobat's page size limits are exceeded. The page size limits for Acrobat are shown in Table 3.5.

Table 3.5 Minimum and maximum page size of Acrobat

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

**Different page size boxes.** While many PDFlib developers only specify the width and height of a page, some advanced applications (especially for prepress work) may want to specify one or more of PDF's additional box entries. PDFlib supports all of PDF's box entries. The following entries, which may be useful in certain environments, can be specified by PDFlib clients (definitions taken from the PDF reference):

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

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

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

```
/* start a new page with custom CropBox */
PDF_begin_page_ext(p, 595, 842, "cropbox {10 10 500 800}");
```

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

**Output accuracy and coordinate range.** PDFlib's numerical output accuracy has been carefully chosen to match the requirements of PDF and the supported environments, while at the same time minimizing output file size. As detailed in Table 3.6 PDFlib's accuracy depends on the absolute value of coordinates. While most developers may safely ignore this issue, demanding applications should take care in their scaling operations in order to not exceed PDF's built-in coordinate limits.

Table 3.6 Output accuracy and coordinate range

<i>absolute value</i>	<i>output</i>
0 ... 0.000015	0
0.000015 ... 32767.999999	rounded to four decimal digits
32768 ... $2^{31} - 1$	rounded to next integer
$\geq 2^{31}$	an exception will be raised

### 3.2.3 Paths

A path is a shape made of an arbitrary number of straight lines, rectangles, or curves. A path may consist of several disconnected sections, called subpaths. There are several operations which can be applied to a path (see Section 8.4.6, »Path Painting and Clipping«, page 233):

- ▶ **Stroke** draws a line along the path, using client-supplied parameters (e.g., color, line width) for drawing.
- ▶ **Filling** paints the entire region enclosed by the path, using client-supplied parameters for filling.
- ▶ **Clipping** reduces the imageable area for subsequent drawing operations by replacing the current clipping area (which is the page size by default) with the intersection of the current clipping area and the area enclosed by the path.

- Merely terminating the path results in an invisible path, which will nevertheless be present in the PDF file. This will only rarely be required.

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

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

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

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

### 3.2.4 Templates

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

**Using templates with PDFlib.** Templates can only be *defined* outside of a page description, and can be *used* within a page description. However, templates may also contain other templates. Obviously, using a template within its own definition is not possible. Referring to an already defined template on a page is achieved with the *PDF\_fit\_image()* function just like images are placed on the page (see Section 5.3, »Placing Images and Imported PDF Pages«, page 136). The general template idiom in PDFlib looks as follows:

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

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

- ▶ The functions in Section 8.6, »Image and Template Functions«, page 243, except *PDF\_fit\_image()*, and *PDF\_close\_image()*. This is not a big restriction since images can be opened outside of a template definition, and freely be used within a template (but not opened).
- ▶ The functions in Section 8.9.7, »Deprecated Hypertext Parameters and Functions«, page 276. Hypertext elements must always be defined on the page where they should appear in the document, and cannot be generated as part of a template.

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

## 3.3 Working with Color

### 3.3.1 Color and Color Spaces

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

- ▶ Gray values between 0=black and 1=white;
- ▶ RGB triples, i.e., three values between 0 and 1 specifying the percentage of red, green, and blue; (0, 0, 0)=black, (1, 1, 1)=white;
- ▶ Four CMYK values between 0=no color and 1=full color, representing cyan, magenta, yellow, and black values; (0, 0, 0, 0)=white, (0, 0, 0, 1)=black. Note that this is different from the RGB specification.
- ▶ Device-independent colors in the CIE L\*a\*b\* color space are specified by a luminance value in the range 0-100 and two color values in the range -127 to 128 (see Section 3.3.4, »Color Management and ICC Profiles«, page 67).
- ▶ ICC-based colors are specified with the help of an ICC profile (see Section 3.3.4, »Color Management and ICC Profiles«, page 67).
- ▶ Spot color (separation color space): a predefined or arbitrarily named custom color with an alternate representation in one of the other color spaces above; this is generally used for preparing documents which are intended to be printed on an offset printing machine with one or more custom colors. The tint value (percentage) ranges from 0=no color to 1=maximum intensity of the spot color. See Section 3.3.3, »Spot Colors«, page 64, for a list of spot color names.
- ▶ Patterns: tiling with an object composed of arbitrary text, vector, or image graphics (see Section 3.3.2, »Patterns and Smooth Shadings«, page 63).
- ▶ Shadings (smooth blends) provide a gradual transition between two colors, and are based on another color space (see Section 3.3.2, »Patterns and Smooth Shadings«, page 63).
- ▶ The indexed color space is a not really a color space on its own, but rather an efficient coding of another color space. It will automatically be generated when an indexed (palette-based) image is imported.

The default color for stroke and fill operations is black.

### 3.3.2 Patterns and Smooth Shadings

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

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

- ▶ First, the pattern must be defined between *PDF\_begin\_pattern()* and *PDF\_end\_pattern()*. Most graphics operators can be used to define a pattern.
- ▶ The pattern handle returned by *PDF\_begin\_pattern()* can be used to set the pattern as the current color using *PDF\_setcolor()*.

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

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

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

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

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

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

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

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

### 3.3.3 Spot Colors

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

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

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

PDFlib supports various built-in spot color libraries as well as custom (user-defined) spot colors. When a spot color name is requested with *PDF\_makespotcolor()* PDFlib will first check whether the requested spot color can be found in one of its built-in libraries. If so, PDFlib will use built-in values for the alternate color. Otherwise the spot color is assumed to be a user-defined color, and the client must supply appropriate alternate col-

<sup>1</sup> See [www.artspdf.com](http://www.artspdf.com)



or values (via the current color). Spot colors can be tinted, i.e., they can be used with a percentage between 0 and 1.

By default, built-in spot colors can not be redefined with custom alternate values. However, this behavior can be changed with the *spotcolorlookup* parameter. This can be useful to achieve compatibility with older applications which may use different color definitions.

PDFlib will automatically generate suitable alternate colors for built-in spot colors when a PDF/X conformance level has been selected (see Section 7.4, »PDF/X«, page 171). For custom spot colors it is the user's responsibility to provide alternate colors which are compatible with the selected PDF/X conformance level.

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

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

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



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

PANTONE <id> <paperstock>

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

to *false*). The following code snippet demonstrates the use of a PANTONE Color with a tint value of 70 percent:

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

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

*Note* PANTONE® Colors are not supported in the PDF/X-1:2001, PDF/X-1a:2001, and PDF/X-1a:2003 modes.

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



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

Spot color names are case-sensitive; use uppercase as shown in the examples. The HKS prefix must always be provided in the swatch name as shown in the examples. Generally, HKS color names must be constructed according to one of the following schemes:

HKS <id> <paperstock>

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

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

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

There is no separate PDFlib function for setting the alternate color for a new spot color; instead, the current fill color will be used. Except for an additional call to set the al-

ternate color, defining and using custom spot colors works similarly to using built-in spot colors:

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

### 3.3.4 Color Management and ICC Profiles

PDFlib supports several color management concepts including device-independent color, rendering intents, and ICC profiles.

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

**Rendering Intents.** Although PDFlib clients can specify device-independent color values, a particular output device is not necessarily capable of accurately reproducing the required colors. In this situation some compromises have to be made regarding the trade-offs in a process called gamut compression, i.e., reducing the range of colors to a smaller range which can be reproduced by a particular device. The rendering intent can be used to control this process. Rendering intents can be specified for individual images by supplying the *renderingintent* parameter or option to *PDF\_load\_image()*. In addition, rendering intents can be specified for text and vector graphics by supplying the *renderingintent* option to *PDF\_create\_gstate()*. Table 3.7 lists the available rendering intents and their meanings.

Table 3.7 Rendering intents

rendering intent	explanation	typical use
Auto	Do not specify any rendering intent in the PDF file, but use the device's default intent instead. This is the default.	unknown or unspecific uses
AbsoluteColorimetric	No correction for the device's white point (such as paper white) is made. Colors which are out of gamut are mapped to nearest value within the device's gamut.	exact reproduction of solid colors; not recommended for other uses.
RelativeColorimetric	The color data is scaled into the device's gamut, mapping the white points onto one another while slightly shifting colors.	vector graphics
Saturation	Saturation of the colors will be preserved while the color values may be shifted.	business graphics
Perceptual	Color relationships are preserved by modifying both in-gamut and out-of-gamut colors in order to provide a pleasing appearance.	scanned images

**ICC profiles.** The International Color Consortium (ICC)<sup>1</sup> defined a file format for specifying color characteristics of input and output devices. These ICC color profiles are considered an industry standard, and are supported by all major color management system

1. See [www.color.org](http://www.color.org)

and application vendors. PDFlib supports color management with ICC profiles in the following areas:

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

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

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

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

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

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

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

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

**Acceptable ICC profiles.** The type of acceptable ICC profiles depends on the *usage* parameter supplied to *PDF\_load\_iccprofile()*:

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

**The sRGB color space and sRGB ICC profile.** PDFlib supports the industry-standard RGB color space called sRGB (formally IEC 61966-2-1)<sup>1</sup>. sRGB is supported by a variety of software and hardware vendors and is widely used for simplified color management for

<sup>1</sup>. See [www.srgb.com](http://www.srgb.com)

consumer RGB devices such as digital still cameras, office equipment such as color printers, and monitors. PDFlib supports the sRGB color space and includes the required ICC profile data internally. Therefore an sRGB profile must not be configured explicitly by the client, but it is always available without any additional configuration. It can be requested by calling `PDF_load_iccprofile()` with `profilename = sRGB`.

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

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

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

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

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

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

```
icchandle = PDF_load_iccprofile(...)
if (icchandle == -1) {
    return;
}
PDF_set_value(p, "setcolor:iccprofilecmyk", icchandle);
PDF_setcolor(p, "fill", "iccbasedcmyk", 0, 1, 0, 0);
```

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

ue. The following examples will set the sRGB color space as the default RGB color space for text, images, and vector graphics:

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

**Defining PDF/X output intents.** An output device (printer) profile can be used to specify an output condition for PDF/X. This is done by supplying *usage = outputintent* in the call to *PDF\_load\_iccprofile()*. For details see Section 7.4.2, »Generating PDF/X-conforming Output«, page 172.

# 4 Text Handling

## 4.1 Overview of Fonts and Encodings

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

### 4.1.1 Supported Font Formats

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

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

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

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

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

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

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

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

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

### 4.1.2 Encodings

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

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

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

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

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

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

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

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



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

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

- ▶ Purely Unicode-based addressing via the *unicode* encoding keyword. In this case the client directly supplies Unicode strings to PDFlib. The Unicode strings may be formatted according to one of several standard methods (such as UTF-16, UTF-8) and byte orderings (little-endian or big-endian).
- ▶ CMap-based addressing for a variety of Chinese, Japanese, and Korean standards. In combination with standard CJK fonts PDFlib supports all CMaps supported by Acrobat. This includes both Unicode-based CMaps and others (see Section 4.7, «Chinese, Japanese, and Korean Text», page 101).
- ▶ Glyph id addressing for TrueType and OpenType fonts via the *glyphid* encoding keyword. This is useful for advanced text processing applications which need access to individual glyphs in a font without reference to any particular encoding scheme, or must address glyphs which do not have any Unicode mapping. The number of valid glyph ids in a font can be queried with the *fontmaxcode* parameter.

### 4.1.3 Support for the Unicode Standard

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

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

## 4.2 Font Format Details

### 4.2.1 PostScript Fonts

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

- ▶ The platform-independent AFM (Adobe Font Metrics) and the Windows-specific PFM (Printer Font Metrics) format for metrics information. While AFM-based font metrics can be rearranged to any encoding supported by the font, PFM font metrics can only be used with the following encodings: *winansi*, *iso8859-1*, *unicode*, *ebcdic*, and *builtin* (the latter only for symbol fonts).
- ▶ The platform-independent PFA (Printer Font ASCII) and the Windows-specific PFB (Printer Font Binary) format for font outline information in the PostScript Type 1 format, (sometimes also called »ATM fonts«).
- ▶ On the Mac, resource-based PostScript Type 1 fonts, sometimes called LWFN (Laser-Writer Font) fonts, are also supported.
- ▶ OpenType fonts with PostScript outlines (\*.otf).

If you can get hold of a PostScript font file, but not the corresponding metrics file, you can try to generate the missing metrics using one of several freely available utilities. However, be warned that such conversions often result in font or encoding problems. For this reason it is recommended to use the font outline and metrics data as supplied by the font vendor.

**PostScript font names.** When working with host fonts it is important to use the exact (case-sensitive) PostScript font name. If you are working with disk-based font files you can use arbitrary alias names (see Section 4.3.1, »How PDFlib Searches for Fonts«, page 78). There are several possibilities to find a PostScript font's exact name:

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

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

**PostScript glyph names.** In order to write a custom encoding file or find fonts which can be used with one of the supplied encodings you will have to find information about the exact definition of the character set to be defined by the encoding, as well as the exact glyph names used in the font files. You must also ensure that a chosen font provides all necessary characters for the encoding. For example, the core fonts supplied with Acrobat 4/5 do not support ISO 8859-2 (Latin 2) nor Windows code page 1250. If you happen to have the FontLab<sup>1</sup> font editor (by the way, a great tool for dealing with all kinds of

<sup>1</sup> See [www.fontlab.com](http://www.fontlab.com)

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

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

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

## 4.2.2 TrueType and OpenType Fonts

**TrueType and OpenType file formats.** PDFlib supports the following file formats for TrueType and OpenType fonts:

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

**TrueType and OpenType font names.** When working with host fonts it is important to use the exact (case-sensitive) TrueType font name (on Windows you can also use the base name of the font plus a style name suffix, see below). If you are working with disk-based font files you can use arbitrary alias names (see Section 4.3.1, »How PDFlib Searches for Fonts«, page 78). In the generated PDF the name of a TrueType font may differ from the name used in PDFlib (or Windows). This is normal, and results from the fact that PDF uses the PostScript name of a TrueType font, which differs from its genuine TrueType name (e.g., *TimesNewRomanPSMT* vs. *Times New Roman*).

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

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

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

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

**Windows font style names.** When querying host fonts from the Windows operating system PDFlib users have access to a feature provided by the Windows font selection machinery: style names can be provided for the weight and slant of a TrueType or OpenType font, for example

Georgia,Bold

This will instruct Windows to search for a particular bold, italic, or other variation of the base font. Depending on the available fonts Windows will select a font which most closely resembles the requested style (it will not create a new font variation). The font found by Windows may be different from the requested font, and the font name in the generated PDF may be different from the requested name; PDFlib does not have any control over Windows' font selection. Also, font style names only work with TrueType and OpenType host fonts, but not for PostScript fonts or fonts configured via a disk-based font file.

The following keywords (separated from the font name with a comma) can be attached to the base font name supplied to `PDF_load_font()` to specify the font weight:

none, thin, extralight, ultralight, light, normal, regular, medium, semibold, demibold, bold, extrabold, ultrabold, heavy, black

The following keyword can be specified alternatively or in addition to the above:

italic

The keywords are case-insensitive. If two style names are used both must be separated with a comma, for example:

Georgia,Bold,Italic

*Note Windows style names for fonts may be useful if you have to deal with localized font names since they provide a universal method to access font variations regardless of their localized names.*

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

```
ftxinstalledfonts -f
```

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

2. See [developer.apple.com/fonts/OSXTools.html](http://developer.apple.com/fonts/OSXTools.html)

### 4.2.3 User-Defined (Type 3) Fonts

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

*Note PostScript Type 3 fonts are not supported.*

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

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

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

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

PDF_end_font(p);
```

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

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

## 4.3 Font Embedding and Subsetting

### 4.3.1 How PDFlib Searches for Fonts

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

- ▶ Disk-based font files which have been statically configured via a UPR configuration file (see Section 3.1.6, »Resource Configuration and File Searching«, page 51) or dynamically via `PDF_set_parameter()` and the *FontOutline* resource category.
- ▶ Fonts which have been installed in the operating system. We refer to such fonts as *host fonts*. Instead of fiddling with font and configuration files simply install the font in the operating system (read: drop it into the appropriate *fonts* directory), and PDFlib will happily use it. Host fonts are available on Mac (only TrueType and OpenType, but not PostScript fonts) and Windows systems. They can explicitly be configured with the *HostFont* UPR resource category in order to control the search order. This feature can be used, for example, to prefer host fonts over the built-in core fonts.
- ▶ Font data passed by the client directly in memory by means of a PDFlib virtual file (PVF). This is useful for advanced applications which have the font data already loaded into memory and want to avoid unnecessary disk access by PDFlib (see Section 3.1.5, »The PDFlib Virtual File System (PVF)«, page 50 for details on virtual files).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 4.3.2 Font Embedding

**The PDF core fonts.** PDF viewers support a core set of 14 fonts which are assumed to be always available. Full metrics information for the core fonts is already built into the PDFlib binary so that no additional font files are required (unless the font is to be embedded). The core fonts are the following:

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

In order to replace one of the core fonts with a font installed on the system (host font) you must configure the font in the *HostFont* resource category. For example, the following line makes sure that instead of using the built-in core font data, the Symbol font will be taken from the host system:

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

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

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

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

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

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

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

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

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

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

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



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

### 4.3.3 Font Subsetting

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

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

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

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

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

The default value of *subsetlimit* is 100 percent. In other words, the subsetting option requested at *PDF\_load\_font()* will be honored unless the client explicitly requests a lower limit than 100 percent.

- ▶ The *subsetminsize* parameter can be used to completely disable subsetting for small fonts. If the original font file is smaller than the value of *subsetminsize* in KB, font subsetting will be disabled for this font. The default value is 100 KB.

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

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

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

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

## 4.4 Encoding Details

### 4.4.1 8-Bit Encodings

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

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

**Notes on the macroman encoding.** This encoding reflects the Mac OS character set, albeit with the old currency symbol at position 219 = 0xDB, and not the Euro glyph as re-defined by Apple (this incompatibility is dictated by the PDF specification). The *macroman\_euro* encoding is identical to *macroman* except that position 219 = 0xDB holds the Euro glyph instead of the currency symbol. Also, the *macroman* and *macroman\_euro* encodings don't include the Apple glyph and the mathematical symbols as defined in the Mac OS character set. These are available in the *macroman\_apple* encoding, but the required glyphs are contained only in few fonts.

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

- ▶ on Mac OS Classic it will be mapped to *macroman*;
- ▶ on IBM eServer iSeries and zSeries with MVS or USS it will be mapped to *ebcdic*;
- ▶ on Windows it will be mapped to *winansi*;
- ▶ on all other systems (including Mac OS X) it will be mapped to *iso8859-1*;

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

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

- ▶ On Windows: the current system code page (see below for details)
- ▶ On Unix and Mac OS X: *iso8859-1*
- ▶ On Mac OS Classic: *macroman*
- ▶ On IBM eServer iSeries: the current job's encoding (*IBMCCSID000000000000*)
- ▶ On IBM eServer zSeries: *ebcdic* (=code page 1047).

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

**Tapping system code pages.** PDFlib can be instructed to fetch code page definitions from the system and transform it appropriately for internal use. This is very convenient since it frees you from implementing the code page definition yourself. Instead of supplying the name of a built-in or user-defined encoding for *PDF\_load\_font()*, simply use

Table 4.2 Availability of glyphs for predefined encodings in several classes of fonts: some languages cannot be represented with Acrobat's core fonts.

code page	supported languages	PS Level 1/2, Acrobat 4/5 <sup>1</sup>	Acrobat 6 <sup>2</sup> core fonts	PostScript 3 fonts <sup>3</sup>	OpenType Pro Fonts <sup>4</sup>	TrueType »Big Fonts« <sup>5</sup>
winansi	identical to cp1252 (superset of iso8859-1)	yes	yes	yes	yes	yes
macroman	Mac Roman encoding, the original Macintosh character set	yes	yes	yes	yes	yes
macroman_euro	similar to macroman, but includes the Euro glyph instead of currency	yes	yes	yes	yes	yes
macroman_apple	similar to macroman_euro, but includes additional mathematical Symbols	–	–	–	yes	yes
ebcdic	EBCDIC code page 1047	yes	yes	yes	yes	yes
pdfdoc	PDFDocEncoding	yes	yes	yes	yes	yes
iso8859-1	(Latin-1) Western European languages	yes	yes	yes	yes	yes
iso8859-2	(Latin-2) Slavic languages of Central Europe	–	–	yes	yes	yes
iso8859-3	(Latin-3) Esperanto, Maltese	–	–	–	yes	yes
iso8859-4	(Latin-4) Estonian, the Baltic languages, Greenlandic	–	–	–	yes	yes
iso8859-5	Bulgarian, Russian, Serbian	–	–	–	yes	yes
iso8859-6	Arabic	–	–	–	–	yes
iso8859-7	Modern Greek	–	–	–	1 miss.	yes
iso8859-8	Hebrew and Yiddish	–	–	–	–	yes
iso8859-9	(Latin-5) Western European, Turkish	5 miss.	5 miss.	yes	yes	yes
iso8859-10	(Latin-6) Nordic languages	–	–	–	1 miss.	yes
iso8859-13	(Latin-7) Baltic languages	–	–	yes	yes	yes
iso8859-14	(Latin-8) Celtic	–	–	–	–	–
iso8859-15	(Latin-9) Adds Euro as well as French and Finnish characters to Latin-1	Euro miss.	yes	yes	yes	yes
iso8859-16	(Latin-10) Hungarian, Polish, Romanian, Slovenian	–	–	yes	yes	yes
cp1250	Central European	–	–	yes	yes	yes
cp1251	Cyrillic	–	–	–	yes	yes
cp1252	Western European (same as winansi)	yes	yes	yes	yes	yes
cp1253	Greek	–	–	–	1 miss.	yes
cp1254	Turkish	5 miss.	–	yes	yes	yes
cp1255	Hebrew	–	–	–	–	yes
cp1256	Arabic	–	–	–	–	5 miss.
cp1257	Baltic	–	–	yes	yes	yes
cp1258	Viet Nam	–	–	–	–	yes

1. Core fonts shipped with Acrobat 4/5 (original Adobe Latin character set; generally Type 1 Fonts since 1982)

2. Acrobat 6 relies on the fonts which are available with the system in order to display Times and Helvetica. Therefore the results vary widely depending on the number and kind of installed fonts. For example, the system fonts shipped with Windows XP contain more glyphs than those available in older versions of Windows.

3. Extended Adobe Latin character set (CE-Fonts), generally Type 1 Fonts shipped with PostScript 3 devices

4. Adobe OpenType Pro fonts contain more glyphs than regular OpenType fonts.

5. Windows TrueType fonts containing large glyph complements, e.g.Tahoma

an encoding name which is known to the system. This feature is only available on selected platforms, and the syntax for the encoding string is platform-specific:

- ▶ On Windows the encoding name is *cp<number>*, where *<number>* is the number of any single-byte code page installed on the system (see Section 4.7.3, »Custom CJK Fonts«, page 105, for information on multi-byte Windows code pages):

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

Single-byte code pages will be transformed into an internal 8-bit encoding, while multi-byte code pages will be mapped to *unicode*. The text must be supplied in a format which is compatible with the chosen code page (e.g. SJIS for *cp932*).

- ▶ On IBM eServer iSeries any *Coded Character Set Identifier* (CCSID) can be used. The CCSID must be supplied as a string, and PDFlib will apply the prefix *IBMCCSID* to the supplied code page number. PDFlib will also add leading 0 characters if the code page number uses fewer than 5 characters. Supplying 0 (zero) as the code page number will result in the current job's encoding to be used:

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

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

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

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

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

- ▶ Generate a description of the encoding in a simple text format.
- ▶ Configure the encoding in the PDFlib resource file (see Section 3.1.6, »Resource Configuration and File Searching«, page 51) or via *PDF\_set\_parameter()*.
- ▶ Provide a font (metrics and possibly outline file) that supports all characters used in the encoding.

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

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

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

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

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

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

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

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

#### 4.4.2 Symbol Fonts and Font-specific Encodings

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

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

```
EncodingScheme FontSpecific
```

Text fonts can be reencoded (adjusted to a certain code page or character set), while symbolic fonts can't, and must use *builtin* encoding instead. However, the widely used *Symbol* and *ZapfDingbats* fonts can also be used with *unicode* encoding.

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

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

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

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

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

### 4.4.3 Glyph ID Addressing for TrueType and OpenType Fonts

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

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

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

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

### 4.4.4 The Euro Glyph

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

- ▶ With *unicode* encoding use the character U+20AC.
- ▶ In *winansi* encoding the location is 0x80 (hexadecimal) or 128 (decimal).
- ▶ The common *iso8859-1* encoding does not contain the Euro character. However, the *iso8859-15* encoding is an extension of *iso8859-1* which adds the Euro character at 0xA4 (hexadecimal) or 164 (decimal).



- ▶ The original *macroman* encoding, which is still the same in PDF, does not contain the Euro character. However, Apple modified this encoding and replaced the old currency glyph with the Euro glyph at 0xDB (hexadecimal) or 219 (decimal). In order to use this modified Mac encoding use *macroman\_euro* instead of *macroman*.

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

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



## 4.5 Unicode Support

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



### 4.5.1 Unicode for Page Content and Hypertext

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

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

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

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

This feature can be activating by using any of PDFlib's internal code pages, or supplying a suitable custom encoding or code page file (see Section 4.4.1, »8-Bit Encodings«, page 83).

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

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

`U+0400`

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

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

<sup>1</sup>. See [www.unicode.org](http://www.unicode.org)

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

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

**Unicode for hypertext strings.** Unicode can be supplied for various hypertext elements, such as bookmarks, contents and title of note annotations (see Figure 4.1), standard and user-defined document information field contents, description and author of file attachments.

While PDF supports only Unicode in big-ending UTF-16 format and PDFDocEncoding, which is a superset of ISO 8859-1 for hypertext elements, PDFlib supports all 8-bit and Unicode-based encodings as well as system-installed code pages which are allowed for *PDF\_load\_font()*, and will automatically apply any required conversions.

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

### 4.5.2 Content Strings, Hypertext Strings, and Name Strings

There are different string types in the PDFlib API depending on their usage:

- Content strings: these will be used to create genuine page content (page descriptions) according to the encoding chosen by the user for a particular font. All *text* parameters of the page content functions in Section 8.3.4, »Simple Text Output«, page 209, and Section 8.3.5, »Multi-Line Text Output with Textflows«, page 216, fall in this class.

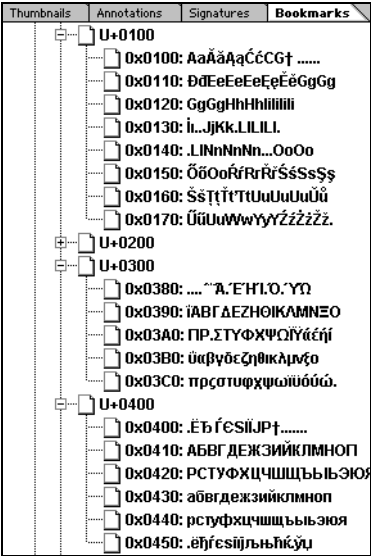
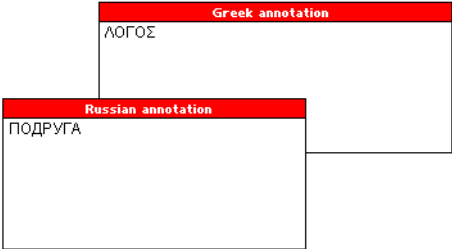


Fig. 4.1  
Unicode bookmarks (left) and Unicode text annotations (right)



- ▶ Hypertext strings: these are mostly used for hypertext functions such as bookmarks and annotations, and are explicitly labeled *Hypertext string* in the function descriptions. Many parameters and options of the functions in Section 8.9, »Hypertext Functions«, page 261, fall in this class, as well as some others.
- ▶ Name strings: these are used for external file names, font names, block names, etc., and are marked as *name string* in the function descriptions. They slightly differ from Hypertext strings, but only in languages which are not Unicode-aware.

### 4.5.3 String Handling in Unicode-capable Languages

The following PDFlib language bindings are Unicode-capable:

- ▶ COM
- ▶ .NET
- ▶ Java
- ▶ REALbasic
- ▶ Tcl

String handling in these environments is straightforward: all strings will automatically be provided to the PDFlib kernel as Unicode strings in UTF-16 format. The language wrappers will correctly deal with Unicode strings provided by the client, and automatically set certain PDFlib parameters. This has the following consequences:

- ▶ Since the language wrapper automatically sets the *textformat*, *hypertextformat*, and *hypertextencoding* parameters, these are not accessible by the client, and must not be used. The PDFlib language wrapper applies all required conversions so that client-supplied hypertext strings will always arrive in PDFlib in *utf16* format and *unicode* encoding.
- ▶ Since the language environment always passes strings in UTF-16 to PDFlib, UTF-8 can not be used with Unicode-capable languages. It must be converted to UTF-16 before, using the native methods provided by the environment.
- ▶ Using *unicode* encoding for page descriptions is the easiest way to deal with encodings in Unicode-aware languages.
- ▶ Non-Unicode CMaps for standard CJK fonts on page descriptions must be avoided since the wrapper will always supply Unicode to the PDFlib core; only Unicode CMaps can be used.

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

### 4.5.4 String Handling in non-Unicode-capable Languages

*Note This section does not apply to the Unicode-capable languages Java and Tcl.*

The following PDFlib language bindings are not Unicode-capable:

- ▶ C
- ▶ C++
- ▶ Cobol
- ▶ Perl
- ▶ PHP
- ▶ Python
- ▶ RPG

Although Unicode text can be used in these languages, handling of the various string types is a bit more complicated:

- ▶ **Content strings:** These are strings used to create genuine page content. Interpretation of these strings is controlled by the *textformat* parameter (detailed below) and the *encoding* parameter of *PDF\_load\_font()*. If *textformat=auto* (which is the default) *utf16* format will be used for the *unicode* and *glyphid* encodings as well as UCS-2 CMaps. For all other encodings the format will be *bytes*. The length of UTF-16 strings must be supplied in a separate length parameter.
- ▶ **Hypertext strings:** string interpretation is controlled by the *hypertextformat* and *hypertextencoding* parameters (detailed below). If *hypertextformat=auto* (which is the default) *utf16* format will be used if *hypertextencoding=unicode*, and *bytes* otherwise. In languages which do not support native string objects (Cobol, C, and RPG) the length of UTF-16 strings must be supplied in a separate length parameter.
- ▶ **Name strings:** these are interpreted slightly differently from page description strings, depending on the *length* parameter and the existence of a BOM at the beginning of the string. In C, if the *length* parameter is different from 0 the string will be interpreted as UTF-16. Otherwise (i.e., if the *length* parameter is 0, the function doesn't provide one, or a language other than C is used) it will be interpreted as UTF-8 if it starts with a UTF-8 BOM, or as EBCDIC UTF-8 if it starts with an EBCDIC UTF-8 BOM, or as *host* if no BOM is found (or *ebcdic* on EBCDIC-based platforms).

**Strings in option lists.** Strings within option lists require special attention since they cannot be expressed as Unicode strings in UTF-16 format, but only as byte strings. For this reason UTF-8 is used for Unicode options. By looking for a BOM at the beginning of an option PDFlib decides how to interpret it. The BOM will be used to determine the format of the string, and the string type (content string, hypertext string, or name string as defined above) will be used to determine the appropriate encoding. More precisely, interpreting a string option works as follows:

- ▶ If the option starts with a UTF-8 BOM (`\xEF\xBB\xBF`) it will be interpreted as UTF-8.
- ▶ If the option starts with an EBCDIC UTF-8 BOM (`\x57\x8B\xAB`) it will be interpreted as EBCDIC UTF-8.
- ▶ If no BOM is found, the string will be treated as *winansi* (or *ebcdic* on EBCDIC-based platforms).

*Note* The *PDF\_utf16\_to\_utf8()* utility function can be used to create UTF-8 strings from UTF-16 strings, which is useful for creating option lists with Unicode values.

**Text Format for Unicode Strings.** The Unicode standard supports several transformation formats for storing the actual byte values which comprise a Unicode string. These vary in the number of bytes per character and the ordering of bytes within a character. Unicode strings in PDFlib can be supplied in UTF-8 or UTF-16 formats with any byte ordering. This can be controlled with the *textformat* parameter for all text on page descriptions, and the *hypertextformat* parameter for all hypertext elements. Table 4.3 lists the values which are supported for both of these parameters.

Table 4.3 Text formats

<i>textformat</i>	<i>explanation</i>
<i>bytes</i>	One byte in the string corresponds to one character. This is mainly useful for 8-bit encodings.
<i>utf8</i>	Strings are expected in UTF-8 format.

Table 4.3 Text formats

textformat	explanation
ebcdicutf8	Strings are expected in EBCDIC-coded UTF-8 format (only on iSeries and zSeries).
utf16	Strings are expected in UTF-16 format. A Unicode Byte Order Mark (BOM) at the start of the string will be evaluated and then removed. If no BOM is present the string is expected in the machine's native byte ordering (on Intel x86 architectures the native byte order is little-endian, while on Sparc and PowerPC systems it is big-endian).
utf16be	Strings are expected in UTF-16 format in big-endian byte ordering. There is no special treatment for Byte Order Marks.
utf16le	Strings are expected in UTF-16 format in little-endian byte ordering. There is no special treatment for Byte Order Marks.
auto	Equivalent to bytes for 8-bit encodings, and utf16 for wide-character addressing (unicode, glyphid, or a UCS2 or UTF16 CMap). This setting will provide proper text interpretation in most environments which do not use Unicode natively.

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

Although the *textformat* setting is in effect for all encodings, it will be most useful for *unicode* encoding. Table 4.4 details the interpretation of text strings for various combinations of font encodings and *textformat* settings.

Table 4.4 Relationship of font encodings and text format

font encoding	textformat = bytes	textformat = utf8, utf16, utf16be, or utf16le
8-bit, or builtin encoding for TTF/OTF	8-bit codes	convert Unicode values to 8-bit codes according to the chosen encoding <sup>1</sup>
builtin encoding for PostScript	8-bit codes	only in Unicode-capable language bindings. PDFlib will throw an exception otherwise
U+XXXX	8-bit codes will be added to the offset XXXX to address Unicode values	convert Unicode values to 8-bit codes according to the chosen Unicode offset
glyphid	8-bit codes address glyph ids from 0 to 255	Unicode values will be interpreted as glyph ids <sup>2</sup>
unicode and UCS2- or UTF16 CMaps	8-bit codes address Unicode values from U+0000 to U+00FF	any Unicode value, encoded according to the chosen text format <sup>1</sup>
any other CMap (not Unicode-based)	any single- or multibyte codes according to the chosen CMap	only in Unicode-capable language bindings. PDFlib will throw an exception otherwise

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

2. If the glyph id is not available in the font PDFlib will issue a warning and replace it with glyph id 0.

**Hypertext encoding.** The *hypertextencoding* parameter works analogous to the *encoding* parameter of *PDF\_load\_font()*, and controls the 8-bit encoding of hypertext strings. It can attain most encoding names known to PDFlib, including *auto* (see Section 4.4, »Encoding Details«, page 83). Note that *glyphid*, *builtin*, and *CMap* names are not allowed for this parameter. The default setting for the *hypertextencoding* parameter is *auto*.

**Hypertext format.** Similarly to the *textformat* parameter, the format of hypertext strings can be controlled with the *hypertextformat* parameter. However, interpretation of the allowed values is somewhat different for the *hypertextformat* parameter. While

*utf8*, *utf16*, *utf16be*, and *utf16le* have the same meaning as for the *textformat* parameter, the behavior of *bytes* and *auto* is slightly different:

- ▶ *auto*: UTF-16 strings with big-endian BOM will be detected (in C such strings must be terminated with a double-null), and Unicode output will be generated. If the string does not start with a big-endian BOM it will be interpreted as an 8-bit encoded string according to the *hypertextencoding* parameter (see above). If it contains at least one character which is not contained in PDFDocEncoding, the complete string will be converted to a big-endian UTF-16 string, and written to the PDF output as Unicode. Otherwise it will be written to the PDF output as 8-bit encoded PDFDocEncoding text.
- ▶ *bytes*: one byte in the string corresponds to one character, and the string will be output without any interpretation. This is mainly useful for 8-bit encodings. In addition, UTF-16 strings with big-endian BOM will automatically be detected. In C, such strings must be terminated with a double-null unless the length in bytes is explicitly supplied in the respective function call.

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

## 4.5.5 Character References

Some environments require the programmer to write source code in 8-bit encodings (such as *winansi*, *macroman*, or *ebcdic*). This makes it cumbersome to include isolated Unicode characters in 8-bit encoded text without changing all characters in the text to multi-byte encoding. In order to aid developers in this situation, PDFlib supports character references, a method known from markup languages such as SGML and HTML. More precisely, PDFlib supports all numeric character references and character entity references defined in HTML 4.0<sup>1</sup>. Numeric character references can be supplied in decimal or hexadecimal notation for the character's Unicode value.

*Note Code points 128-159 (decimal) or 0x80-0x9F (hexadecimal) do not reference winansi code points, but are unassigned.*

The following are examples for valid character references along with a description of the resulting character:

&#173;	soft hyphen
&#xAD;	soft hyphen
&shy;	soft hyphen
&#229;	letter a with small circle above (decimal)
&#xE5;	letter a with small circle above (hexadecimal, lowercase x)
&#Xe5;	letter a with small circle above (hexadecimal, uppercase X)
&#x20AC;	Euro glyph (hexadecimal)
&#8364;	Euro glyph (decimal)
&euro;	Euro glyph (entity name)
&lt;	less than sign
&gt;	greater than sign
&amp;	ampersand sign
&Alpha;	Greek Alpha

Character references can be used in all content strings, hypertext strings, and name strings, e.g. in text which will be placed on the page using the *show* or *textflow* functions, as well as text supplied to the hypertext functions. Character references will not be substituted in option lists, however.

1. See [www.w3.org/TR/REC-html40/charset.html#h-5.3](http://www.w3.org/TR/REC-html40/charset.html#h-5.3)

Character references will not be converted by default; you must explicitly set the *charref* parameter to *true* if you want to use character references globally in your text:

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

Character references can also be enabled for textflow processing by supplying the *charref* option to *PDF\_create\_textflow()* (either directly or as an inline option), *PDF\_fit\_textline()*, or *PDF\_fill\_textblock()*.

When character references are enabled you can supply numeric or entity references in 8-bit-encoded text:

```
PDF_set_parameter(p, "charref", "true");
PDF_set_parameter(p, "textformat", "bytes");
font = PDF_load_font(p, "Helvetica", 0, "unicode", "");
PDF_setfont(p, font, 24);
PDF_show_xy(p, "Price: 500&euro;", 50, 500);
```

*Note Although you can reference any Unicode character with character references (e.g. Greek characters and mathematical symbols), the font will not automatically be switched. In order to actually use such characters you must explicitly select an appropriate font if the current font does not contain the specified characters.*

## 4.5.6 Unicode-compatible Fonts

Precise Unicode semantics are important for PDFlib's internal processing, and crucial for properly extracting text from a PDF document, or otherwise reusing the document, e.g., converting the contents to another format. This is especially important when creating Tagged PDF which has strict requirements regarding Unicode compliance (see Section 7.5.1, »Generating Tagged PDF with PDFlib«, page 176). In addition to Tagged PDF Unicode compatibility is relevant for the textflow feature.

**Unicode-compatible fonts.** A font loaded with *PDF\_load\_font()* – more precisely: a combination of font and encoding – is considered Unicode-compatible if the encoding used for loading the font complies to all of the following conditions:

- ▶ The encoding *builtin* is only allowed for the *Symbol* and *ZapfDingbats* fonts and PostScript-based OpenType fonts.
- ▶ The encoding is not *glyphid*.
- ▶ If the encoding is one of the predefined CMaps in Table 4.6 it must be one of the UCS2 or UTF16 CMaps.

**Unicode-compatible output.** If you want to make sure that text can reliably be extracted from the generated PDF, and for generating Tagged PDF the output must be Unicode-compatible. PDF output created with PDFlib will be Unicode-compatible if all of the following conditions are true:

- ▶ All fonts used in the document must be Unicode-compatible as defined above, or use one of the predefined CMaps in Table 4.6.
- ▶ If the encoding has been constructed with *PDF\_encoding\_set\_char()* and glyph names without corresponding Unicode values, or loaded from an encoding file, all glyph names must be contained in the Adobe Glyph List or the list of well-known glyph names in the Symbol font.
- ▶ The *unicodemap* parameter or option is *true*.

- ▶ All text strings must have clearly defined semantics according to the Unicode standard, i.e. characters from the Private Use Area (PUA) are not allowed.
- ▶ PDF pages imported with PDI must be Unicode-compatible. PDI does not change the Unicode compatibility status of imported pages: it will neither remove nor add Unicode information.

When creating Tagged PDF output, text portions which violate these rules can still be made Unicode-compatible by supplying proper Unicode text with the *ActualText* option in *PDF\_begin\_item()*.



# 4.6 Text Metrics and Text Variations

## 4.6.1 Font and Character Metrics

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

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

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

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

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

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

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

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

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

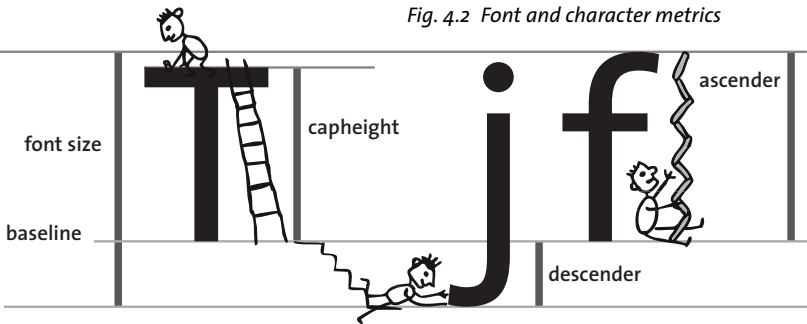


Fig. 4.2 Font and character metrics

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

$12 \text{ points} * 600/1000 = 7.2 \text{ points}$

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

#### 4.6.2 Kerning

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

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

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

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

Tele Vaso

No kerning

Tele Vaso

Kerning applied

Te Va

Character movement caused by kerning

Fig. 4.3 Kerning

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

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

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

### 4.6.3 Text Variations

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

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

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

- ▶ The base font is a TrueType or OpenType font, including standard and custom CJK fonts. The base font must not be one of the PDF core fonts (see Section 4.3.2, »Font Embedding«, page 79).
- ▶ The encoding is *winansi*, *macroman*, or one of the predefined CJK CMaps listed in Table 4.6 (since otherwise PDFlib will force font embedding).
- ▶ The *embedding* option must be set to *false*.
- ▶ The base font must be installed on the target system where the PDF will be viewed.

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

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

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

The *fontstyle* feature should not be confused with the similar concept of Windows font style names. While fontstyle only works under the conditions above and relies on Acrobat for simulating the artificial font style, the Windows style names are entirely based on the Windows font selection engine and cannot be used to simulate non-existent styles.

**Simulated italic fonts.** As an alternative to the *fontstyle* feature the *italicangle* parameter or option can be used to simulate italic fonts when only a regular font is available. This method creates a fake italic font by skewing the regular font by a user-provided angle, and does not suffer from the fontstyle restrictions mentioned above. Negative values will slant the text counterclockwise. Be warned that using a real italic or oblique font will result in much more pleasing output. However, if an italic font is not available

the *italicangle* parameter can be used to easily simulate one. This feature may be especially useful for CJK fonts. Typical values for the *italicangle* parameter are in the range -12 to -15 degrees:

```
PDF_set_value(p, "italicangle", -12);          /* create fake italic */
```

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

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

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

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

**Text rendering modes.** PDFlib supports several rendering modes which affect the appearance of text. This includes outline text and the ability to use text as a clipping path. Text can also be rendered invisibly which may be useful for placing text on scanned images in order to make the text accessible to searching and indexing, while at the same time assuring it will not be visible directly. The rendering modes are described in Table 8.16. They can be set with *PDF\_set\_value()* and the *textrendering* parameter.

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

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

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

## 4.7 Chinese, Japanese, and Korean Text

### 4.7.1 CJK support in Acrobat and PDF

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

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

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

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

### 4.7.2 Standard CJK Fonts and CMaps

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

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

2. See [www.adobe.com/products/acrobat/acrrasianfontpack.html](http://www.adobe.com/products/acrobat/acrrasianfontpack.html)

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

*Note* Acrobat's standard CJK fonts do not support bold and italic variations. However, these can be simulated with the artificial font style feature (see Section 4.6.3, »Text Variations«, page 99).

As can be seen from the table, the default CMaps support most CJK encodings used on Mac, Windows, and Unix systems, as well as several other vendor-specific encodings. In particular, the major Japanese encoding schemes Shift-JIS, EUC, ISO 2022, and Unicode (UCS-2 and UTF-16) are supported. Tables with all supported characters are available from Adobe<sup>1</sup>; CMap descriptions can be found in Table 4.6.

*Note* Unicode-capable language bindings must only use Unicode-compatible CMaps (UCS2 or UTF16). Other CMaps are not supported.

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

locale	font name	sample	supported CMaps (encodings)
Simplified Chinese	STSong-Light <sup>1</sup>	国际 国际	GB-EUC-H, GB-EUC-V, GBpc-EUC-H, GBpc-EUC-V, GBK-EUC-H, GBK-EUC-V, GBKp-EUC-H <sup>4</sup> , GBKp-EUC-V <sup>2</sup> , GBK2K-H <sup>2</sup> , GBK2K-V <sup>2</sup> , UniGB-UCS2-H, UniGB-UCS2-V, UniGB-UTF16-H <sup>5</sup> , UniGB-UTF16-V <sup>5</sup>
	STSongStd-Light-Acro <sup>2</sup>		
	AdobeSongStd-Light-Acro <sup>3</sup>		
Traditional Chinese	MHei-Medium <sup>1</sup>	中文 中文 中文	B5pc-H, B5pc-V, HKscs-B5-H <sup>4</sup> , HKscs-B5-V <sup>4</sup> , ETen-B5-H, ETen-B5-V, ETenms-B5-H, ETenms-B5-V, CNS-EUC-H, CNS-EUC-V, UniCNS-UCS2-H, UniCNS-UCS2-V, UniCNS-UTF16-H <sup>5</sup> , UniCNS-UTF16-V <sup>5</sup>
	MSung-Light <sup>1</sup>		
	MSungStd-Light-Acro <sup>2</sup> AdobeMingStd-Light-Acro <sup>3</sup>		
Japanese	HeiseiKakuGo-W5 <sup>1</sup>	日本語 日本語 日本語	83pv-RKSJ-H, goms-RKSJ-H, goms-RKSJ-V, gomsp-RKSJ-H, gomsp-RKSJ-V, 90pv-RKSJ-H, Add-RKSJ-H, Add-RKSJ-V, EUC-H, EUC-V, Ext-RKSJ-H, Ext-RKSJ-V, H, V, UniJIS-UCS2-H, UniJIS-UCS2-V, UniJIS-UCS2-HW-H, UniJIS-UCS2-HW-V, UniJIS-UTF16-H <sup>5</sup> , UniJIS-UTF16-V <sup>5</sup>
	HeiseiMin-W3 <sup>1</sup>		
	KozMinPro-Regular-Acro <sup>2</sup>		
	KozGoPro-Medium-Acro <sup>3</sup>		
Korean	HYGoThic-Medium <sup>1</sup>	한국 한국 한국	KSC-EUC-H, KSC-EUC-V, KSCms-UHC-H, KSCms-UHC-V, KSCms-UHC-HW-H, KSCms-UHC-HW-V, KSCpc-EUC-H, UniKS-UCS2-H, UniKS-UCS2-V, UniKS-UTF16-H <sup>5</sup> , UniKS-UTF16-V <sup>5</sup>
	HYSMyeongJo-Medium <sup>1</sup>		
	HYSMyeongJoStd-Medium-Acro <sup>2</sup>		
	AdobeMyungjoStd-Medium-Acro <sup>3</sup>		

1. Available in Acrobat 4; Acrobat 5 and 6 will substitute these with different fonts.  
2. Available in Acrobat 5 only  
3. Available in Acrobat 6 only  
4. Only available when generating PDF 1.4 or above  
5. Only available when generating PDF 1.5

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

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

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

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

For multi-byte encodings, the high-order byte of a character must appear first. Alternatively, the byte ordering and text format can be selected with the *textformat* parameter (see Section 4.5.1, »Unicode for Page Content and Hypertext«, page 89) provided a Unicode CMap (UCS-2 or UTF-16) is used.

Since several of the supported encodings may contain null characters in the text strings, C developers must take care not to use the *PDF\_show()* etc. functions, but instead *PDF\_show2()* etc. which allow for arbitrary binary strings along with a length parameter. For all other language bindings, the text functions support binary strings, and *PDF\_show2()* etc. are not required.

**Restrictions for standard CJK fonts and CMaps.** The following features are not supported for standard CJK fonts in combination with non-Unicode CMaps (Unicode CMaps are those with UCS2 or UTF16 in their name):

- ▶ calculating the extent of text with *PDF\_stringwidth()* (but see Section 4.7.4, »Forcing monospaced Fonts«, page 107)
- ▶ using *PDF\_create\_textflow()* and related Textflow functions
- ▶ activating underline/overline/strikeout mode
- ▶ retrieving the *textx/texty* position

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

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

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

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

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

locale	CMap name	character set and text format
Simplified Chinese	UniGB-UCS2-H UniGB-UCS2-V	Unicode (UCS-2) encoding for the Adobe-GB1 character collection
	UniGB-UTF16-H UniGB-UTF16-V	Unicode (UTF-16BE) encoding for the Adobe-GB1 character collection. Contains mappings for all characters in the GB18030-2000 character set.
	GB-EUC-H GB-EUC-V	Microsoft Code Page 936 (charset 134), GB 2312-80 character set, EUC-CN encoding
	GBpc-EUC-H GBpc-EUC-V	Macintosh, GB 2312-80 character set, EUC-CN encoding, Script Manager code 2
	GBK-EUC-H, -V	Microsoft Code Page 936 (charset 134), GBK character set, GBK encoding
	GBKp-EUC-H GBKp-EUC-V	Same as GBK-EUC-H, but replaces half-width Latin characters with proportional forms and maps code 0x24 to dollar (\$) instead of yuan (¥).
	GBK2K-H, -V	GB 18030-2000 character set, mixed 1-, 2-, and 4-byte encoding
Traditional Chinese	UniCNS-UCS2-H UniCNS-UCS2-V	Unicode (UCS-2) encoding for the Adobe-CNS1 character collection
	UniCNS-UTF16-H UniCNS-UTF16-V	Unicode (UTF-16BE) encoding for the Adobe-CNS1 character collection. Contains mappings for all of HKSCS-2001 (2- and 4-byte character codes)
	B5pc-H, -V	Macintosh, Big Five character set, Big Five encoding, Script Manager code 2
	HKscs-B5-H HKscs-B5-V	Hong Kong SCS (Supplementary Character Set), an extension to the Big Five character set and encoding
	ETen-B5-H, -V	Microsoft Code Page 950 (charset 136), Big Five with ETen extensions
	ETenms-B5-H ETenms-B5-V	Same as ETen-B5-H, but replaces half-width Latin characters with proportional forms
	CNS-EUC-H, -V	CNS 11643-1992 character set, EUC-TW encoding
Japanese	UniJIS-UCS2-H, -V	Unicode (UCS-2) encoding for the Adobe-Japan1 character collection
	UniJIS-UCS2-HW-H UniJIS-UCS2-HW-V	Same as UniJIS-UCS2-H, but replaces proportional Latin characters with half-width forms
	UniJIS-UTF16-H UniJIS-UTF16-V	Unicode (UTF-16BE) encoding for the Adobe-Japan1 character collection. Contains mappings for all characters in the JIS X 0213:1000 character set.
	83pv-RKSJ-H	Mac, JIS X 0208 with KanjiTalk6 extensions, Shift-JIS, Script Manager code 1
	9oms-RKSJ-H 9oms-RKSJ-V	Microsoft Code Page 932 (charset 128), JIS X 0208 character set with NEC and IBM extensions
	9omsp-RKSJ-H 9omsp-RKSJ-V	Same as 9oms-RKSJ-H, but replaces half-width Latin characters with proportional forms
	9opv-RKSJ-H	Mac, JIS X 0208 with KanjiTalk7 extensions, Shift-JIS, Script Manager code 1
	Add-RKSJ-H, -V	JIS X 0208 character set with Fujitsu FMR extensions, Shift-JIS encoding
	EUC-H, -V	JIS X 0208 character set, EUC-JP encoding
	Ext-RKSJ-H, -V	JIS C 6226 (JIS78) character set with NEC extensions, Shift-JIS encoding
	H, V	JIS X 0208 character set, ISO-2022-JP encoding
Korean	UniKS-UCS2-H, -V	Unicode (UCS-2) encoding for the Adobe-Korea1 character collection
	UniKS-UTF16-H, -V	Unicode (UTF-16BE) encoding for the Adobe-Korea1 character collection
	KSC-EUC-H, -V	KS X 1001:1992 character set, EUC-KR encoding
	KSCms-UHC-H KSCms-UHC-V	Microsoft Code Page 949 (charset 129), KS X 1001:1992 character set plus 8822 additional hangul, Unified Hangul Code (UHC) encoding
	KSCms-UHC-HW-H KSCms-UHC-HW-V	Same as KSCms-UHC-H, but replaces proportional Latin characters with half-width forms
	KSCpc-EUC-H	Mac, KS X 1001:1992 with Mac OS KH extensions, Script Manager Code 3



These statements locate one of the Japanese standard fonts, choosing a Shift-JIS-compatible CMap (*Ext-RKSJ*) and horizontal writing mode (*H*). The *fontname* parameter must be the exact name of the font without any encoding or writing mode suffixes. The *encoding* parameter is the name of one of the supported CMaps (the choice depends on the font) and will also indicate the writing mode (see above). PDFlib supports all of Acrobat's default CMaps, and will complain when it detects a mismatch between the requested font and the CMap. For example, PDFlib will reject a request to use a Korean font with a Japanese encoding.

### 4.7.3 Custom CJK Fonts

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

- ▶ The font will be converted to a CID font and embedded in the PDF output regardless of the embedding setting provided by the client. Since PDFlib respects font embedding restrictions which may be defined in a font, fonts which do not allow embedding can not be used as custom CJK fonts.
- ▶ By default, font subsetting will be applied to all embedded custom CJK fonts; this can be controlled with various parameters, see Section 4.3, »Font Embedding and Subsetting«, page 78.
- ▶ Proportional Latin characters and half-width characters are fully supported for custom CJK fonts.
- ▶ Japanese host font names can be supplied to *PDF\_load\_font()* as UTF-8 with initial BOM, or UCS-2.

*Note* Original Composite Fonts (OCF) and raw PostScript CID fonts are not supported. Windows EUDC fonts are supported, but linking individual end-user defined characters into all fonts is not supported.

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

- ▶ On Windows, any code page installed on the system can be used. The code page number must be prefixed with *cp* (see Table 4.7 for examples). The *textformat* parameter must be set to *auto*, and the text must be supplied in a format which is compatible with the chosen code page.
- ▶ *unicode* encoding
- ▶ 8-bit encodings (although these are unlikely to be useful for CJK text)
- ▶ *glyphid* addressing (see Section 4.4.3, »Glyph ID Addressing for TrueType and OpenType Fonts«, page 87)

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

Table 4.7 Examples of CJK code pages on Windows (must be used with *textformat=auto*)

locale	code page	format	character set
Simplified Chinese	cp936	GBK	GBK
Traditional Chinese	cp950	Big Five	Big Five with Microsoft extensions
Japanese	cp932	Shift-JIS	JIS X 0208:1997 with Microsoft extensions
Korean	cp949	UHC	KS X 1001:1992, remaining 8822 hangul as extension
	cp1361	Johab	Johab

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

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

**Custom CJK font example with Japanese Shift-JIS text.** The following example uses the MS Mincho font to display some Japanese text which is supplied in Shift-JIS format according to Windows code page 932:

```
font = PDF_load_font(p, "MS Mincho", 0, "cp932", "");

PDF_setfont(p, font, 24);
PDF_set_text_pos(p, 50, 500);

PDF_show2(p, "\x82\xA9\x82\xC8\x8A\xBF\x8E\x9A", 8);
```

**Custom CJK font example with Chinese Unicode text.** The following example uses the ArialUnicodeMS font to display some Chinese text. The font must either be installed on the system or must be configured according to Section 4.3.1, »How PDFlib Searches for Fonts«, page 78):

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

PDF_setfont(p, font, 24);
PDF_set_text_pos(p, 50, 500);

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

**End-user defined characters (EUDC).** PDFlib does not support linking end-user defined characters into fonts, but you can use the EUDC editor available in Windows to create custom characters for use with PDFlib. Proceed as follows:

- ▶ Use the *eudcedit.exe* to create one or more custom characters at the desired Unicode position(s).
- ▶ Locate the *EUDC.TTE* file in the directory *\Windows\fonts* and copy it to some other directory. Note that this file will be invisible in Windows Explorer, but you can use the *dir* and *copy* commands in a DOS box to find the file. Now configure the font for use with PDFlib, using one of the methods discussed in Section 4.3.1, »How PDFlib Searches for Fonts«, page 78):

```
PDF_set_parameter(p, "FontOutline", "EUDC=EUDC.TTE")
PDF_set_parameter(p, "SearchPath", "...directory name...")
```

or place *EUDC.TTE* in the current directory.

- ▶ As an alternative to the preceding step you can use the following function call to configure the font file directly from the Windows directory. This way you will always access the current EUDC font used in Windows:

```
PDF_set_parameter(p, "FontOutline", "EUDC=C:\Windows\fonts\EUDC.TTE")
```

- Use the following call to load the font:

```
font = PDF_load_font(p, "EUDC", 0, "unicode", "")
```

and supply the Unicode character codes chosen in the first step to output the character.

#### 4.7.4 Forcing monospaced Fonts

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

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

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

## 4.8 Placing and Fitting Single-Line Text

The function `PDF_fit_textline()` for placing a single line of text on a page offers a wealth of formatting options. The most important options will be discussed in this section using some common application examples. A complete description of these options can be found in Table 8.17. Most options for `PDF_fit_textline()` are identical to those of `PDF_fit_image()`. Therefore we will only use text-related examples here; it is recommended to take a look at the examples in Section 5.3, »Placing Images and Imported PDF Pages«, page 136, for an introduction.

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

`PDF_fit_textline()` uses the so-called text box to determine the positioning of the text: the width of the text box is identical to the width of the text, and the box height is identical to the height of capital letters in the font. The text box can be extended to the left and right or top and bottom using the *margin* option. The margin will be scaled along with the text line.

### 4.8.1 Simple Text Placement

**Placing text in the bottom center.** We place text at the reference point such that the text box will be positioned with the center of its bottom line at the reference point (see Figure 4.4):

```
PDF_fit_textline(p, text, 297, 0, "position {50 0}");
```

This code fragment places the text box with the bottom center (*position {50 0}*) at the reference point (297, 0).

**Placing text in the top right corner.** Now we place the text at the reference point such that the text box will be placed with the upper right corner at the reference point (see Figure 4.5):

```
PDF_fit_textline(p, text, 595, 842, "position 100");
```

Fig. 4.4  
Placing text in the  
bottom center



Fig. 4.5  
Placing text in the upper  
right corner



This code fragment places the text box with the upper right corner (*position 100*) at the reference point (595, 842).

**Placing text with a margin.** To extend the previous example we can add a horizontal margin to the text to achieve a certain distance to the right. This may be useful for placing text in table columns:

```
PDF_fit_textline(p, text, 595, 842, "position 100 margin {20 0}");
```

## 4.8.2 Placing Text in a Box

**Placing centered text in a box.** We define a box and place the text centered within the box (see Figure 4.6):

```
PDF_fit_textline(p, text, 10, 200, "boxsize {500 220} position 50");
```

This code fragment places the text centered (*position 50*) in a box with the lower left corner at (10, 200), 500 units wide and 220 units high (*boxsize {500 220}*).

**Proportionally fitting text to a box.** We extend the previous example and fit the text into the box completely (see Figure 4.7):

```
PDF_fit_textline(p, text, 10, 200, "boxsize {500 220} position 50 fitmethod meet");
```

Note that the font size will be changed when text is fit into the box with *fitmethod meet*. In order to prevent the text from being scaled up use *auto* instead of *meet*.

**Completely fitting text to a Box.** We can further modify the previous example such that the text will not be fit into the box proportionally, but completely covers the box. However, this combination will only rarely be used since the text may be distorted (see Figure 4.8):

```
PDF_fit_textline(p, text, 10, 200, "boxsize {500 220} position 50 fitmethod entire");
```

Fig. 4.6  
Placing centered text in a box



Fig. 4.7  
Proportionally fitting text to a box

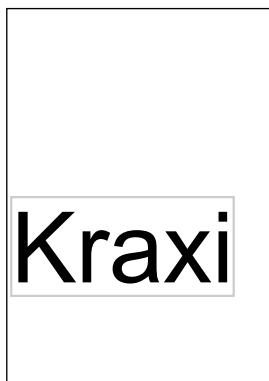
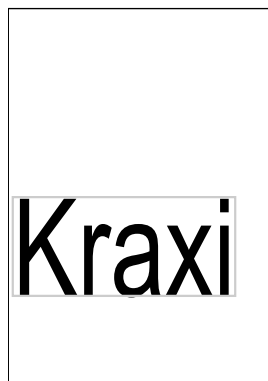


Fig. 4.8  
Completely fitting text to a box



### 4.8.3 Aligning Text

**Simple alignment.** Our next goal is to rotate text such that its original lower left corner will be placed at a given reference point (see Figure 4.9). This may be useful, for example, for placing a rotated column heading in a table header:

```
PDF_fit_textline(p, text, 5, 5, "orientate west");
```

This code fragment orientates the text to the west (90° counterclockwise) and then translates it the lower left corner of the rotated text to the reference point (5, 5).

**Aligning text at a vertical line.** Positioning text along a vertical line (i.e., a box with zero width) is a somewhat extreme case which may be useful nevertheless (see Figure 4.10):

```
PDF_fit_textline(p, text, 0, 0, "boxsize {0 600} position {0 50} orientate west");
```

This code fragment rotates the text, and places it at the center of the line from (0, 0) to (0, 600).

Fig. 4.9  
Simple Aligning

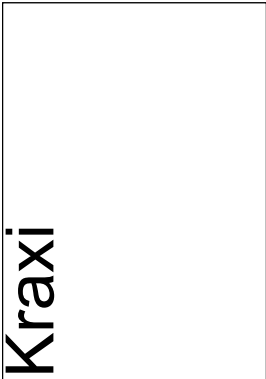
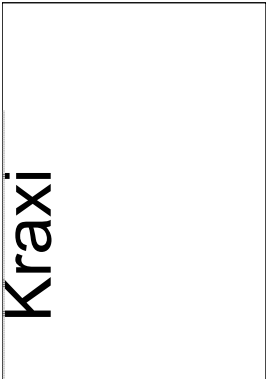


Fig. 4.10  
Aligning text at a vertical line



# 4.9 Multi-Line Textflows

In addition to placing single lines of text on the page, PDFlib supports a feature called Textflow which can be used to place arbitrarily long text portions. The text may extend across any number of lines or pages, and its appearance can be controlled with a variety of options. Character properties such as font, size, and color can be applied to any portion of the text. Textflow properties such as justified or ragged text, paragraph indentation and tab stops can be specified; line breaking opportunities designated by soft hyphens in the text will be taken into account. Figure 4.11 and Figure 4.12 demonstrate how various parts of an invoice can be placed on the page using the textflow feature. We will discuss the options for controlling the output in more detail in the following sections.

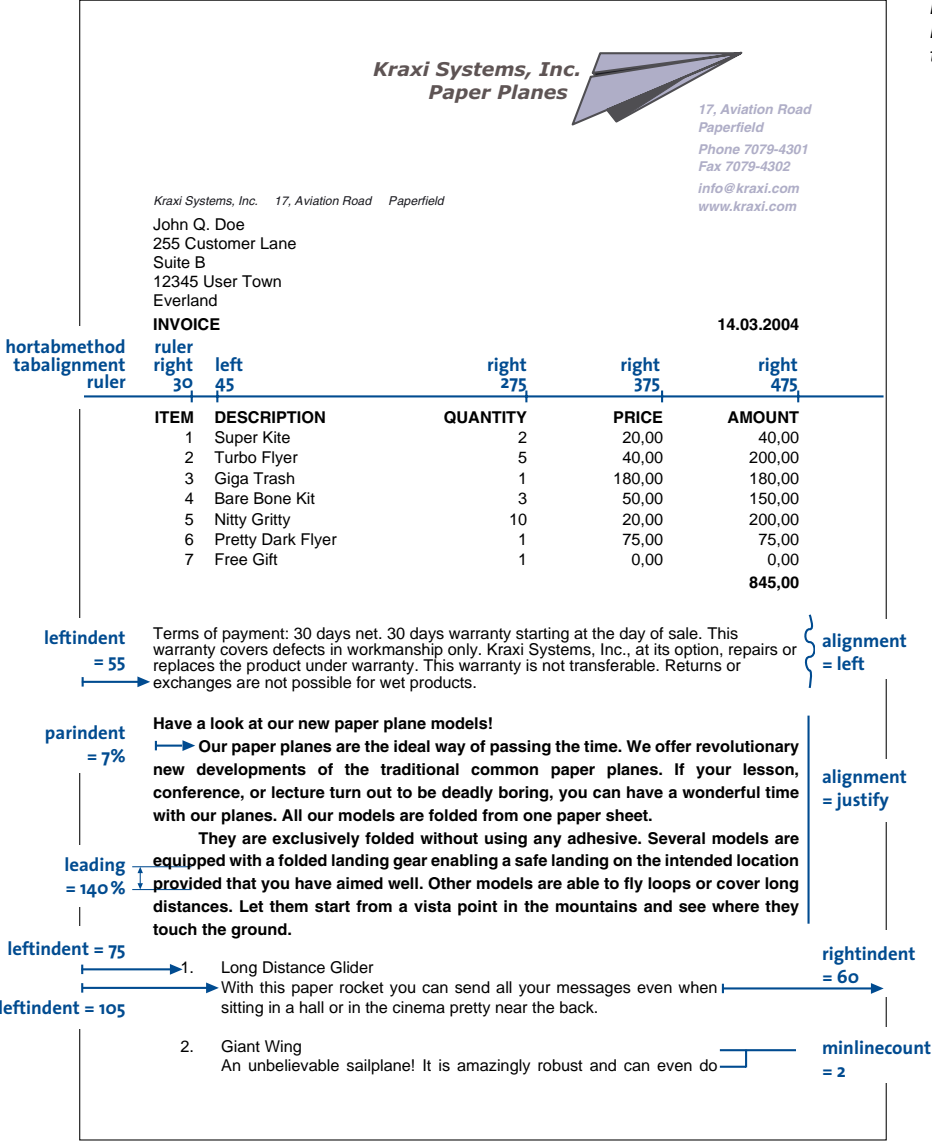
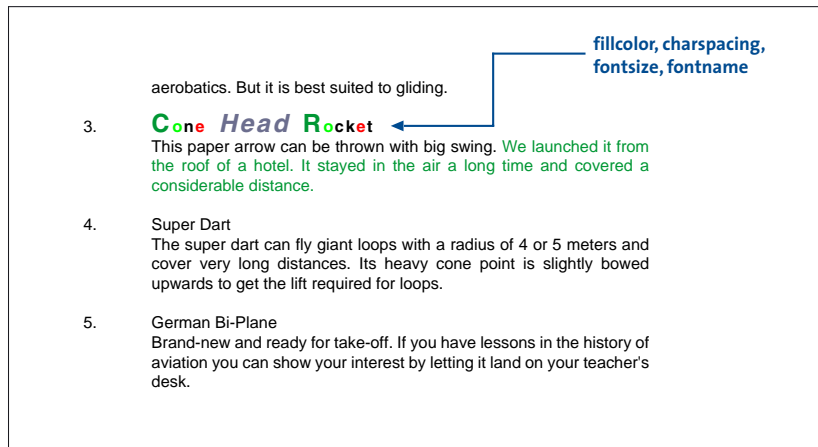


Fig. 4.11  
Formatting  
text flows

Fig. 4.12  
Formatting  
text flows



A multi-line Textflow can be placed into one or more rectangles (so-called fitboxes) on one or more pages. The following steps are required for placing a textflow on the page:

- ▶ The function `PDF_create_textflow()` analyzes the text, creates a textflow object and returns a handle. It does not place any text on the page.
- ▶ The function `PDF_fit_textflow()` places all or parts of the textflow in the supplied fitbox. To completely place the text this step must possibly be repeated several times where each of the function calls provides a new fitbox which may be located on the same or another page.
- ▶ The function `PDF_delete_textflow()` deletes the Textflow object after it has been placed in the document.

The function `PDF_create_textflow()` for creating Textflows supports a variety of options for controlling the formatting process. These options can be provided in the function's option list, or embedded as *inline* options in the text. We will discuss Textflow placement using some common application examples. A complete list of Textflow options can be found in Table 8.21.

Many of the options supported in `PDF_create_textflow()` are identical to those of `PDF_fit_textline()`. It is therefore recommended to familiarize yourself with the examples in Section 4.8, »Placing and Fitting Single-Line Text«, page 108. In the next sections we will focus on options related to multi-line text.

## 4.9.1 Placing Textflows in the Fitbox

**Placing text in a single fitbox.** Let's start with an easy example. The following code fragment places a Textflow in a single fitbox on the page using default formatting options. Font, font size, and encoding have been specified explicitly (you can see the result in Figure 4.13):

```
textflow =
    PDF_create_textflow(p, text, 0, "fontname=Helvetica fontsize=9 encoding=winansi");
PDF_fit_textflow(p, textflow, left_x + offset, left_y, right_x + offset, right_y, "");
PDF_delete_textflow(p, textflow);
```

**Placing text in two fitboxes.** If the text placed on the page with `PDF_fit_textflow()` doesn't fit into the fitbox, the output will be interrupted and the function will return



Fig. 4.13  
Simple textflow  
placement

Terms of payment: 30 days net. 30 days warranty starting at the day of sale. This warranty covers defects in workmanship only. Kraxi Systems, Inc., at its option, repairs or replaces the product under warranty. This warranty is not transferable. Returns or exchanges are not possible for wet products.

the string `_boxfull`. PDFlib will remember the amount of text already placed on the page, and will continue with the remainder of the text when the function is called again. The following code fragment demonstrates how to place a Textflow in two fit-boxes (you can see the result in Figure 4.14):

```
textflow =
    PDF_create_textflow(p, text, 0, "fontname=Helvetica fontsize=9 encoding=winansi");
result = PDF_fit_textflow(p, textflow, left_x, left_y, right_x, right_y, "");
/* Check whether the text could be fully placed in the fitbox */
if (!strcmp(result, "_boxfull"))
{
    y = PDF_get_value(p, "texty", 0);
    PDF_fit_textflow(p, textflow, left_x, left_y, right_x, right_y, "");
}
PDF_delete_textflow(p, textflow);
```

**Placing text on multiple pages.** If the text placed with `PDF_fit_textflow()` doesn't fully fit into the fitbox, it may be necessary to create a new page. The fundamental code for placing a textflow across multiple pages looks as follows:

```
textflow = PDF_create_textflow(p, text, 0, optlist);
while (1)
{
    PDF_begin_page_ext(p, pagewidth, pageheight, "");
    result = PDF_fit_textflow(p, textflow, left_x, left_y, right_x, right_y, "");
    PDF_end_page_ext(p, "");
    if (strcmp(result, "_boxfull"))
        break;
}
PDF_delete_textflow(p, textflow);
```

## 4.9.2 Paragraph Formatting Options

In the previous example we used default settings for the paragraphs. For example, the default alignment is left-justified, and the leading is 100% (which equals the font size).

In order to fine-tune the paragraph formatting we can feed more options to `PDF_create_textflow()`. For example, we can indent the text 15 units from the left and 10 units from the right margin. In addition, the first line of each paragraph should be indented

### first fitbox

Terms of payment: 30 days net. 30 days warranty starting at the day of sale. This warranty covers defects in workmanship only. Kraxi Systems, Inc., at its option, repairs or

### second fitbox

replaces the product under warranty. This warranty is not transferable. Returns or exchanges are not possible for wet products.

Fig. 4.14  
Placing a Textflow in  
two fitboxes

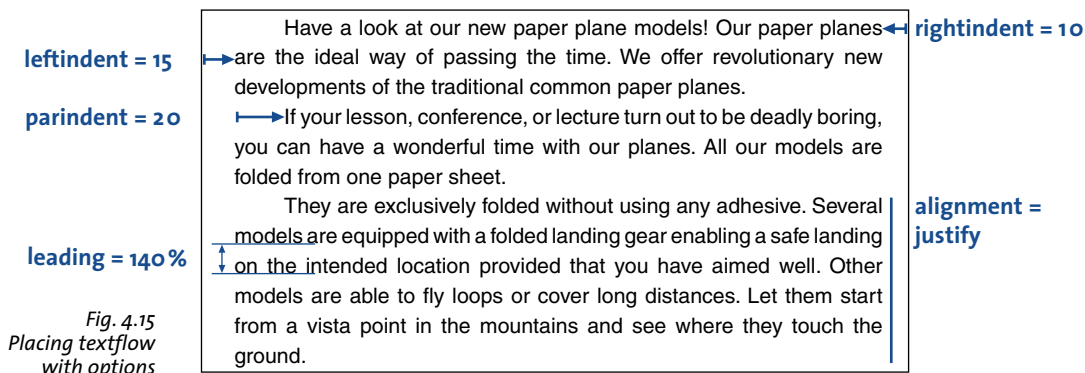


Fig. 4.15  
Placing textflow  
with options

by 10 units. The text should be justified against both margins, and the leading increased to 140%. Finally, we'll reduce the font size to 8 units. The extended code with option list to achieve this looks as follows (you can see the result in Figure 4.15):

```
char    optlist[] =
        "leftindent=15 rightindent=10 parindent=20 alignment=justify "
        "leading=140% fontname=Helvetica fontsize=8 encoding=winansi"

/* place textflow in the fitbox using the options */
textflow = PDF_create_textflow(p, text, 0, optlist);
PDF_fit_textflow(p, textflow, left_x, left_y, right_x, right_y, "");
PDF_delete_textflow(p, textflow);
```

### 4.9.3 Inline Option Lists and Macros

The text in Figure 4.15 is not yet correct. The headline »Have a look at our new paper plane models!« should sit on a line of its own, should use a larger font size, and should be centered. There are several ways to achieve this.

Up to now we provided formatting options in an option list supplied directly to the function. In order to continue the same way we would have to split the text, and place it in two separate calls, one for the headline and another one for the remaining text. However, this would be cumbersome.

For this reason the textflow feature supports so-called inline options. This simply means that the options are embedded in the text. Inline option lists are provided as part of the body text. By default, they are delimited by »<<« and »>>« characters. We will therefore integrate the options for formatting the heading and the remaining paragraphs into our body text as follows (inline option lists are colored in all subsequent samples; end-of-paragraph characters are visualized with arrows):

```
<leftindent=15 rightindent=10 alignment=center fontname=Helvetica fontsize=12
encoding=winansi>Have a look at our new paper plane models! ◀
<alignment=justify fontname=Helvetica leading=140% fontsize=8 encoding=winansi>
Our paper planes are the ideal way of passing the time. We offer
revolutionary new developments of the traditional common paper planes. ◀
<parindent=20>If your lesson, conference, or lecture
turn out to be deadly boring, you can have a wonderful time
with our planes. All our models are folded from one paper sheet. ◀
They are exclusively folded without using any adhesive. Several
models are equipped with a folded landing gear enabling a safe
landing on the intended location provided that you have aimed well.
```

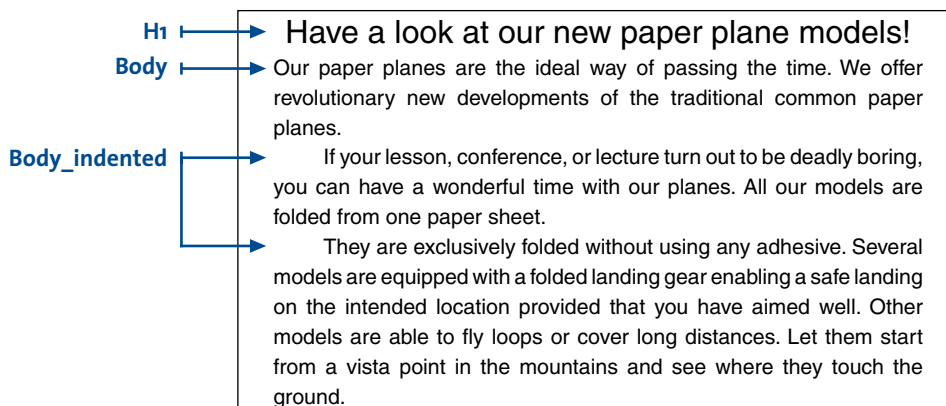


Fig. 4.16  
Combining inline  
options with macros

Other models are able to fly loops or cover long distances. Let them start from a vista point in the mountains and see where they touch the ground.

The characters for bracketing option lists can be redefined with the *begoptlistchar* and *endoptlistchar* options (see Table 8.21). Supplying the keyword *none* for the *begoptlistchar* option completely disables the search for option lists. This is useful if the text doesn't contain any inline option lists, and you want to make sure that »« and »»« will be processed as regular characters.

**Macros.** Basically, the text above contains several different types of paragraphs, such as heading or body text with or without indentation. Each of these paragraph types is formatted individually and occurs multiply in longer Textflows. In order to avoid starting each paragraph with the corresponding inline options, we can combine these to form macros, and refer to the macros in the text via their names. As shown in Figure 4.16 we can define three macros called *H1* for the heading, *Body* for main paragraphs, and *Body\_indented* for indented paragraphs. In order to use a macro we place the & character in front of its name and put it into an option list. The following code fragment defines three macros according to the previously used inline options and uses them in the text:

```
<macro {
H1 {leftindent=15 rightindent=10 alignment=center
fontname=Helvetica fontsize=12 encoding=winansi}

Body {leftindent=15 rightindent=10 alignment=justify leading=140%
fontname=Helvetica fontsize=8 encoding=winansi}

Body_indented {parindent=20 leftindent=15 rightindent=10 alignment=justify
leading=140% fontname=Helvetica fontsize=8 encoding=winansi}
}>
<&H1>Have a look at our new paper plane models! ␣
<&Body>Our paper planes are the ideal way of passing the time. We offer
revolutionary new developments of the traditional common paper planes. ␣
<&Body_indented>If your lesson, conference, or lecture
turn out to be deadly boring, you can have a wonderful time
with our planes. All our models are folded from one paper sheet. ␣
They are exclusively folded without using any adhesive. Several
models are equipped with a folded landing gear enabling a safe
```

landing on the intended location provided that you have aimed well. Other models are able to fly loops or cover long distances. Let them start from a vista point in the mountains and see where they touch the ground.

**Explicitly setting options.** Note that all options which are not set in macros will retain their previous values. In order to avoid side effects caused by unwanted »inheritance« of options you should explicitly specify all settings required for a particular macro. This way you can ensure that the macros will behave consistently regardless of their ordering or combination with other option lists.

On the other hand, you can take advantage of this behavior for deliberately retaining certain settings from the context instead of supplying them explicitly. For example, a macro could specify the font name without supplying the *fontsize* option. As a result the font size will always match that of the preceding text.

**Inline Options or Options passed as Function Parameters?** When using Textflows it makes an important difference whether the text is contained literally in the program code or comes from some external source, and whether the formatting instructions are separate from the text or part of it. In most applications the actual text will come from some external source such as a database. In practise there are two main scenarios:

- ▶ Text contents from external source, formatting options in the program: An external source delivers small text fragments which are assembled within the program, and combined with formatting options (in the function call) at runtime.
- ▶ Text contents and formatting options from external source: Large amounts of text including formatting options come from an external source. The formatting is provided by inline options in the text, represented as simple options or macros. When it comes to macros a distinction must be made between macro definition and macro call. This allows an interesting intermediate form: the text content comes from an external source and contains macro calls for formatting. However, the macro definitions are only blended in at runtime. This has the advantage that the formatting can easily be changed without having to modify the external text. For example, when generating greeting cards one could define different styles via macros to give the card a romantic, technical, or other touch.

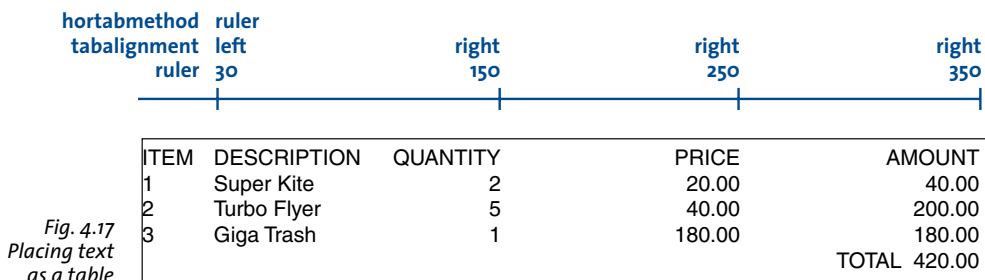
#### 4.9.4 Tab Stops

In the next example we will place a table with left- and right-aligned columns using tab characters. The table contains the following lines of text, where individual entries are separated from each other with a tab character (indicated by arrows):

```
ITEM → DESCRIPTION → QUANTITY → PRICE → AMOUNT ←
1 → Super Kite → 2 → 20.00 → 40.00 ←
2 → Turbo Flyer → 5 → 40.00 → 200.00 ←
3 → Giga Trash → 1 → 180.00 → 180.00 ←
→ → → → TOTAL 420.00
```

The following code fragment places the table, using the *ruler* option for defining the tab positions, *tabalignment* for specifying the alignment of tab stops, and *hortabmethod* for specifying the method used to process tab stops (the result can be seen in Figure 4.17):

```
/* assemble option list */
char optlist[] =
```



```
"ruler      {30    150  250  350} "
"tabalignment {left right right right} "
"hortabmethod ruler leading=120% fontname=Helvetica fontsize=9 encoding=winansi";

/* place textflow in fitbox */
textflow = PDF_create_textflow(p, table, 0, optlist);
PDF_fit_textflow(p, textflow, left_x, left_y, right_x, right_y, "");
PDF_delete_textflow(p, textflow);
```

## 4.9.5 Numbered Lists

The following example demonstrates how to format a numbered list using the inline option *leftindent* (you can see the result in Figure 4.18):

```
1.<leftindent 10>Long Distance Glider: With this paper rocket you can send all
your messages even when sitting in a hall or in the cinema pretty near the back. ◀
<leftindent 0>2.<leftindent 10>Giant Wing: An unbelievable sailplane! It is amazingly
robust and can even do aerobatics. But it is best suited to gliding. ◀
<leftindent 0>3.<leftindent 10>Cone Head Rocket: This paper arrow can be thrown with big
swing. We launched it from the roof of a hotel. It stayed in the air a long time and
covered a considerable distance.
```

Setting and resetting the indentation value is cumbersome, especially since it is required for each paragraph. A more elegant solution defines a macro called *list*. For convenience it defines a macro *indent* which is used as a constant. The macro definitions are as follows:

```
<macro {
indent {25}

list {parindent=-&indent leftindent=&indent hortabsize=&indent
hortabmethod=ruler ruler={&indent}}
}>
<&list>1. → Long Distance Glider: With this paper rocket you can send all your messages
even when sitting in a hall or in the cinema pretty near the back. ◀
```

1. Long Distance Glider: With this paper rocket you can send all your messages even when sitting in a hall or in the cinema pretty near the back.
2. Giant Wing: An unbelievable sailplane! It is amazingly robust and can even do aerobatics. But it is best suited to gliding.
3. Cone Head Rocket: This paper arrow can be thrown with big swing. We launched it from the roof of a hotel. It stayed in the air a long time and covered a considerable distance.

Fig. 4.18  
Numbered list







**leftindent = &indent**      
**parindent = - &indent**

Fig. 4.19  
Numbered list  
with macros

1.  Long Distance Glider: With this paper rocket you can send all your messages even when sitting in a hall or in the cinema pretty near the back.
2. Giant Wing: An unbelievable sailplane! It is amazingly robust and can even do aerobatics. But it is best suited to gliding.
3.  Cone Head Rocket: This paper arrow can be thrown with big swing. We launched it from the roof of a hotel. It stayed in the air a long time and covered a considerable distance.

2.  Giant Wing: An unbelievable sailplane! It is amazingly robust and can even do aerobatics. But it is best suited to gliding. 
3.  Cone Head Rocket: This paper arrow can be thrown with big swing. We launched it from the roof of a hotel. It stayed in the air a long time and covered a considerable distance.

The *leftindent* option specifies the distance from the left margin. The *parindent* option, which is set to the negative of *leftindent*, cancels the indentation for the first line of each paragraph. The options *horthabsize*, *horthabmethod*, and *ruler* specify a tab stop which corresponds to *leftindent*. It makes the text after the number to be indented with the amount specified in *leftindent*. Figure 4.19 shows the *parindent* and *leftindent* options at work.

## 4.9.6 Control Characters, Character Mapping, and Symbol Fonts

**Control characters in Textflows.** Various characters are given special treatment in Textflows. PDFlib supports symbolic character names which can be used instead of the corresponding character codes in the *charmapping* option (which replaces characters in the text before processing it, see below). Table 4.8 lists all control characters which are evaluated by the Textflow functions along with their symbolic names, and explains their meaning. An option must only be used once per option list, but multiple option lists can be provided one after the other. For example, the following sequence will create an empty line:

```
<nextline><nextline>
```

**Replacing characters or sequences of characters.** The *charmapping* option can be used to replace some characters in the text with others. Let's start with an easy case where we will replace all tabs in the text with space characters. The *charmapping* option to achieve this looks as follows:

```
charmapping {horthab space}
```

This command uses the symbolic character names *horthab* and *space*. You can find a list of all known character names in Table 4.8. To achieve multiple mappings at once we can use the following command which will replace all tabs and line break combinations with space characters:

```
charmapping {horthab space    CRLF space    LF space    CR space}
```

The following command removes all soft hyphens:

```
charmapping {shy {shy 0}}
```

Table 4.8 Control characters and their meaning in textflows

Unicode character	symbolic name for charmapping	equivalent option	meaning within textflows in Unicode-compatible fonts
U+0020	SP, space	space	align words and break lines
U+00A0	NBSP, nbsp	(none)	(no-break space) space character which will not break lines
U+0009	HT, hortab	(none)	horizontal tab: will be processed according to the ruler, tabalignchar, and tabalignment options
U+002D	HY, hyphen	(none)	separator character for hyphenated words
U+00AD	SHY, shy	(none)	(soft hyphen) hyphenation opportunity, only visible at line breaks
U+000B U+2028	VT, verrtab LS, linesep	nextline	(next line) forces a new line
U+000A U+000D U+000D and U+000A U+0085 U+2029	LF, linefeed CR, return CRLF  NEL, newline PS, parasep	next- paragraph	(next paragraph) Same effect as »next line«; in addition, the parindent option will affect the next line.
U+000C	FF, formfeed	return	end of a paragraph; the function PDF_fit_textflow() will return the string _nextpage.

Each tab character will be replaced with four space characters:

```
charmapping {hortab {space 4}}
```

Each arbitrary long sequence of linefeed characters will be reduced to a single linefeed character:

```
charmapping {linefeed {linefeed -1}}
```

Each sequence of CRLF combinations will be replaced with a single space:

```
charmapping {CRLF {space -1}}
```

We will take a closer look at the last example. Let's assume you receive text where the lines have been separated with fixed line breaks by some other software, and therefore cannot be properly formatted. You want to replace the linebreaks with space characters in order to achieve proper formatting within the fitbox. To achieve this we replace arbitrarily long sequences of linebreaks with a single space character. The initial text looks as follows:

```
To fold the famous rocket looper proceed as follows: ↵ ↵
Take a sheet of paper. Fold it ↵
lengthwise in the middle. ↵
Then, fold down the upper corners. Fold the ↵
long sides inwards ↵
that the points A and B meet on the central fold.
```

The following code fragment demonstrates how to replace the redundant linebreak characters and format the resulting text:

```
/* assemble option list */
char optlist[] = "fontname=Helvetica fontsize=9 encoding=winansi alignment=justify "
```

To fold the famous rocket looper proceed as follows:

Take a sheet of paper. Fold it lengthwise in the middle. Then, fold down the upper corners. Fold the long sides inwards that the points A and B meet on the central fold.

Fig. 4.20  
Top: text with redundant line breaks

To fold the famous rocket looper proceed as follows: Take a sheet of paper. Fold it lengthwise in the middle. Then, fold down the upper corners. Fold the long sides inwards that the points A and B meet on the central fold.

Bottom: replacing the linebreaks with the *charmapping* option

```
"charmapping {CRLF {space -1}}"  
/* place textflow in fitbox */  
textflow = PDF_create_textflow(p, text, 0, optlist);  
PDF_fit_textflow(p, textflow, left_x, left_y, right_x, right_y, "");  
PDF_delete_textflow(p, textflow);
```

Figure 4.20 shows Textflow output with the unmodified text and the repaired version with the *charmapping* option.

**Symbol fonts in textflows.** Symbol fonts, more precisely: text in a font which is not Unicode-compatible according to Section 4.5.6, »Unicode-compatible Fonts«, page 95, deserves some special attention when used within textflows:

- ▶ The control characters listed in Table 4.8 will not be treated specially, i.e. they have no special meaning.
- ▶ Some textflow options will be ignored since they do not make sense for symbol fonts, e.g. `tabalignchar`. Table 8.21 lists all options which will be ignored for fonts which are not Unicode-compatible.
- ▶ Since inline options lists cannot be used in text portions with symbol fonts (since the symbols don't have any intrinsic meaning it would be impossible to locate and interpret option lists), the length of text fragments consisting of symbol characters must explicitly be specified using the *textlen* option.
- ▶ After *textlen* characters a new inline option list must be placed in the text which switches to another font/encoding combination.

The following text contains a single glyph from the Symbol font inserted between Latin characters:

```
<fontname=Helvetica fontsize=12 encoding=winansi>The Greek letter  
<fontname=Symbol encoding=builtin textlen=1>A  
<fontname=Helvetica encoding=winansi> symbolizes beginning.
```

Omitting the *textlen* option for Symbol fragments, or failing to supply another inline option lists immediately after the Symbol fragment will result in an exception.



Our paper planes are the ideal way of passing the time. We offer revolutionary brand new developments of the traditional common paper planes. If your lesson, conference, or lecture turn out to be deadly boring, you can have a wonderful time with our planes. All our models are folded from one paper sheet. They are exclusively folded without using any adhesive. Several models are equipped with a folded landing gear enabling a safe landing on the intended location provided that you have aimed well. Other models are able to fly loops or cover long distances. Let them start from a vista point in the mountains and see where they touch the ground.

Fig. 4.21  
Justified text with soft hyphen characters,  
using default settings and a wide fitbox

Our paper planes are the ideal way of passing the time. We offer revolutionary brand new developments of the traditional common paper planes. If your lesson, conference, or lecture turn out to be deadly boring, you can have a wonderful time with our planes. All our models are folded from one paper sheet. They are exclusively folded without using any adhesive. Several models are equipped with a folded landing gear enabling a safe landing on the intended location provided that you have aimed well. Other models are able to fly loops or cover long distances. Let them start from a vista point in the mountains and see where they touch the ground.

Fig. 4.22  
Justified text without soft hyphens, using  
default settings and a wide fitbox.

## 4.9.7 Hyphenation

PDFlib does not automatically hyphenate text, but honors possible hyphenation opportunities if those are explicitly designated in the text by soft hyphen characters. The soft hyphen character is at position `U+00AD` in Unicode, but several methods are available for specifying the soft hyphen in non-Unicode environments:

- ▶ In all *cp1250* – *cp1258* (including *winansi*) and *iso8859-1* – *iso8859-16* encodings the soft hyphen is at decimal 173, octal 255, or hexadecimal `0xAD`.
- ▶ In *ebcdic* encoding the soft hyphen is at decimal 202, octal 312, or hexadecimal `0xCA`.
- ▶ A character entity reference (see Section 4.5.5, »Character References«, page 94) can be used if an encoding does not contain the soft hyphen character (e.g. *macroman*):  
`&shy;`

In addition to breaking opportunities designated by soft hyphens, words can be forcefully hyphenated in extreme cases when other methods of adjustment, such as changing the wordspacing or shrinking text are not possible.

**Justified text with or without hyphen characters.** In the following example we will print the following text with justified alignment. The text contains soft hyphen characters (visualized here as dashes):

Our paper planes are the ideal way of pas - sing the time. We offer revolu - tionary brand new dev - elop - ments of the tradi - tional common paper planes. If your lesson, confe - rence, or lecture turn out to be deadly boring, you can have a wonder - ful time with our planes. All our models are folded from one paper sheet. They are exclu - sively folded without using any adhe - sive. Several models are equip - ped with a folded landing gear enab - ling a safe landing on the intended loca - tion provided that you have aimed well. Other models are able to fly loops or cover long dist - ances. Let them start from a vista point in the mount - ains and see where they touch the ground.

Figure 4.21 shows the generated text output with default option settings for justified text. It looks perfect since the conditions are optimal: the fitbox is wide enough, and

there are explicit break opportunities specified by the soft hyphen characters. As you can see in Figure 4.22 the output looks okay even without explicit soft hyphens. The option list in both cases looks as follows:

```
fontname=Helvetica fontsize=9 encoding=winansi alignment=justify
```

## 4.9.8 Controlling the Linebreak Algorithm

PDFlib implements a sophisticated line-breaking algorithm.<sup>1</sup> Table 4.9 lists textflow options which control the line-breaking algorithm.

Table 4.9 Options for controlling the line-breaking algorithm

option	type	explanation
adjust-method	keyword	The method used to adjust a line when a text portion doesn't fit into a line after compressing or expanding the distance between words subject to the limits specified by the minspacing and maxspacing options. Default: auto
		auto The following methods are applied in order: shrink, spread, nofit, split.
		clip Same as nofit, except that the long part at the right edge of the fit box (taking into account the rightindent option) will be clipped.
		nofit The last word will be moved to the next line provided the remaining (short) line will not be shorter than the percentage specified in the nofitlimit option. Even justified paragraphs will look slightly ragged in this case.
		shrink If a word doesn't fit in the line the text will be compressed subject to the shrinklimit option until the word fits. If it still doesn't fit the nofit method will be applied.
		split The last word will not be moved to the next line, but will forcefully be hyphenated. For text fonts a hyphen character will be inserted, but not for symbol fonts.
		spread The last word will be moved to the next line and the remaining (short) line will be justified by increasing the distance between characters in a word, subject to the spreadlimit option. If justification still cannot be achieved the nofit method will be applied.
avoidbreak	boolean	If true, avoid any line breaks until avoidbreak is reset to false. Default: false
hyphenchar	integer	Unicode value of the character which replaces a soft hyphen at line breaks. Default: U+00AD (SOFT HYPHEN) if available in the font, U+002D (HYPHEN-MINUS) otherwise
maxspacing minspacing	float or percentage	Specifies the maximum or minimum distance between words (in user coordinates, or as a percentage of the width of the space character). The calculated word spacing is limited by the provided values (but the wordspacing option will still be added). Defaults: minspacing=50%, maxspacing=500%
nofitlimit	float or percentage	Lower limit for the length of a line with the nofit method (in user coordinates or as a percentage of the width of the fitbox). Default: 75%.
shrinklimit	percentage	Lower limit for compressing text with the shrink method; the calculated shrinking factor is limited by the provided value, but will be multiplied with the value of the horzscaling option. Default: 85%
spreadlimit	float or percentage	Upper limit for the distance between two characters for the spread method (in user coordinates or as a percentage of the font size); the calculated character distance will be added to the value of the charspacing option. Default: 0

1. For interested users we'll note that PDFlib honors the recommendations in »Unicode Standard Annex #14: Line Breaking Properties« (see [www.unicode.org/reports/tr14](http://www.unicode.org/reports/tr14)), but ignores the breaking rules for characters beyond U+20D0, i.e. these are treated as characters without breaking opportunity. Combining marks are not taken into account.

Our paper planes  
are the ideal way of  
passing the time. We  
offer revolutionary  
brand new develop-  
ments of the traditional  
common paper planes.  
If your lesson, conf-  
erence, or lecture  
turn out to be deadly  
boring, you can have  
a wonderful time  
with our planes. All  
our models are  
folded from one  
paper sheet. They  
are exclusively folded  
without using any

decrease the distance between words (default method, minspacing option)

compress the line (shrink method, shrinklimit option)

force hyphenation (split method)

increase the distance between words (default method, maxspacing option)

Fig. 4.23

Justified text in a narrow fitbox with default settings

**Line-breaking rules.** When a word or other sequence of text surrounded by space characters doesn't fully fit into a line, it must be moved to the next line. In this situation the line-breaking algorithm decides after which characters a line break is possible.

For example, a formula such as  $-12+235/8*45$  will never be broken, while the string `PDF-345+LIBRARY` may be broken to the next line at the minus character. If the text contains soft hyphen characters it can also be broken after such a character.

For parentheses and quotation marks it depends on whether we have an opening or closing character: opening parentheses and quotations marks do not offer any break opportunity. In order to find out whether a quotation mark starts or ends a sequence, pairs of quotation marks are examined.

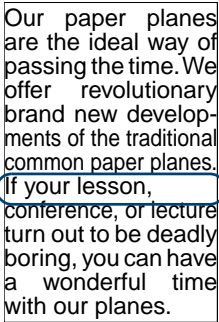
**Justified text in a narrow fitbox.** The narrower the fitbox, the more you must deal with the options for controlling justified text. Figure 4.23 demonstrates the results of the various methods for justifying text in a narrow fitbox. The option settings in Figure 4.23 are basically okay, with the exception of *maxspacing* which provides a rather large distance between words. However, it is recommended to keep this for narrow fitboxes

Our paper planes  
are the ideal way of  
passing the time. We  
offer revolutionary  
brand new develop-  
ments of the traditional  
common paper planes.  
If your lesson, conference,  
or lecture turn out to  
be deadly boring,  
you can have a  
wonderful time with  
our planes. All our  
models are folded  
from one paper  
sheet. They are  
exclusively folded without  
using any adhesive.

Fig. 4.24

Compressing lines down to 50%

compress the line (shrink method, shrinklimit option)



Our paper planes  
are the ideal way of  
passing the time. We  
offer revolutionary  
brand new develop-  
ments of the traditional  
common paper planes.  
If your lesson,  
conference, or lecture  
turn out to be deadly  
boring, you can have  
a wonderful time  
with our planes.

shorten the line (nofit method, nofitlimit option)

Fig. 4.26  
Justified text with minimum width of 50%

since otherwise the ugly forced hyphenation caused by the *split* method will occur more often.

If the fitbox is so narrow that occasionally forced hyphenations occur you should consider inserting soft hyphens, or modify the options which control justified text.

**Option shrinklimit for justified text.** The most visually pleasing solution is to reduce the *shrinklimit* option which specifies a lower limit for the shrinking factor applied by the *shrink* method. Figure 4.24 shows how to avoid forced hyphenation by compressing text down to 50%. The option list looks as follows:

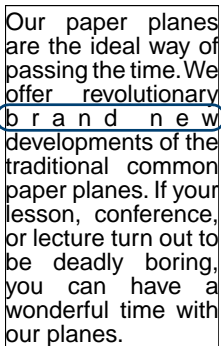
```
fontname=Helvetica fontsize=9 encoding=winansi alignment=justify shrinklimit=50%
```

**Option spreadlimit for justified text.** Expanding text, which is achieved by the *spread* method and controlled by the *spreadlimit* option, is another method for controlling line breaks. This unpleasing method should be rarely used, however. Figure 4.25 demonstrates a very large maximum character distance of 5 units. The option list looks as follows:

```
fontname=Helvetica fontsize=9 encoding=winansi alignment=justify spreadlimit=5
```

**Option nofitlimit for justified text.** The *nofitlimit* option controls how small a line can get when the *nofit* method is applied. Reducing the default value of 75% is preferable to forced hyphenation when the fitbox is very narrow. Figure 4.26 shows the generated text output with a minimum text width of 50%. The option list looks as follows:

```
fontname=Helvetica fontsize=9 encoding=winansi alignment=justify nofitlimit=50
```



Our paper planes  
are the ideal way of  
passing the time. We  
offer revolutionary  
brand new develop-  
ments of the traditional  
common paper planes.  
If your lesson,  
conference, or lecture  
turn out to be deadly  
boring, you can have  
a wonderful time with  
our planes.

Fig. 4.25  
Justified text with a maximum character spacing of 5 units

expand the line (spread method, spreadlimit option)

# 5 Importing and Placing Objects

PDFlib offers a variety of features for importing raster images and pages from existing PDF documents, and placing them on the page. This chapter covers the details of dealing with raster images and importing pages from existing PDF documents. It also presents samples which demonstrate how to place images and PDF pages on an output page.

## 5.1 Importing Raster Images

### 5.1.1 Basic Image Handling

Embedding raster images with PDFlib is easy to accomplish. First, the image file has to be opened with a PDFlib function which does a brief analysis of the image parameters. The *PDF\_load\_image()* function returns a handle which serves as an image descriptor. This handle can be used in a call to *PDF\_fit\_image()*, along with positioning and scaling parameters:

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

The last argument to the *PDF\_fit\_image()* function is an option list which supports a variety of options for positioning, scaling, and rotating the image. Details regarding these options are discussed in Section 5.3, »Placing Images and Imported PDF Pages«, page 136.

**Re-using image data.** PDFlib supports an important PDF optimization technique for using repeated raster images. Consider a layout with a constant logo or background on multiple pages. In this situation it is possible to include the actual image data only once in the PDF, and generate only a reference on each of the pages where the image is used. Simply load the image file once, and call *PDF\_fit\_image()* every time you want to place the logo or background on a particular page. You can place the image on multiple pages, or use different scaling factors for different occurrences of the same image (as long as the image hasn't been closed). Depending on the image's size and the number of occurrences, this technique can result in enormous space savings.

**Inline images.** As opposed to reusable images, which are written to the PDF output as image XObjects, inline images are written directly into the respective content stream (page, pattern, template, or glyph description). This results in some space savings, but should only be used for small amounts of image data (up to 4 KB) per a recommendation in the PDF reference. The primary use of inline images is for bitmap glyph descriptions in Type 3 fonts.

Inline images can be generated with the *PDF\_load\_image()* interface by supplying the *inline* option. Inline images cannot be reused, i.e., the corresponding handle must not be supplied to any call which accepts image handles. For this reason if the *inline* option has

been provided `PDF_load_image()` internally performs the equivalent of the following code:

```
PDF_fit_image(p, image, 0, 0, "");
PDF_close_image(p, image);
```

**Scaling and dpi calculations.** PDFlib never changes the number of pixels in an imported image. Scaling either blows up or shrinks image pixels, but doesn't do any downsampling (the number of pixels in an image will always remain the same). A scaling factor of 1 results in a pixel size of 1 unit in user coordinates. In other words, the image will be imported at 72 dpi if the user coordinate system hasn't been scaled (since there are 72 default units to an inch).

### 5.1.2 Supported Image File Formats

PDFlib deals with the image file formats described below. If possible, PDFlib passes the compressed image data unchanged to the PDF output since PDF internally supports most compression schemes used in common image file formats. This technique (called *pass-through mode* in the descriptions below) results in very fast image import, since decompressing the image data and subsequent recompression are not necessary. However, PDFlib cannot check the integrity of the compressed image data in this mode. Incomplete or corrupt image data may result in error or warning messages when using the PDF document in Acrobat (e.g., *Read less image data than expected*).

If an image file can't be imported successfully `PDF_load_image()` will return an error code by default. However, if you need to know more details about the failure set the *imagewarning* option in `PDF_load_image()` to *true* (see Section 8.6, »Image and Template Functions«, page 243). Alternatively, the *imagewarning* parameter can be set on a global basis:

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

This will cause `PDF_load_image()` to raise an exception with more details about the failure in the corresponding exception message.

**PNG images.** PDFlib supports all flavors of PNG images (Portable Network Graphics). PNG images are handled in pass-through mode in most cases. PNG images which make use of interlacing or contain an alpha channel (which will be lost anyway, see below) will have to be uncompressed, which takes significantly longer than pass-through mode. If a PNG image contains transparency information, the transparency is retained in the generated PDF (see Section 5.1.3, »Image Masks and Transparency«, page 128). However, alpha channels are not supported by PDFlib.

**JPEG images.** JPEG images are always handled in pass-through mode. PDFlib supports the following flavors of JPEG image compression:

- Baseline JPEG compression which accounts for the vast majority of JPEG images.
- Progressive JPEG compression.

JPEG images can be packaged in several different file formats. PDFlib supports all common JPEG file formats, and will read resolution information from the following flavors:

- JFIF, which is generated by a wide variety of imaging applications.

- JPEG files written by Adobe Photoshop and other Adobe applications. PDFlib applies a workaround which is necessary to correctly process Photoshop-generated CMYK JPEG files.

*Note PDFlib does not interpret resolution information from JPEG images in the SPIFF file format, nor color space information from JPEG images in the EXIF file format. JPEG images with multiple scans (this is a very rare flavor) are not supported due to restrictions in Acrobat.*

**GIF images.** GIF images are always handled in pass-through mode (PDFlib does not implement LZW decompression). PDFlib supports the following flavors of GIF images:

- Due to restrictions in the compression schemes supported by the PDF file format, the entry in the GIF file called »LZW minimum code size« must have a value of 8 bits. Unfortunately, there is no easy way to determine this value for a certain GIF file. An image which contains more than 128 distinct color values will always qualify (e.g., a full 8-bit color palette with 256 entries). Images with a smaller number of distinct colors may also work, but it is difficult to tell in advance because graphics programs may use 8 bits or less as LZW minimum code size in this case, and PDFlib may therefore reject the image. The following trick which works in Adobe Photoshop and similar image processing software is known to result in GIF images which are accepted by PDFlib: load the GIF image, and change the image color mode from »indexed« to »RGB«. Now change the image color mode back to »indexed«, and choose a color palette with more than 128 entries, for example the Mac or Windows system palette, or the Web palette.
- The image must not be interlaced.

For other GIF image flavors conversion to the PNG graphics format is recommended.

*Note In a particular test case PDFlib converted a GIF image to a PDF file which displays just fine, but results in a PostScript error when printed to a PostScript Level 2 or 3 printer. Since the problem does not occur with Ghostscript, we consider this a bug in the PostScript interpreter. You can work around the problem by selecting PostScript Level 1 output in Acrobat's print dialog.*

**TIFF images.** PDFlib will handle most TIFF images in pass-through mode. PDFlib supports the following flavors of TIFF images:

- compression schemes: uncompressed, CCITT (group 3, group 4, and RLE), ZIP (=Flate), LZW (with restrictions), and PackBits (=RunLength) are handled in pass-through mode; other compression schemes, such as JPEG, are handled by uncompressing.
- color: black and white, grayscale, RGB, CMYK, CIElab, and YCbCr images; any alpha channel or mask which may be present in the file will be ignored.
- TIFF files containing more than one image (see Section 5.1.5, »Multi-Page Image Files«, page 131)
- Color depth must be 1, 2, 4, 8, or 16 bits per color sample. In PDF 1.5 mode 16 bit color depth will be retained in most cases with pass-through mode, but reduced to 8 bit for certain image files (ZIP compression with little-endian/Intel byte order, or 16-bit palette images).

Multi-strip TIFF images are converted to multiple images in the PDF file which will visually exactly represent the original image, but can be individually selected with Acrobat's TouchUp object tool. Multi-strip TIFF images can be converted to single-strip images with the *tiffcp* command line tool which is part of the TIFFlib package.<sup>1</sup> The ImageMagick<sup>2</sup> tool always writes single-strip TIFF images.

PDFlib fully interprets the orientation tag which specifies the desired image orientation in some TIFF files. PDFlib can be instructed to ignore the orientation tag (as many applications do) by setting the *ignoreorientation* option to true.

Some TIFF features (e.g., spot color) and certain combinations of features (e.g., LZW compression plus alpha channel, mask, or tiling) are not supported.

**BMP images.** BMP images cannot be handled in pass-through mode. PDFlib supports the following flavors of BMP images:

- ▶ BMP versions 2 and 3;
- ▶ color depth 1, 4, and 8 bits per component, including  $3 \times 8 = 24$  bit TrueColor;
- ▶ black and white or RGB color (indexed and direct);
- ▶ uncompressed as well as 4-bit and 8-bit RLE compression;
- ▶ PDFlib will not mirror images if the pixels are stored in bottom-up order (this is a rarely used feature in BMP which is interpreted differently in applications).

**CCITT images.** Group 3 or Group 4 fax compressed image data are always handled in pass-through mode. Note that this format actually means raw CCITT-compressed image data, *not* TIFF files using CCITT compression. Raw CCITT compressed image files are usually not supported in end-user applications, but can only be generated with fax-related software. Since PDFlib is unable to analyze CCITT images, all relevant image parameters have to be passed to *PDF\_load\_image()* by the client.

**Raw data.** Uncompressed (raw) image data may be useful for some special applications. The nature of the image is deduced from the number of color components: 1 component implies a grayscale image, 3 components an RGB image, and 4 components a CMYK image.

### 5.1.3 Image Masks and Transparency

**Transparency in PDF.** PDF supports various transparency features, all of which are implemented in PDFlib:

- ▶ Masking by position: an image may carry the intrinsic information »print the foreground or the background«. This is realized by a 1-bit mask image, and is often used in catalog images.
- ▶ Masking by color value: pixels of a certain color are not painted, but the previously painted part of the page shines through instead (»ignore all blue pixels in the image«). In TV and video technology this is also known as bluescreening, and is most often used for combining the weather man and the map into one image.
- ▶ PDF 1.4 introduced alpha channels or soft masks. These can be used to create a smooth transition between foreground and background, or to create semi-transparent objects (»blend the image with the background«). Soft masks are represented by 1-component images with 1-8 bits per pixel.

PDFlib supports three kinds of transparency information in images: implicit transparency, explicit transparency, and image masks.

1. See [www.libtiff.org](http://www.libtiff.org)

2. See [www.imagemagick.org](http://www.imagemagick.org)



**Implicit transparency.** In the implicit case, the transparency information from an external image file is respected, provided the image file format supports transparency or an alpha channel (this is not the case for all image file formats). Transparency information is detected in the following image file formats:

- ▶ GIF image files may contain a single transparent color value which is respected by PDFlib.
- ▶ PNG image files may contain several flavors of transparency information, or a full alpha channel. PDFlib will retain single transparent color values; if multiple color values with an attached alpha value are given, only the first one with an alpha value below 50 percent is used. A full alpha channel is ignored.

*Note There is a bug in Acrobat 5 which prevents the use of transparent monochrome images. Instead of displaying the image Acrobat will issue an error message »There was an error processing a page. A drawing error occurred.« The bug does not exist in Acrobat 6. As a workaround you can remove transparency or save the image with 4 or more bits per pixel.*

**Explicit transparency.** The explicit case requires two steps, both of which involve image operations. First, an image must be prepared for later use as a transparency mask. This is accomplished by opening the image with the *mask* option. In PDF 1.3, which supports only 1-bit masks, using this option is required; in PDF 1.4 it is optional. The following kinds of images can be used for constructing a mask:

- ▶ PNG images
- ▶ TIFF images (only single-strip)
- ▶ raw image data

Pixel values of 0 in the mask will result in the corresponding area of the masked image being painted, while high pixel values result in the background shining through. If the pixel has more than 1 bit per pixel, intermediate values will blend the foreground image against the background, providing for a transparency effect. In the second step the mask is applied to another image which itself is acquired through one of the image functions:

```
mask = PDF_load_image(p, "png", maskfilename, 0, "mask");
if (mask == -1)
    return;
sprintf(optlist, "masked %d", mask);
image = PDF_load_image(p, type, filename, optlist)
if (image == -1)
    return;
PDF_fit_image(p, image, x, y, "");
```

Note the different use of the option list for *PDF\_load\_image()*: *mask* for defining a mask, and *masked* for applying a mask to another image.

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

*Note PDFlib converts multi-strip TIFF images to multiple PDF images, which would be masked individually. Since this is usually not intended, this kind of images will be rejected both as a mask as well as a masked target image. Also, it is important to not mix the implicit and explicit cases, i.e., don't use images with transparent color values as mask.*

**Image masks.** Image masks are images with a bit depth of 1 (bitmaps) in which 0-bits are treated as transparent: whatever contents already exist on the page will shine through the transparent parts of the image. 1-bit pixels are colorized with the current fill color. The following kinds of images can be used as image masks:

- ▶ PNG images
- ▶ TIFF images (single- or multi-strip)
- ▶ JPEG images (only as soft mask, see below)
- ▶ BMP; note that BMP images are oriented differently than other image types. For this reason BMP images must be reflected along the x axis before they can be used as a mask.
- ▶ raw image data

Image masks are simply opened with the *mask* option, and placed on the page after the desired fill color has been set:

```
mask = PDF_load_image(p, "tiff", maskfilename, 0, "mask");
PDF_setcolor(p, "fill", "rgb", (float) 1, (float) 0, (float) 0, (float) 0);
if (mask != -1) {
    PDF_fit_image(p, mask, x, y, "");
}
```

If you want to apply a color to an image without the 0-bit pixels being transparent you must use the *colorize* option (see Section 5.1.4, »Colorizing Images«, page 130).

**Soft masks.** Soft masks generalize the concept of image masks to masks with more than 1 bit. They have been introduced in PDF 1.4 and blend the image against some existing background. PDFlib accepts all kinds of single-channel (grayscale) images as soft mask. They can be used the same way as image masks, provided the PDF output compatibility is at least PDF 1.4.

**Ignoring transparency.** Sometimes it is desirable to ignore any transparency information which may be contained in an image file. For example, Acrobat's anti-aliasing feature (also known as »smoothing«) isn't used for 1-bit images which contain black and transparent as their only colors. For this reason imported images with fine detail (e.g., rasterized text) may look ugly when the transparency information is retained in the generated PDF. In order to deal with this situation, PDFlib's automatic transparency support can be disabled with the *ignoremask* option when opening the file:

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

## 5.1.4 Colorizing Images

Similarly to image masks, where a color is applied to the non-transparent parts of an image, PDFlib supports colorizing an image with a spot color. This feature works for black and white or grayscale images in the following formats:

- ▶ BMP
- ▶ PNG
- ▶ JPEG
- ▶ TIFF (single- or multi-strip)
- ▶ GIF

For images with an RGB palette, colorizing is only reasonable when the palette contains only gray values, and the palette index is identical to the gray value. PDFlib does not check this condition, however.

In order to colorize an image with a spot color you must supply the *colorize* option when opening the image, and supply the respective spot color handle which must have been retrieved with *PDF\_makespotcolor()*:

```
PDF_setcolor(p, "both", "cmyk", 1, .79, 0, 0);
spot = PDF_makespotcolor(p, "PANTONE Reflex Blue CV", 0);
sprintf(optlist, "colorize %d", spot);
image = PDF_load_image(p, "tiff", "image.tif", 0, optlist)
if (image != -1) {
    PDF_fit_image(p, image, x, y, "");
}
```

### 5.1.5 Multi-Page Image Files

PDFlib supports TIFF files which contain more than one image, also known as multi-page files. In order to use multi-page TIFFs, additional string and numerical parameters are used in the call to *PDF\_load\_image()*:

```
image = PDF_load_image(p, "tiff", filename, 0 "page 2");
```

The *page* option indicates that a multi-image file is to be used. The last parameter specifies the number of the image to use. The first image is numbered 1. This option may be increased until *PDF\_load\_image()* returns -1, signalling that no more images are available in the file.

A code fragment similar to the following can be used to convert all images in a multi-image TIFF file to a multi-page PDF file:

```
for (frame = 1; /* */ ; frame++) {
    sprintf(optlist, "page %d", frame);
    image = PDF_load_image(p, "tiff", filename, 0, optlist);
    if (image == -1)
        break;
    PDF_begin_page(p, width, height);
    PDF_fit_image(p, image, 0.0, 0.0, "");
    PDF_close_image(p, image);
    PDF_end_page(p);
}
```

### 5.1.6 OPI Support

When loading an image additional information according to OPI (Open Prepress Interface) version 1.3 or 2.0 can be supplied in the call to *PDF\_load\_image()*. PDFlib accepts all standard OPI 1.3 or 2.0 PostScript comments as options (not the corresponding PDF keywords!), and will pass through the supplied OPI information to the generated PDF output without any modification. The following example attaches OPI information to an image:

```
optlist13 =
"OPI-1.3 { ALDImageFilename bigfile.tif "
"ALDImageDimensions {400 561} "
"ALDImageCropRect {10 10 390 550} "
"ALDImagePosition {10 10  10 540  390 540  390 10} }";
```

```
image = PDF_load_image(p, "tiff", filename, 0, optlist13);
```

*Note* Some OPI servers, such as the one included in Helios EtherShare, do not properly implement OPI processing for PDF Image XObjects, which PDFlib generates by default. In such cases generation of Form XObjects can be forced by supplying the template option to `PDF_load_image()`.

## 5.2 Importing PDF Pages with PDI (PDF Import Library)

*Note All functions described in this section require PDFlib+PDI. The PDF import library (PDI) is not contained in PDFlib or PDFlib Lite. Although PDI is integrated in all precompiled editions of PDFlib, a license key for PDI (or PPS, which includes PDI) is required.*

### 5.2.1 PDI Features and Applications

When the optional PDI (PDF import) library is attached to PDFlib, pages from existing PDF documents can be imported. PDI contains a parser for the PDF file format, and prepares pages from existing PDF documents for easy use with PDFlib. Conceptually, imported PDF pages are treated similarly to imported raster images such as TIFF or PNG: you open a PDF document, choose a page to import, and place it on an output page, applying any of PDFlib's transformation functions for translating, scaling, rotating, or skewing the imported page. Imported pages can easily be combined with new content by using any of PDFlib's text or graphics functions after placing the imported PDF page on the output page (think of the imported page as the background for new content). Using PDFlib and PDI you can easily accomplish the following tasks:

- ▶ overlay two or more pages from multiple PDF documents (e.g., add stationary to existing documents in order to simulate preprinted paper stock);
- ▶ place PDF ads in existing documents;
- ▶ clip the visible area of a PDF page in order to get rid of unwanted elements (e.g., crop marks), or scale pages;
- ▶ impose multiple pages on a single sheet for printing;
- ▶ process multiple PDF/X-conforming documents to create a new PDF/X file;
- ▶ add some text (e.g., headers, footers, stamps, page numbers) or images (e.g., company logo) to existing PDF pages;
- ▶ copy all pages from an input document to the output document, and place barcodes on the pages.

In order to place a PDF background page and populate it with dynamic data (e.g., mail merge, personalized PDF documents on the Web, form filling) we recommend using PDI along with PDFlib blocks (see Chapter 6).

### 5.2.2 Using PDI Functions with PDFlib

**General considerations.** It is important to understand that PDI will only import the actual page contents, but not any hypertext features (such as sound, movies, embedded files, hypertext links, form fields, JavaScript, bookmarks, thumbnails, and notes) which may be present in the imported PDF document. These hypertext features can be generated with the corresponding PDFlib functions. PDFlib blocks will also be ignored when importing a page.

You can not re-use individual elements of imported pages with other PDFlib functions. For example, re-using fonts from imported documents for some other content is not possible. Instead, all required fonts must be configured in PDFlib. If multiple imported documents contain embedded font data for the same font, PDI will not remove any duplicate font data. On the other hand, if fonts are missing from some imported PDF, they will also be missing from the generated PDF output file. As an optimization you should keep the imported document open as long as possible in order to avoid the same fonts to be embedded multiple times in the output document.

PDI does not change the color of imported PDF documents in any way. For example, if a PDF contains ICC color profiles these will be retained in the output document.

PDFlib uses the template feature for placing imported PDF pages on the output page. Since some third-party PDF software does not correctly support the templates, restrictions in certain environments other than Acrobat may apply (see Section 3.2.4, »Templates«, page 61).

PDFlib-generated output which contains imported pages from other PDF documents can be processed with PDFlib+PDI again. However, due to restrictions in PostScript printing the nesting level should not exceed 10.

**Code fragments for importing PDF pages.** Dealing with pages from existing PDF documents is possible with a very simple code structure. The following code snippet opens a page from an existing document, and copies the page contents to a new page in the output PDF document (which must have been opened before):

```
int      doc, page, pageno = 1;
char     *filename = "input.pdf";

...

doc = PDF_open_pdi(p, filename, "", 0);
if (doc == -1) {
    printf("Couldn't open PDF input file '%s'\n", filename);
    exit(1);
}
page = PDF_open_pdi_page(p, doc, pageno, "");
if (page == -1) {
    printf("Couldn't open page %d of PDF file '%s'\n", pageno, filename);
    exit(2);
}

/* dummy page size, will be modified by the adjustpage option */
PDF_begin_page(p, 20, 20);
PDF_fit_pdi_page(p, page, 0, 0, "adjustpage");
PDF_close_pdi_page(p, page);
...add more content to the page using PDFlib functions...
PDF_end_page(p);
```

The last argument to *PDF\_fit\_pdi\_page()* is an option list which supports a variety of options for positioning, scaling, and rotating the imported page. Details regarding these options are discussed in Section 5.3, »Placing Images and Imported PDF Pages«, page 136.

**Dimensions of imported PDF pages.** Imported PDF pages are regarded similarly to imported raster images, and can be placed on the output page using *PDF\_fit\_pdi\_page()*. By default, PDI will import the page exactly as it is displayed in Acrobat, in particular:

- ▶ cropping will be retained (in technical terms: if a CropBox is present, PDI favors the CropBox over the MediaBox; see Section 3.2.2, »Page Sizes and Coordinate Limits«, page 59);
- ▶ rotation which has been applied to the page will be retained.

Alternatively, you can use the *pdiusebox* option to explicitly instruct PDI to use any of the MediaBox, CropBox, BleedBox, TrimBox or ArtBox entries of a page (if present) for determining the size of the imported page (see Table 8.43 for details).

Many important properties, such as width and height of an imported PDF page, all of the Box entries, and the number of pages in a document, can be queried via PDFlib's parameter mechanism. The relevant parameters are listed in Table 8.42 and Table 8.43. These properties can be useful in making decisions about the placement of imported PDF pages on the output page.

**Imported PDF pages with layers.** Acrobat 6 (PDF 1.5) introduced the layer functionality (technically known as *optional content*). PDI will ignore any layer information which may be present in a file. All layers in the imported page, including invisible layers, will be visible in the generated output.

**Imported PDF with OPI information.** OPI information present in the input PDF will be retained in the output unmodified.

### 5.2.3 Acceptable PDF Documents

Generally, PDI will happily process all kinds of PDF documents which can be opened with Acrobat, regardless of PDF version number or features used within the file. In order to import pages from encrypted documents (i.e., files with permission settings or password) the corresponding master password must be supplied.

However, in rare cases a PDF document or a particular page of a document may be rejected by PDI.

If a PDF document or page can't be imported successfully `PDF_open_pdi()` and `PDF_open_pdi_page()` will return an error code by default. However, if you need to know more details about the failure you can query the reason with `PDF_get_errmsg()`. Alternatively, you can set the `pdiwarning` option or parameter to `true`, which will result in an exception if the document cannot be opened:

```
PDF_set_parameter(p, "pdiwarning", "true");          /* enable PDI warnings */
```

This will cause `PDF_open_pdi()` and `PDF_open_pdi_page()` to raise an exception with more details about the failure in the corresponding exception message. The following kinds of PDF documents can not be imported with PDI:

- ▶ PDF documents which use a higher PDF version number than the PDF output document that is currently being generated. The reason is that PDFlib can no longer make sure that the output will actually conform to the requested PDF version after a PDF with a higher version number has been imported. Solution: set the version of the output PDF to the required level using the `compatibility` option in `PDF_begin_document()`.
- ▶ PDF documents with PDF/X conformance level which is not compatible to the PDF/X level of the current output document.
- ▶ PDF documents with a damaged cross-reference table. You can identify such files by Acrobat's warning message *File is damaged but is being repaired*. Solution: open and re-save the file with Acrobat. In addition, the following kinds of PDF documents will be rejected by default; however, they can be opened for querying information (as opposed to importing pages) by setting the `infomode` option to `true`:
- ▶ Encrypted PDF documents without the corresponding password.
- ▶ Tagged PDF when the `tagged` option in `PDF_begin_document()` is `true`.

## 5.3 Placing Images and Imported PDF Pages

The `PDF_fit_image()` function for placing raster image and templates, as well as `PDF_fit_pdi_page()` for placing imported PDF pages offer a wealth of options for controlling the placement on the page. This section demonstrates the most important options by looking at some common application tasks. A complete list and descriptions of all options can be found in Table 8.38.

Embedding raster images is easy to accomplish with PDFlib. The image file must first be loaded with `PDF_load_image()`. This function returns an image handle which can be used along with positioning and scaling options in `PDF_fit_image()`.

Embedding imported PDF pages works along the same line. The PDF page must be opened with `PDF_open_pdi_page()` to retrieve a page handle for use in `PDF_fit_pdi_page()`. The same positioning and scaling options can be used as for raster images.

All samples in this section work the same for raster images, templates, and imported PDF pages. Although code samples are only presented for raster images we talk about placing objects in general. Note that before calling any of the *fit* functions a call to `PDF_load_image()` or `PDF_open_pdi()` and `PDF_open_pdi_page()` must be issued. For the sake of simplicity these calls are not reproduced here.

### 5.3.1 Scaling, Orientation, and Rotation

**Simple Placing.** Let's start with the simplest case (see Figure 5.1): an object will be placed at a certain position at its original size:

```
PDF_fit_image(p, image, 80, 100, "");
```

In this code fragment the object will be placed with its lower left corner at the point (80, 100) in the user coordinate system. This point is called the reference point. The option list (the last function parameter) is empty. This means the object will be placed in its original size at the provided reference point.

**Placing with Scaling.** The following variation is also very easy to use (see Figure 5.2) We place the object as in the previous example, but will modify the object's scaling:

```
PDF_fit_image(p, image, 80, 100, "scale 0.5");
```

Fig. 5.1  
Simple placing



Fig. 5.2  
Placing with scaling





This code fragment places the object with its lower left corner at the point (80, 100) in the user coordinate system. In addition, the object will be scaled in *x* and *y* direction by a scaling factor of 0.5, which makes it appear at 50 percent of its original size.

**Placing with orientation.** In the next code fragment we will orientate the object in direction west (see Figure 5.3):

```
PDF_fit_image(p, image, 80, 100, "scale 0.5 orientate west");
```

This code fragment orientates the object towards western direction (90 degrees counterclockwise), and then translates the object's lower left corner (after applying the *orientate* option) to the reference point (*x*, *y*). The object will be rotated in itself.

**Placing with rotation.** Rotating an object (see Figure 5.4) works similarly to orientation. However, it not only affects the placed object but the whole coordinate system. Before placing the object the coordinate system will be rotated at the reference point (*x*, *y*) by 90 degrees counterclockwise. The rotated object's lower right corner (which is the unrotated object's lower left corner) will end up at the reference point. The function call to achieve this looks as follows:

```
PDF_fit_image(p, image, 80, 100, "scale 0.5 rotate 90");
```

Since there is no translation in this case the object will be partially moved outside the page.

**Comparing orientation and rotation.** Orientation and rotation are quite similar concepts, but are different nevertheless, and you should be aware of these differences. Figure 5.5 and Figure 5.6 demonstrate the principal difference between the *orientate* and *rotate* options:

- ▶ The *orientate* option rotates the object at the reference point (*x*, *y*) and subsequently translates it. This option supports the direction keywords *north*, *east*, *west*, and *south*.
- ▶ The *rotate* option rotates the object at the reference point (*x*, *y*) without any translation. This option supports arbitrary rotation angles. These have to be specified numerically in degrees (a full circle has 360 degrees).

Fig. 5.3  
Placing with orientation

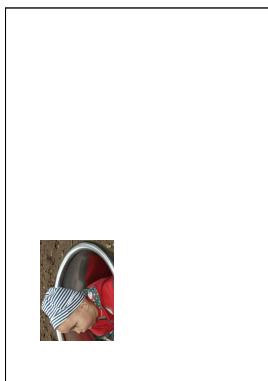
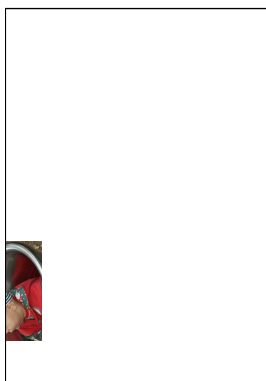


Fig. 5.4  
Placing with rotation



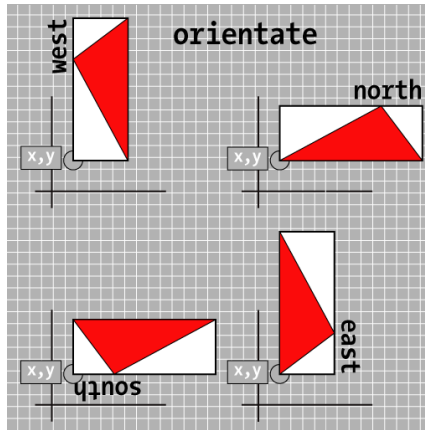


Fig. 5.5  
The orientate option

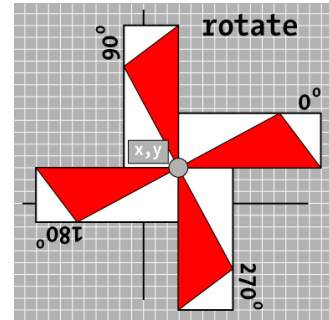


Fig. 5.6  
The rotate option

### 5.3.2 Adjusting the Page Size

In the next example (see Figure 5.7) we will automatically adjust the page size to the object's size. This can be useful, for example, for archiving images in the PDF format. The reference point  $(x, y)$  can be used to specify whether the page will be exactly the object's size, or somewhat larger or smaller. When enlarging the page size (see Figure 5.7) some border will be kept around the image; when the page size is smaller than the image some parts of the image will be clipped. Let's start with exactly matching the page size to the object's size:

```
PDF_fit_image(p, image, 0, 0, "adjustpage");
```

The next code fragment makes the page size larger by 40 units in  $x$  and  $y$  direction than the object, resulting in some border around the object:

```
PDF_fit_image(p, image, 40, 40, "adjustpage");
```

The next code fragment makes the page size smaller by 40 units in  $x$  and  $y$  direction than the object. The object will be clipped at the page borders, and some area within the object (with a width of 40 units) will be invisible:

```
PDF_fit_image(p, image, -40, -40, "adjustpage");
```



Fig. 5.7  
Adjusting the page size. Left to right: exact, enlarge, shrink

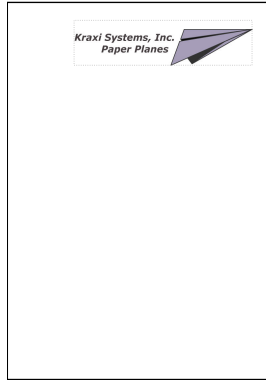


Fig. 5.8  
Fitting an object  
to the box

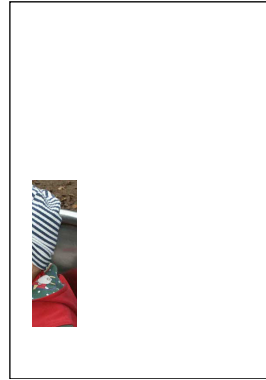


Fig. 5.9  
Clipping an object when  
fitting it to the box

In addition to placing by means of x and y coordinates (which specify the object's distance from the page edges, or the coordinate axes in the general case) you can also specify a target box. This is a rectangular area in which the object will be placed subject to various formatting rules. These can be controlled with the *boxsize*, *fitmethod* and *position* options.

**Fitting an object to a box.** First, let's place a company logo in the upper right area of the page (see Figure 5.8). The size of the target rectangle where the logo is to appear is fixed. However, we don't know how to scale the logo so that it fits into the box while avoiding any distortion (the ratio of width and height must not be changed). The following statement does the job:

```
PDF_fit_image(p, image, 350, 750, "boxsize {200 100} position 0 fitmethod meet");
```

This code fragment places the lower left corner of a box which is 200 units wide and 100 units high (*boxsize {200 100}*) at the point (350, 750). The object's lower left corner will be placed at the box's lower left corner (*position 0*). The object will be scaled without any distortion to make its height and/or width exactly fit into the box (*fitmethod meet*).

This concept offers a broad range of variations. For example, the *position* option can be used to specify which point within the object is to be used as the reference point (specified as a percentage of width and height). The *position* option will also specify the reference point within the target box. If both width and height position percentages are identical it is sufficient to specify a single value. For example, *position 50* can be used to select the object's and box's midpoint as reference point for placing the object.

**Clipping an object when fitting it to the box.** Using another flavor of the *fitmethod* option we can clip the object such that it exactly fits into the target box (see Figure 5.9). In this case the object won't be scaled:

```
PDF_fit_image(p, image, 50, 80, "boxsize {100 400} position 50 fitmethod clip");
```

This code fragment places a box of width 100 and height 400 (*boxsize {100 400}*) at the coordinates (50, 80). The object will be placed in its original size in the middle of the box (*position 50*), and will be cropped if it exceeds the box (*fitmethod clip*).

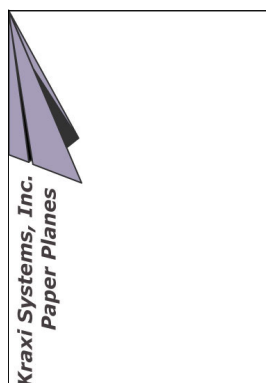


Fig. 5.10  
Fitting a logo to the page

**Adjusting an object to the page.** Adjusting an object to a given page size can easily be accomplished by choosing the page as target box for placing the object. The following statement uses an A4-sized page with dimensions 595 x 842:

```
PDF_fit_image(p, image, 0, 0, "boxsize {595 842} position 0 fitmethod slice");
```

In this code fragment a box is placed at the lower left corner of the page. The size of the box equals the size of an A4 page. The object is placed in the lower left corner of the box and scaled proportionally until it fully covers the box and therefore the page. If the object exceeds the box it will be cropped. Note that *fitmethod slice* results in the object being scaled (as opposed to *fitmethod clip* which doesn't scale the object). Of course the *position* and *fitmethod* options could also be varied in this example.

**Fitting a logo to the page.** How can we achieve the rotated company logo in Figure 5.10? It is rotated by 90 degrees counterclockwise, starts in the lower left corner, and covers the full height of the page:

```
PDF_fit_image(p, image, 0, 0, "boxsize {595 842} orientate west fitmethod meet");
```

The reference point is (0, 0) and orientation is specified as *orientate west*. In order to make the logo cover the full page height we choose the box height to be equal to the page height (842), and choose a large enough value for the box's width (595). The logo's proportions should not be changed, therefore we choose *fitmethod meet*.

# 6 Variable Data and Blocks

PDFlib supports a template-driven PDF workflow for variable data processing. Using the concept of blocks, imported pages can be populated with variable amounts of text, images, or PDF graphics which can be pulled from an external source. This can be used to easily implement applications which require customized PDF documents, for example:

- ▶ mail merge
- ▶ flexible direct mailings
- ▶ transactional and statement processing
- ▶ business card personalization

*Note* Block processing requires the PDFlib Personalization Server (PPS). Although PPS is contained in all commercial PDFlib packages, you must purchase a license key for PPS; a PDFlib or PDFlib+PDI license key is not sufficient. The PDFlib Block plugin for Adobe Acrobat is required for creating blocks in PDF templates.

## 6.1 Installing the PDFlib Block Plugin

The Block plugin and its sibling, the PDF form field conversion plugin, work only with the full version of Acrobat 5, Acrobat 6 Standard and Acrobat 6 Professional. The plugins don't work with Acrobat 6 Elements or any version of Acrobat Reader/Adobe Reader.

*Note* If the PDFlib Block plugin doesn't seem to work make sure that in Edit, Preferences, [General...], Startup (Acrobat 6) or Options (Acrobat 5) the »Use only certified plug-ins« box is unchecked.

**Installing the PDFlib Block plugins for Acrobat on Windows.** To install the PDFlib Block plugin and the PDF form field conversion plugin in Acrobat 5 or 6 the plugin files must be copied to a subdirectory in the Acrobat plugin folder. This is done automatically by the plugin installer, but can also be done manually. The plugin files are called *Block.api* and *AcroFormConversion.api*, and a typical location of the plugin folder looks as follows:

C:\Program Files\Adobe\Acrobat 6.0\Acrobat\plug\_ins\PDFlib

**Installing the PDFlib Block plugins for Acrobat on the Mac.** With Acrobat 6 the plugin folder will not be visible in the finder. Instead of dragging the plugin files to the plugin folder use the following steps (make sure that Acrobat is not running):

- ▶ Extract the plugin files by double-clicking the disk image.
- ▶ Locate the Acrobat application icon in the finder. It is usually located in a folder which has a name similar to the following:

/Applications/Adobe Acrobat 6.0 Professional

- ▶ Single-click on the icon and select *File, Get Info*.
- ▶ In the window that pops up click the triangle next to *Plug-ins*.
- ▶ Click *Add...* and select the *PDFlib* folder from the folder which has been created in the first step. Note that the *PDFlib* folder will not immediately show up in the list of plug-ins, but only when you open up the info window next time.

To install the plugins for Acrobat 5, start by double-clicking the disk image. Drag the PDFlib folder to the Acrobat 5 plugin folder. A typical plugin folder name is as follows:

/Applications/Adobe Acrobat 5.0/Plug-Ins

## 6.2 Overview of the PDFlib Block Concept

### 6.2.1 Complete Separation of Document Design and Program Code

PDFlib data blocks make it easy to place variable text, images, or graphics on imported pages. In contrast to simple PDF pages, pages containing data blocks intrinsically carry information about the required processing which will be performed later on the server side. The PDFlib block concept completely separates the following tasks:

- ▶ A designer creates the page layout, and specifies the location of variable text and image elements along with relevant properties such as font size, color, or image scaling. After creating the layout as a PDF document, the designer uses the PDFlib Block plugin for Acrobat to specify variable data blocks and their associated properties.
- ▶ A programmer writes code to connect the information contained in PDFlib blocks on imported PDF pages with dynamic information, e.g., database fields. The programmer doesn't need to know any details about a block (whether it contains a name or a ZIP code, the exact location on the page, its formatting, etc.) and is therefore independent from any layout changes. PDFlib will take care of all block-related details based on the block properties found in the file.

In other words, the code written by the programmer is »data-blind« – it is generic and does not depend on the particulars of any block. For example, the designer may decide to use the first name of the addressee in a mailing instead of the last name. The generic block handling code doesn't need to be changed, and will generate correct output once the designer changed the block properties with the Acrobat plugin to use the first name instead of the last name.

**Example: adding variable text to a template.** Adding dynamic text to a PDF template is a very common task. The following code fragment will open a page in an input PDF document (the template), place it on the output page, and fill some variable text into a text block called *firstname*:

```
doc = PDF_open_pdi(p, filename, "", 0);
if (doc == -1) {
    printf("Couldn't open PDF template '%s'\n", filename);
    return (1);
}
page = PDF_open_pdi_page(p, doc, pageno, "");
if (page == -1) {
    printf("Couldn't open page %d of PDF template '%s'\n", pageno, filename);
    return (2);
}

PDF_begin_page_ext(p, width, height, "");
PDF_fit_pdi_page(p, page, 0.0, 0.0, "");
PDF_fill_textblock(p, page, "firstname", "Serge", 0, "encoding winansi");
PDF_close_pdi_page(p, page);
PDF_end_page_ext(p, "");
```

## 6.2.2 Block Properties

The behavior of blocks can be controlled with block properties. The properties are assigned to a block with the PDFlib Block plugin for Acrobat.

**Standard block properties.** PDFlib blocks are defined as rectangles on the page which are assigned a name, a type, and an open set of properties which will later be processed on the server side. The name is an arbitrary string which identifies the block, such as *firstname*, *lastname*, or *zipcode*. PDFlib supports the following kinds of blocks:

- ▶ Type *Text* means that the block will hold one or more lines of textual data. Multi-line text will be formatted with the Textflow feature. Note that text blocks cannot be linked such that text will flow from one block to another.
- ▶ Type *Image* means that the block will hold a raster image. This is similar to importing a TIFF or JPEG file in a DTP application.
- ▶ Type *PDF* means that the block will hold arbitrary PDF graphics imported from a page in another PDF document. This is similar to importing an EPS graphic in a DTP application.

A block may carry a number of standard properties depending on its type. For example, a text block may specify the font and size of the text, an image or PDF block may specify the scaling factor or rotation. For each type of block the PDFlib API offers a dedicated function for processing the block. These functions search an imported PDF page for a block by its name, analyze its properties, and place some client-supplied data (text, raster image, or PDF page) on the new page according to the corresponding block properties.

**Custom block properties.** Standard block properties make it possible to quickly implement variable data processing applications, but these are limited to the set of properties which are internally known to PDFlib and can automatically be processed. In order to provide more flexibility, the designer may also assign custom properties to a block. These can be used to extend the block concept in order to match the requirements of the most demanding variable data processing applications.

There are no rules for custom properties since PDFlib will not process custom properties in any way, except making them available to the client. The client code can examine the custom properties and act in whatever way it deems appropriate. Based on some custom property of a block the code may make layout-related or data-gathering decisions. For example, a custom property for a scientific application could specify the number of digits for numerical output, or a database field name may be defined as a custom block property for retrieving the data corresponding to this block.

**Overriding block properties.** In certain situations the programmer would like to use only some of the properties provided in a block definition, but override other properties with custom values. This can be useful in various situations:

- ▶ The scaling factor for an image or PDF page will be calculated instead of taken from the block definition.
- ▶ Change the block coordinates programmatically, for example when generating an invoice with a variable number of data items.
- ▶ Individual spot color names could be supplied in order to match the requirements of individual customers in a print shop application.

Property overrides can be achieved by supplying property names and the corresponding values in the option list of all *PDF\_fill\_\*block()* functions as follows:

```
PDF_fill_textblock(p, page, "firstname", "Serge", 0, "fontsize 12");
```

This will override the block's internal *fontsize* property with the supplied value 12. Almost all names of general properties can be used as options, as well as those specific to a particular block type. For example, the *underline* option is only allowed for *PDF\_fill\_textblock()*, while the *scale* option is allowed for both *PDF\_fill\_imageblock()* and *PDF\_fill\_pdfblock()* since *scale* is a valid property for both image and PDF blocks.

Property overrides apply only to the respective function calls; they will not be stored in the block definition.

**Coordinate systems.** The coordinates describing a block reference the PDF default coordinate system. When the page containing the block is placed on the output page, several positioning and scaling options may be supplied to *PDF\_fit\_pdi\_page()*. These parameters are taken into account when the block is being processed. This makes it possible to place a template page on the output page multiply, every time filling its blocks with data. For example, a business card template may be placed four times on an imposition sheet. The block functions will take care of the coordinate system transformations, and correctly place the text for all blocks in all invocations of the page. The only requirement is that the client must place the page and then process all blocks on the placed page. Then the page can be placed again at a different location on the output page, followed by more block processing operations referring to the new position, and so on.

*Note* The Block plugin will display the block coordinates differently from what is stored in the PDF file. The plugin uses Acrobat's convention which has the coordinate origin in the upper left corner of the page, while the internal coordinates (those stored in the block) use PDF's convention of having the origin at the lower left corner of the page.

## 6.2.3 Why not use PDF Form Fields?

Experienced Acrobat users may ask why we implemented a new block concept for PDFlib, instead of relying on the established form field scheme available in PDF. The primary distinction is that PDF form fields are optimized for interactive filling, while PDFlib blocks are targeted at automated filling. Applications which need both interactive and automated filling can easily achieve this by using a feature which automatically converts form fields to blocks (see Section 6.3.4, »Converting PDF Form Fields to PDFlib Blocks«, page 150).

Although there are many parallels between both concepts, PDFlib blocks offer several advantages over PDF form fields as detailed in Table 6.1.




Table 6.1 Comparison of PDF form fields and PDFlib blocks

Feature	PDF form fields	PDFlib blocks
design objective	for interactive use	for automated filling
typographic features (beyond choice of font and font size)	–	Kerning, word and character spacing, underline/overline/strikeout
font control	font embedding	font embedding and subsetting, encoding
text formatting controls	none	left-, center-, right-aligned, justified; font changes; various formatting algorithms and controls
merged result is integral part of PDF page description	no	yes
users can edit merged field contents	yes	no
extensible set of properties	no	yes (custom block properties)
color support	RGB	grayscale, RGB, CMYK, spot color, Lab
PDF/X compatible	no	yes (both template with blocks and merged results)
graphics and text properties can be overridden upon filling	no	yes

## 6.3 Creating PDFlib Blocks

### 6.3.1 Creating Blocks interactively with the PDFlib Block Plugin

**Activating the PDFlib Block tool.** The PDFlib Block plugin for creating PDFlib blocks is similar to the form tool in Acrobat. All blocks on the page will be visible when the block tool is active. When another Acrobat tool is selected the blocks will be hidden, but they are still present. You can activate the block tool in several ways:

- ▶ by clicking the block icon  in Acrobat's *Advanced Editing* toolbar (in Acrobat 5: *Editing* toolbar);
- ▶ via the menu item *PDFlib Blocks, PDFlib Block Tool*;
- ▶ by using the keyboard shortcut *P* (in Acrobat 6 make sure to enable *Edit, Preferences, [General...], General, Use single key accelerators to access tools*, which is disabled by default).

**Creating and modifying blocks.** Once you activated the block tool you can simply drag the cross-hair pointer to create a block at the desired position on the page and the desired size. Blocks will always be rectangular with edges parallel to the page edges. When you create a new block the block properties dialog appears where you can edit various properties of the block (see Section 6.3.2, »Editing Block Properties«, page 148). The block tool will automatically create a block name which can be changed in the properties dialog. Block names must be unique within a page. You can change the block type in the *General* tab to one of Text, Image, or PDF. The *General* and *Custom* tabs will always be available, while only one of the Text, Image, and PDF tabs will be active at a time depending on the chosen block type.

*Note* After you added blocks or made changes to existing blocks in a PDF, use Acrobat's »Save as...« Command (as opposed to »Save«) to achieve smaller file sizes.

*Note* When using the Acrobat plugin Enfocus PitStop to edit documents which contain PDFlib blocks you may see the message »This document contains PieceInfo from PDFlib. Press OK to continue editing or Cancel to abort.« This message can be ignored; it is safe to click OK in this situation.

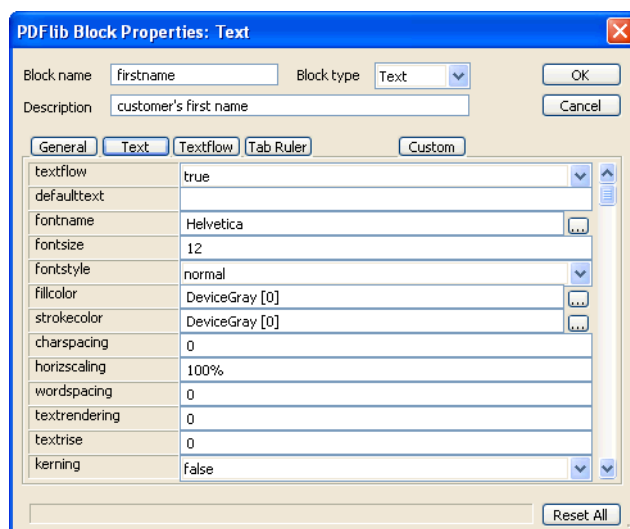
**Selecting blocks.** Several block operations, such as copying or moving, work with selected blocks. You can select one or more blocks with the block tool as follows:

- ▶ To select a single block simply click on it with the mouse.
- ▶ Hold down the Shift key while clicking on another block to extend the selection.
- ▶ Press Ctrl-A (on Windows) or Cmd-A (on the Mac) or *Edit, Select All* to select all blocks on a page.

**The context menu.** When one or more blocks are selected you can open the context menu to quickly access block-related functions (which are also available in the PDFlib Blocks menu). To open the context menu, click on the selected block(s) with the right mouse button on Windows, or Ctrl-click the block(s) on the Mac.

For example, to delete a block, select it with the block tool and press the *Delete* key, or use *Edit, Delete* in the context menu.

**Fine-tuning block size and position.** Using the block tool you can move one or more selected blocks to a different position. Hold down the Shift key while dragging a block to restrain the positioning to horizontal and vertical movements. This may be useful for



*Fig. 6.1*  
Editing block properties: the *Textflow*  
panel is only visible if *textflow=true*;  
the *Tab Ruler* panel is only visible if  
*hortabmethod=ruler*

exactly aligning blocks. When the pointer is located near a block corner, the pointer will change to an arrow and you can resize the block. To adjust the position or size of multiple blocks, select two or more blocks and use the *Align*, *Center*, *Distribute*, or *Size* commands from the *PDFlib Blocks* menu or the context menu. The position of one or more blocks can also be changed by using the arrow keys.

**Creating blocks by selecting an image or graphic.** As an alternative to manually dragging block rectangles you can use existing page contents to define the block size. First, make sure that the menu item *PDFlib Blocks, Click Object to define Block* is enabled. Now you can use the block tool to click on an image on the page in order to create a block with the size of the image. You can also click on other graphical objects, and the block tool will try to select the surrounding graphic (e.g., a logo). The *Click Object* feature is intended as an aid for defining blocks. If you want to reposition or resize the block you can do so afterwards without any restriction. The block will not be locked to the image or graphics object which was used as a positioning and sizing aid.

The *Click Object* feature will try to recognize which vector graphics and images form a logical element on the page. When some page content is clicked, its bounding box (the surrounding rectangle) will be selected unless the object is white or very large. In the next step other objects which are partially contained in the detected rectangle will be added to the selected area, and so on. The final area will be used as the basis for the generated block rectangle. The end result is that the *Click Object* feature will try to select complete graphics, and not only individual lines.

The *Click Object* feature isn't perfect: it will not always select what you want, depending on the nature of the page content. Keep in mind that this feature is only intended as a positioning aid for quickly creating block rectangles.

**Automatically detecting font properties.** The *PDFlib Block* plugin can analyze the underlying font which is present on the location where a block is positioned, and can automatically fill in the corresponding properties of the block:

fontname, fontsize, fillcolor, charspacing, horizscaling, wordspacing, textrendering, textrise

Since automatic detection of font properties can result in undesired behavior when the background shall be ignored it can be activated or deactivated using *PDFlib Blocks, Detect underlying font and color*. By default this feature is turned off.

**Locking blocks.** Blocks can be locked to protect them against accidentally moving, re-sizing, or deleting them. With the block tool active, select the block and choose *Lock* from its context menu. While a block is locked you cannot move, resize, or delete it, nor display its properties dialog.

**Using Blocks with PDF/X.** Unlike PDF form fields, PDFlib blocks are PDF/X-compatible. Both the input document containing blocks, as well as the generated output PDF can be made PDF/X conforming. However, in preparing block files for a PDF/X workflow you may run into the following problem:

- ▶ PDF/X-1:2001, PDF/X-1a:2001, and PDF/X-3:2002 are based on PDF 1.3, and do not support Acrobat 5 files;
- ▶ The PDFlib Block plugin requires Acrobat 5 or above.

How to work around this depends on your Acrobat version:

- ▶ Acrobat 6: You can save the file as PDF 1.3 directly from Acrobat using *File, Reduce File Size...*, and choosing *Acrobat 4.0 and later*.
- ▶ Acrobat 5: For saving the generated PDF with blocks in the PDF/X-conforming PDF version 1.3 use an additional plugin by callas software called *pdfSaveAs1.3*. Fully functional demo versions are available on the callas web site<sup>1</sup>.

### 6.3.2 Editing Block Properties

When you create a new block, double-click an existing one, or choose *Properties* from the context menu, the properties dialog will appear where you can edit all settings related to the selected block (see Figure 6.1). As detailed in Section 6.4, »Standard Properties for Automated Processing«, page 153, there are several types of properties:

- ▶ Name, type, description, and the properties in the *General* tab apply to all blocks.
- ▶ Properties in the *Text*, *Image*, and *PDF* tabs apply only to the respective block type. Only the tab corresponding to the selected block's type will be active, while the other tabs are inactive.
- ▶ If a block of type *Text* has the *textflow* property set to *true*, another tab called *Textflow* will appear with Textflow-related settings.
- ▶ If a block of type *Text* has the *textflow* property set to *true*, and the *hortabmethod* property in the *Textflow* tab is set to *ruler*, still another tab called *Tab Ruler* will appear where you can edit tabulator settings.
- ▶ Properties in the *Custom* tab can be defined by the user, and apply to any block type.

To change a property's value enter the desired number or string in the property's input area (e.g. *linewidth*), choose a value from the available drop-down lists (e.g. *fitmethod*), or select a color, font, or file by clicking the »...« button at the right-hand side of the dialog (e.g. *backgroundcolor*). For the *fontname* property you can either choose from the list of fonts installed on the system (after clicking the »...« button symbol to open the font se-

<sup>1</sup>. See [www.callassoftware.com](http://www.callassoftware.com)

lection dialog), or type a custom font name. Regardless of the method for entering a font name, the font must be available on the system where the blocks will be filled.

When you are done editing properties, click OK to close the properties dialog. The properties just defined will be stored in the PDF file as part of the block definition.

**Stacked blocks.** Overlapping blocks can be difficult to select since clicking an area with the mouse will always select the topmost block. In such a situation the *Choose Block* entry in the context menu can be used to select one of the blocks by name. As soon as a block has been selected the next action (e.g. double-click) within its area will not affect other blocks, but only the selected one. This way block properties can easily be edited even for blocks which are partially or completely covered by other blocks.

**Using and restoring default properties.** In order to save some amount of typing and clicking, the block tool will remember the property values which have been entered into the previous block's properties dialog. These values will be reused when you create a new block. Of course you can override these values with different ones at any time.

Pressing the *Reset All* button in the properties dialog will reset most block properties to their respective defaults. However, the following items will remain unmodified:

- ▶ the *Name*, *Type*, *Rect*, and *Description* properties
- ▶ all custom properties.

**Shared properties.** By holding the Shift key and using the block tool to select multiple blocks you can select an arbitrary number of blocks on a page. Double-clicking one of the selected blocks or pressing the *Enter* key will display the properties dialog which now applies to all selected blocks. However, since not all properties can be shared among multiple blocks, only a subset of all properties will be available for editing. Section 6.4, »Standard Properties for Automated Processing«, page 153, details which properties can be shared among multiple blocks. Custom properties cannot be shared.

### 6.3.3 Copying Blocks between Pages and Documents

The block plugin offers several methods for moving and copying blocks within the current page, the current document, or other documents:

- ▶ move or copy blocks by dragging them with the mouse, or pasting blocks to another page or open document
- ▶ duplicate blocks on one or more pages of the same document
- ▶ export blocks to a new file (with empty pages) or to an existing document (apply the blocks to existing pages)
- ▶ import blocks from other documents

In order to update the page contents while maintaining block definitions you can replace the underlying page(s) while keeping the blocks. Use *Document, Pages, Replace* (Acrobat 6) or *Document, Replace Pages...* (Acrobat 5).

**Moving and copying blocks.** You can relocate blocks or create copies of blocks by selecting one or more blocks and dragging them to a new location while pressing the Ctrl key (on Windows) or Alt key (on the Mac). The mouse cursor will change while the key is pressed. A copied block will have the same properties as the original block, with the exception of its name and position which will automatically be changed.

You can also use copy/paste to copy blocks to another location on the same page, to another page in the same document, or to another document which is currently open in Acrobat:

- ▶ Activate the block tool and select the blocks you want to copy.
- ▶ Use Ctrl-C (on Windows) or Cmd-C (on the Mac) or *Edit, Copy* to copy the selected blocks to the clipboard.
- ▶ Use Ctrl-V (on Windows) or Cmd-V (on the Mac) or *Edit, Paste* to paste the blocks which are currently in the clipboard.

**Duplicating blocks on other pages.** You can create duplicates of one or more blocks on an arbitrary number of pages in the current document simultaneously:

- ▶ Activate the block tool and select the blocks you want to duplicate.
- ▶ Choose *Import and Export, Duplicate...* from the *PDFlib Blocks* menu or the context menu.
- ▶ Choose which blocks to duplicate (selected blocks or all on the page) and the range of target pages where you want duplicates of the blocks.

**Exporting and importing blocks.** Using the export/import feature for blocks it is possible to share the block definitions on a single page or all blocks in a document among multiple PDF files. This is useful for updating the page contents while maintaining existing block definitions. To export block definitions to a separate file proceed as follows:

- ▶ Activate the block tool and Select the blocks you want to export.
- ▶ Choose *Import and Export, Export...* from the *PDFlib Blocks* menu or the context menu. Enter the page range and a file name for the file containing the block definitions.

You can import block definitions via *PDFlib Blocks, Import and Export, Import...*. Upon importing blocks you can choose whether to apply the imported blocks to all pages in the document, or only to a page range. If more than one page is selected the block definitions will be copied unmodified to the pages. If there are more pages in the target range than in the imported block definition file you can use the *Repeate Template* checkbox. If it is enabled the sequence of blocks in the imported file will be repeated in the current document until the end of the document is reached.

**Copying blocks to another document upon export.** When exporting blocks you can immediately apply them to the pages in another document, thereby propagating the blocks from one document to another. In order to do so choose an existing document to export the blocks to. If you activate the checkbox *Delete existing blocks* all blocks which may be present in the target document will be deleted before copying the new blocks into the document.

## 6.3.4 Converting PDF Form Fields to PDFlib Blocks

As an alternative to creating PDFlib blocks manually you can automatically convert PDF form fields to blocks. This is especially convenient if you have complicated PDF forms which you want to fill automatically with the PDFlib Personalization Server, or need to convert a large number of existing PDF forms for automated filling. In order to convert all form fields on a page to PDFlib blocks choose *PDFlib Blocks, Convert Form Fields, Current Page*. To convert all form fields in a document choose *All Pages* instead. Finally, you can convert only selected form fields (choose Acrobat's Form Tool or the Select Object Tool to select form fields) with *Selected Form Fields*.

**Form field conversion details.** Automatic form field conversion will convert form fields of the types selected in the *PDFlib Blocks, Convert Form Fields, Conversion Options...* dialog to blocks of type *Text*. By default all form field types will be converted. Attributes of the converted fields will be transformed to the corresponding block properties according to Table 6.2.

Table 6.2 Conversion of PDF form fields to PDFlib blocks

PDF form field attribute...	...will be converted to the PDFlib block property
<b>all fields</b>	
Position	General, Rect
Name	General, Name
Short Description	General, Description
Appearance, Text, Font	Text, fontname
Appearance, Text, Size	Text, fontsize; »auto« font size will be converted to a fixed font size of 2/3 of the block height, and the fitmethod will be set to »auto«. For multi-line fields/blocks this combination will automatically result in a suitable font size which may be smaller than the initial value of 2/3 of the block height.
Appearance, Text, Text Color	Text, textcolor and Text, fillcolor
Appearance, Border, Border Color	General, bordercolor
Appearance, Border, Background Color	General, backgroundcolor
Appearance, Border, Width	General, linewidth: Thin=1, Medium=2, Thick=3
Appearance, Common Properties, Form Field is...	General, Status: Visible=active, Hidden=ignore, Visible but doesn't print=ignore, Hidden but printable=active
Appearance, Common Prop., Orient.	General, orientate: 0=north, 90=west, 180=south, 270=east
<b>text fields</b>	
Options, Default	Text, defaulttext
Options, Alignment	General, position: Left={0 50}, Center={50 50}, Right={100, 50}
Options, Multi-line	Text, textflow: checked=true, unchecked=false
<b>radio buttons and check boxes</b>	
If »Default is Checked« is selected: Options, Radio Style and Options, Check Style	Text, defaulttext: Check=4, Circle=l, Cross=8, Diamond=u, Square=n, Star=H (these characters represent the respective symbols in the ZapfDingbats font)
<b>List boxes and combo boxes</b>	
Options, Selected (default) item	Text, defaulttext
<b>buttons</b>	
Options, Button Face Attributes, Text	Text, defaulttext

**Binding blocks to the corresponding form fields.** In order to keep PDF form fields and the generated PDFlib blocks synchronized, the generated blocks can be bound to the corresponding form fields. This means that the block tool will internally maintain the relationship of form fields and blocks. When the conversion process is activated again, bound blocks will be updated to reflect the attributes of the corresponding PDF form fields. Bound blocks are useful to avoid duplicate work: when a form is updated for interactive use, the corresponding blocks can automatically be updated, too.

If you do not want to keep the converted form fields after blocks have been generated you can choose the option *Delete converted Form Fields* in the *PDFlib Blocks, Convert*

*Form Fields, Conversion Options...* dialog. This option will permanently remove the form fields after the conversion process. Any actions (e.g., JavaScript) associated with the affected fields will also be removed from the document.

**Batch conversion.** If you have many PDF documents with form fields that you want to convert to PDFlib blocks you can automatically process an arbitrary number of documents using the batch conversion feature. The batch processing dialog is available via *PDFlib Blocks, Convert Form Fields, Batch conversion...*:

- ▶ The input files can be selected individually; alternatively the full contents of a folder can be processed.
- ▶ The output files can be written to the same folder where the input files are, or to a different folder. The output files can receive a prefix to their name in order to distinguish them from the input files.
- ▶ When processing a large number of documents it is recommended to specify a log file. After the conversion it will contain a full list of processed files as well as details regarding the result of each conversion along with possible error messages.

During the conversion process the converted PDF documents will be visible in Acrobat, but you cannot use any of Acrobat's menu functions or tools.



## 6.4 Standard Properties for Automated Processing

PDFlib supports general properties which can be assigned to any type of block. In addition there are properties which are specific to the block types *Text*, *Image*, and *PDF*. Some properties are *shared*, which means that they can be assigned to multiple blocks at once using the Block plugin.

Properties support the same data types as option lists (see Section 3.1.4, »Option Lists«, page 48) except handles and action lists.

Many block properties have the same name as options for *PDF\_fit\_image()* (e.g., *fitmethod*) and other functions, or as PDFlib parameters (e.g., *charspacing*). In these cases the behavior is exactly the same as the one documented for the respective option or parameter.

**Property processing in PDFlib.** The PDFlib Block functions *PDF\_fill\_\*block()* will process block properties in the following order:

- ▶ If the *backgroundcolor* property is present and contains a color space keyword different from *None*, the block rectangle will be filled with the specified color.
- ▶ All other properties except *bordercolor* and *linewidth* will be processed.
- ▶ If the *bordercolor* property is present and contains a color space keyword different from *None*, the block rectangle will be stroked with the specified color and linewidth.

There will be no clipping; if you want to make sure that the block contents do not exceed the block rectangle avoid *fitmethod nofit*.

If a separation color is used in a block property the specified spot color name must either be known to PDFlib internally (see Section 3.3.3, »Spot Colors«, page 64), or must have been specified earlier in the PDFlib client program using *PDF\_makespotcolor()*. Otherwise the block functions will fail.

### 6.4.1 General Properties

General properties apply to all kinds of blocks (*Text*, *Image*, *PDF*). They are required for block administration, describe the appearance of the block rectangle itself, and manage how the contents will be placed within the block. Required entries will automatically be generated by the PDFlib Block Plugin. Table 6.3 lists the general properties.

Table 6.3 General block properties

keyword	type	possible values and explanation
<b>Block administration</b>		
Name	string	(Required) Name of the block. Block names must be unique within a page, but not within a document. The three characters [ ] / are not allowed in block names.
Description	string	Human-readable description of the block's function, coded in PDFDocEncoding or Unicode (in the latter case starting with a BOM). This property is for user information only, and will be ignored when processing the block.
Locked	boolean	(Shareable) If true, the block and its properties can not be edited with the Block plugin. This property will be ignored when processing the block. Default: false.
Rect	rectangle	(Required) The block coordinates. The origin of the coordinate system is in the lower left corner of the page. However, the Block plugin will display the coordinates in Acrobat's notation, i.e., with the origin in the upper left corner of the page.
Status	keyword	Keyword describing how the block will be processed (Default: active): active      The block will be fully processed according to its properties. ignore      The block will be ignored. static      No variable contents will be placed; instead, the block's default text, image, or PDF contents will be used if available.
Subtype	keyword	(Required) Depending on the block type, one of Text, Image, or PDF.
Type	keyword	(Required) Always Block
<b>Block appearance</b>		
background-color	color	(Shareable) If this property is present and contains a color space keyword different from None, a rectangle will be drawn and filled with the supplied color. This may be useful to cover existing page contents. Default: None
bordercolor	color	(Shareable) If this property is present and contains a color space keyword different from None, a rectangle will be drawn and stroked with the supplied color. Default: None
linewidth	float	(Shareable; must be greater than 0) Stroke width of the line used to draw the block rectangle; only used if bordercolor is set. Default: 1
<b>Content placing</b>		
fitmethod	keyword	(Shareable) Strategy to use if the supplied content doesn't fit into the box. Possible values are auto, nofit, clip, meet <sup>1</sup> , slice <sup>1</sup> , and entire <sup>1</sup> . For simple text blocks, image, and PDF blocks this property will be interpreted according to Table 8.17 and Table 8.38). Default: auto. For textflow blocks where the block is too small for the text the interpretation is as follows: auto      fontsize and leading will be decreased until the text fits. nofit      Text will run beyond the bottom margin of the block. clip      Text will be clipped at the block margin.
orientate <sup>1</sup>	keyword	(Shareable) Specifies the desired orientation of the content when it is placed (see Table 8.38). Possible values are north, east, south, west. Default: north
position <sup>1</sup>	float list	(Shareable) One or two values specifying the position of the reference point within the content (see Table 8.17 for text, Table 8.38 for image/PDF). Default: o
rotate	float	(Shareable) Rotation angle in degrees by which the block will be rotated counter-clockwise before processing begins. The reference point is center of the rotation. Default: 0

1. This keyword or property is not supported for textflow blocks (text blocks with textflow=true).

### 6.4.2 Text Properties

Text-related properties apply to blocks of type *Text* (in addition to general properties). All text-related properties can be shared. The encoding for the text must be specified as an option for *PDF\_fill\_textblock()* when filling the block unless the *font* option has been supplied.

**Properties for all text blocks.** Text blocks can be single-line or multi-line. Table 6.4 lists the text-related properties which apply to both types.

Table 6.4 Text block properties

keyword	type	possible values and explanation
charspacing	float or percentage	Character spacing (see Table 8.16). Percentages are based on fontsize. Default: 0
defaulttext	string	Text which will be used if no substitution text is supplied by the client <sup>1</sup>
fillcolor	color	Fill color of the text. Default: gray 0 (=black)
fontname <sup>2</sup>	string	Name of the font as required by <i>PDF_load_font()</i>
fontsize <sup>2</sup>	float	Size of the font in points
fontstyle	keyword	Font style, must be one of normal, bold, italic, or bolditalic (see Table 8.14)
horizscaling	float or percentage	Horizontal text scaling (see Table 8.16). Default: 100%
italicangle	float	Italic angle of text in degrees (see Table 8.16). Default: 0
kerning	boolean	Kerning behavior (see Table 8.16). Default: false
margin	float list	One or two float values describing additional horizontal and vertical extensions of the text box (see Table 8.17). Default: 0
monospace	integer 1...2048	Forces the same width for all characters in the font (see Table 8.14) Default: absent (metrics from the font will be used)
overline	boolean	Overline mode (see Table 8.16). Default: false
strikeout	boolean	Strikeout mode (see Table 8.16). Default: false
strokecolor	color	Stroke color of the text. Default: gray 0 (=black)
textflow	boolean	Controls single- or multiline processing (default: false): false      Text can span a single line and will be processed with <i>PDF_fit_text()</i> . true        Text can span multiple lines and will be processed with <i>PDF_fit_textflow()</i> . The general properties position, fitmethod, and orientate will be ignored. In addition to the standard text properties all textflow-related properties (see Table 6.5) can be specified.
textrendering	integer	Text rendering mode (see Table 8.16). Default: 0
textrise	float or percentage	Text rise parameter (see Table 8.16). Percentages are based on fontsize. Default: 0
underline	boolean	Underline mode (see Table 8.16). Default: false
wordspacing	float or percentage	Word spacing (see Table 8.16). Percentages are based on fontsize. Default: 0

1. The text will be interpreted in winansi encoding or Unicode.  
2. This property is required in a text block; it will automatically be enforced by the PDFlib Block plugin.

**Properties for textflow blocks.** Textflow-related properties apply to blocks of type *Text* where the *textflow* property is *true*. The text-related properties will be used to construct the initial option list for processing the textflow (corresponding to the *optlist* parameter of *PDF\_create\_textflow()*). Inline option lists can not be specified with the plugin, but they can be supplied on the server as part of the text contents when filling the block with *PDF\_fill\_textblock()*. All textflow-related properties can be shared. Table 6.5 lists the textflow-related properties.

Table 6.5 Textflow block properties

keyword	type	possible values and explanation
<b>Option for text semantics:</b>		
<i>tabalignchar</i>	integer	Unicode value of the character at which decimal tabs will be aligned. Default: the ' ' character (U+002E)
<b>Options for controlling the text layout:</b>		
<i>alignment</i>	keyword	Specifies formatting for lines in a paragraph (default: left): <div> <div>left</div> <div>center</div> <div>right</div> <div>justify</div> </div> left-aligned, starting at <i>leftindent</i> centered between <i>leftindent</i> and <i>rightindent</i> right-aligned, ending at <i>rightindent</i> left- and right-aligned
<i>fixedleading</i>	boolean	If true, the first leading value found in each line will be used. Otherwise the maximum of all leading values in the line will be used. Default: false
<i>horthabsize</i>	float or percentage	Width of a horizontal tab <sup>1</sup> . The interpretation depends on the <i>horthabmethod</i> option. Default: 7.5%
<i>horthabmethod</i>	keyword	Treatment of horizontal tabs in the text. If the determined position is to the left of the current text position, the tab will be ignored (default: relative): <div> <div>relative</div> <div>typewriter</div> <div>ruler</div> </div> The position will be advanced by the amount specified in <i>horthabsize</i> . The position will be advanced to the next multiple of <i>horthabsize</i> . The position will be advanced to the n-th tab value in the ruler option, where n is the number of tabs found in the line so far. If n is larger than the number of tab positions the relative method will be applied.
<i>lastalignment</i>	keyword	Formatting for the last line in a paragraph. All keywords of the <i>alignment</i> option are supported, plus the following (default: auto): <div> <div>auto</div> </div> Use the value of the <i>alignment</i> option unless it is justify. In the latter case left will be used.
<i>leading</i>	float or percentage	Distance between adjacent text baselines in user coordinates, or as a percentage of the font size. Default: 100%
<i>parindent</i>	float or percentage	Left indent of the first line of a paragraph <sup>1</sup> . The amount will be added to <i>leftindent</i> . Specifying this option within a line will act like a tab. Default: 0
<i>rightindent</i> <i>leftindent</i>	float or percentage	Right or left indent of all text lines <sup>1</sup> . If <i>leftindent</i> is specified within a line and the determined position is to the left of the current text position, this option will be ignored for the current line. Default: 0
<i>ruler</i> <sup>2</sup>	list of floats or percentages	List of absolute tab positions for <i>horthabmethod=ruler</i> <sup>1</sup> . The list may contain up to 32 non-negative entries in ascending order. Default: integer multiples of <i>horthabsize</i>
<i>tabalignment</i>	list of keywords	Alignment for tab stops. Each entry in the list defines the alignment for the corresponding entry in the ruler option (default: left) <div> <div>center</div> <div>decimal</div> <div>left</div> <div>right</div> </div> Text will be centered at the tab position. The first instance of <i>tabalignchar</i> will be left-aligned at the tab position. If no <i>tabalignchar</i> is found, right alignment will be used instead. Text will be left-aligned at the tab position. Text will be right-aligned at the tab position.

Table 6.5 Textflow block properties

keyword	type	possible values and explanation
<b>Options for controlling the line-breaking algorithm:</b>		
adjust-method	keyword	Method used to adjust a line when a text portion doesn't fit into a line after compressing or expanding the distance between words subject to the limits specified by the minspacing and maxspacing options. Default: auto
	auto	The following methods are applied in order: shrink, spread, nofit, split.
	clip	Same as nofit, except that the long part at the right edge of the fit box (taking into account the rightindent option) will be clipped.
	nofit	The last word will be moved to the next line provided the remaining (short) line will not be shorter than the percentage specified in the nofitlimit option. Even justified paragraphs may look slightly ragged.
	shrink	If a word doesn't fit in the line the text will be compressed subject to shrinklimit. If it still doesn't fit the nofit method will be applied.
	split	The last word will not be moved to the next line, but will forcefully be hyphenated. For text fonts a hyphen character will be inserted, but not for symbol fonts.
	spread	The last word will be moved to the next line and the remaining (short) line will be justified by increasing the distance between characters in a word, subject to spreadlimit. If justification still cannot be achieved the nofit method will be applied.
maxspacing minspacing	float or percentage	The maximum or minimum distance between words (in user coordinates, or as a percentage of the width of the space character). The calculated word spacing is limited by the provided values (but the wordspacing option will still be added). Defaults: minspacing=50%, maxspacing=500%
nofitlimit	float or percentage	Lower limit for the length of a line with the nofit method (in user coordinates or as a percentage of the width of the fitbox). Default: 75%.
shrinklimit	percentage	Lower limit for compressing text with the shrink method; the calculated shrinking factor is limited by the provided value, but will be multiplied with the value of the horzscaling option. Default: 85%
spreadlimit	float or percentage	Upper limit for the distance between two characters for the spread method (in user coordinates or as a percentage of the font size); the calculated character distance will be added to the value of the charspacing option. Default: 0

- 1. In user coordinates, or as a percentage of the width of the fit box
- 2. Rulers can be edited in the »Tabs« section of the Block properties dialog.

### 6.4.3 Image Properties

Image-related properties apply to blocks of type *Image* (in addition to general properties). All image-related properties can be shared. Table 6.6 lists the image-related properties.

Table 6.6 *Image block properties*

keyword	type	possible values and explanation
defaultimage	string	Path name of an image which will be used if no substitution image is supplied by the client. It is recommended to use file names without absolute paths, and use the SearchPath feature in the PPS client application. This will make block processing independent from platform and file system details.
dpi	float list	One or two values specifying the desired image resolution in pixels per inch in horizontal and vertical direction. With the value 0 the image's internal resolution will be used if available, or 72 dpi otherwise. This property will be ignored if the fitmethod property has been supplied with one of the keywords auto, meet, slice, or entire. Default: 0
scale	float list	One or two values specifying the desired scaling factor(s) in horizontal and vertical direction. This option will be ignored if the fitmethod property has been supplied with one of the keywords auto, meet, slice, or entire. Default: 1

### 6.4.4 PDF Properties

PDF-related properties apply to blocks of type *PDF* (in addition to general properties). All PDF-related properties can be shared. Table 6.7 lists the PDF-related properties.

Table 6.7 *PDF block properties*

keyword	type	possible values and explanation
defaultpdf	string	Path name of a PDF document which will be used if no substitution PDF is supplied by the client. It is recommended to use file names without absolute paths, and use the SearchPath feature in the PPS client application. This will make block processing independent from platform and file system details.
default-pdfpage	integer	Page number of the page in the default PDF document. Default: 1
scale	float list	One or two values specifying the desired scaling factor(s) in horizontal and vertical direction. This option will be ignored if the fitmethod property has been supplied with one of the keywords auto, meet, slice, or entire. Default: 1
pdiusebox	keyword	(Possible values: media, crop, bleed, trim, art) Use the placed page's MediaBox, CropBox, BleedBox, TrimBox, or ArtBox for determining its size (see Table 8.43). Default: crop

### 6.4.5 Custom Properties

Custom properties apply to blocks of any type of block (in addition to general and type-specific properties). Custom properties are optional, and can not be shared. Table 6.8 lists the custom properties.

Table 6.8 Custom block properties

keyword	type	possible values and explanation
any name not containing the three characters [ ] /	string, name, float, float list	The interpretation of the values corresponding to custom properties is completely up to the client application.

## 6.5 Querying Block Names and Properties

In addition to automatic block processing PDFlib supports some features which can be used to enumerate block names and query standard or custom properties.

**Finding the numbers and names of blocks.** The client code must not even know the names or numbers of the blocks on an imported page since these can also be queried. The following statement returns the number of blocks on the page:

```
blockcount = PDF_get_pdi_value(p, "vdp/blockcount", doc, page, 0);
```

The following statement returns the name of block number 5 on the page (block counting starts at 0), or an empty string if no such block exists (however, an exception will be thrown if the *pdiwarning* parameter or option is set to *true*):

```
blockname = PDF_get_pdi_parameter(p, "vdp/Blocks[5]/Name", doc, page, 0, &len);
```

The returned block name can subsequently be used to query the block's properties or populate the block with text, image, or PDF content.

In the path syntax for addressing block properties the following expressions are equivalent, assuming that the block with the sequential *<number>* has its *Name* property set to *<blockname>*:

```
Blocks[<number>]/  
Blocks/<blockname>/
```

**Finding block coordinates.** The two coordinate pairs (*llx*, *lly*) and (*urx*, *ury*) describing the lower left and upper right corner of a block named *foo* can be queried as follows:

```
llx = PDF_get_pdi_value(p, "vdp/Blocks/foo/Rect[0]", doc, page, 0);  
lly = PDF_get_pdi_value(p, "vdp/Blocks/foo/Rect[1]", doc, page, 0);  
urx = PDF_get_pdi_value(p, "vdp/Blocks/foo/Rect[2]", doc, page, 0);  
ury = PDF_get_pdi_value(p, "vdp/Blocks/foo/Rect[3]", doc, page, 0);
```

Note that these coordinates are provided in the default user coordinate system (with the origin in the bottom left corner, possibly modified by the page's CropBox), while the Block plugin displays the coordinates according to Acrobat's user interface coordinate system with an origin in the upper left corner of the page. Also note that the *topdown* parameter is not taken into account when querying block coordinates.

**Querying custom properties.** Custom properties can be queried as in the following example, where the property *zipcode* is queried from a block named *b1*:

```
zip = PDF_get_pdi_parameter(p, "vdp/Blocks/b1/Custom/zipcode", doc, page, 0, &len);
```

**Name space for custom properties.** In order to avoid confusion when PDF documents from different sources are exchanged, it is recommended to use an Internet domain name as a company-specific prefix in all custom property names, followed by a colon ':' and the actual property name. For example, ACME corporation would use the following property names:

```
acme.com:digits  
acme.com:refnumber
```



Since standard and custom properties are stored differently in the block, standard PDFlib property names (as defined in Section 6.4, »Standard Properties for Automated Processing«, page 153) will never conflict with custom property names.

## 6.6 PDFlib Block Specification

The PDFlib Block syntax is fully compliant with the PDF Reference, which specifies an extension mechanism that allows applications to store private data attached to the data structures comprising a PDF page. A detailed description of the PDFlib block syntax is provided here for the benefit of users who wish to create PDFlib blocks by other means than the PDFlib block plugin. Plugin users can safely skip this section.

### 6.6.1 PDF Object Structure for PDFlib Blocks

The page dictionary contains a */PieceInfo* entry, which has another dictionary as value. This dictionary contains the key */PDFlib* with an application data dictionary as value. The application data dictionary contains two standard keys listed in Table 6.9.

Table 6.9 Entries in a PDFlib application data dictionary

Key	type	value
<i>LastModified</i>	date string	(Required) The date and time when the blocks on the page were created or most recently modified.
<i>Private</i>	dictionary	(Required) A block list (see Table 6.10)

A Block list is a dictionary containing general information about block processing, plus a list of all blocks on the page. Table 6.10 lists the keys in a block list dictionary.

Table 6.10 Entries in a block list dictionary

Key	type	value
<i>Version</i>	number	(Required) The version number of the block specification to which the file complies. This document describes version 3 of the block specification.
<i>Blocks</i>	dictionary	(Required) Each key is a name object containing the name of a block; the corresponding value is the block dictionary for this block (see Table 6.12). The <i>/Name</i> key in the block dictionary must be identical to the block's name in this dictionary.
<i>PluginVersion</i>	string	(Required unless the <i>pdfmark</i> key is present <sup>1</sup> ) A string containing a version identification of the PDFlib Block plugin which has been used to create the blocks.
<i>pdfmark</i>	boolean	(Required unless the <i>PluginVersion</i> key is present <sup>1</sup> ) Must be true if the block list has been generated by use of pdfmarks.

1. Exactly one of the *PluginVersion* and *pdfmark* keys must be present.

**Data types for block properties.** Properties support the same data types as option lists (see Section 3.1.4, »Option Lists«, page 48) except handles and action lists. Table 6.11 details how these types are mapped to PDF data types.

Table 6.11 Data types for block properties

block type	PDF type	remarks
boolean	boolean	
string	string	
keyword	name	It is an error to provide keywords outside the list of keywords supported by a particular property.
float, integer	number	While option lists support both point and comma as decimal separators, PDF numbers support only point.

Table 6.11 Data types for block properties

block type	PDF type	remarks
percentage	array with two elements	The first element in the array is the number, the second element is a string containing a percent character.
color	array with two elements	<p>The first element in the array specifies a color space, and the second element specifies a color value as follows. The following entries are supported for the first element in the array:</p> <p><i>/DeviceGray</i> The second element is a single gray value.</p> <p><i>/DeviceRGB</i> The second element is an array of three RGB values.</p> <p><i>/DeviceCMYK</i> The second element is an array of four CMYK values.</p> <p><i>[/Separation/spotname]</i> The first element is an array containing the keyword <i>/Separation</i> and a color name. The second element is a tint value.</p> <p><i>[/Lab]</i> The first element is an array containing the keyword <i>/Lab</i>. The second element is an array of three Lab values.</p> <p>To specify the absence of color the respective property must be omitted.</p>

**Block dictionary keys.** Block dictionaries may contain the keys in Table 6.12. Only keys from one of the Text, Image or PDF groups may be present depending on the */Subtype* key in the General group (see Table 6.3).

Table 6.12 Entries in a block dictionary

Key	type	value
general properties		(Some keys are required) General properties according to Table 6.3
text properties		(Optional) Text and textflow properties according to Table 6.4 and Table 6.5
image properties		(Optional) Image properties according to Table 6.6
PDF properties		(Optional) PDF properties according to Table 6.7
Custom	dict	(Optional) A dictionary containing key/value pairs for custom properties according to Table 6.8.
Internal	dict	(Optional) This key is reserved for private use, and applications should not depend on its presence or specific behavior. Currently it is used for maintaining the relationship between converted form fields and corresponding blocks.

**Example.** The following fragment shows the PDF code for two blocks, a text block called *job\_title* and an image block called *logo*. The text block contains a custom property called *format*:

```
<<
  /Contents 12 0 R
  /Type /Page
  /Parent 1 0 R
  /MediaBox [ 0 0 595 842 ]
  /PieceInfo << /PDFlib 13 0 R >>
>>

13 0 obj
<<
  /Private <<
    /Blocks <<
      /job_title 14 0 R
```

```

        /logo 15 0 R
    >>
    /Version 3
    /PluginVersion (2.0.0)
>>
/LastModified (D:20040513200730)
>>
endobj

14 0 obj
<<
    /Type /Block
    /Rect [ 70 740 200 800 ]
    /Name /job_title
    /Subtype /Text
    /fitmethod /auto
    /fontname (Helvetica)
    /fontsize 12
    /Custom << /format 5 >>
>>
endobj

15 0 obj
<<
    /Type /Block
    /Rect [ 250 700 400 800 ]
    /Name /logo
    /Subtype /Image
    /fitmethod /auto
>>

```

## 6.6.2 Generating PDFlib Blocks with pdfmarks

As an alternative to creating PDFlib blocks with the plugin, blocks can be created by inserting appropriate pdfmark commands into a PostScript stream, and distilling it to PDF. Details of the pdfmark operator are discussed in the Acrobat documentation. The following fragment shows pdfmark operators which can be used to generate the block definition in the preceding section:

```

% ----- Setup for the blocks on a page -----
[/_objdef {B1} /type /dict /OBJ pdfmark          % Blocks dict

[{ThisPage} <<
    /PieceInfo <<
        /PDFlib <<
            /LastModified (D:20040513200730)
            /Private <<
                /Version 3
                /pdfmark true
                /Blocks {B1}
            >>
        >>
    >>
>> /PUT pdfmark

% ----- text block -----
[{B1} <<
    /job_title <<

```

```
        /Type /Block
        /Name /job_title
        /Subtype /Text
        /Rect [ 70 740 200 800 ]
        /fitmethod /auto
        /fontsize 12
        /fontname (Helvetica)
        /Custom << /format 5 >>
    >>
>> /PUT pdfmark

% ----- image block -----
[{B1} <<
    /logo <<
        /Type /Block
        /Name /logo
        /Subtype /Image
        /Rect [ 250 700 400 800 ]
        /fitmethod /auto
    >>
>> /PUT pdfmark
```





# 7 Generating various PDF Flavors

## 7.1 Acrobat and PDF Versions

At the user's option PDFlib generates output according to PDF 1.3 (Acrobat 4), PDF 1.4 (Acrobat 5), or PDF 1.5 (Acrobat 6). This can be controlled with the *compatibility* option in *PDF\_begin\_document()*. In PDF 1.3 compatibility mode the PDFlib features for PDF 1.4 (listed in Table 7.1) and PDF 1.5 (listed in Table 7.2) will not be available. Trying to use one of these features in PDF 1.3 mode will result in an exception.

Table 7.1 PDFlib features for PDF 1.4 which are not available in PDF 1.3 compatibility mode

Feature	PDFlib API functions and parameters
smooth shadings (color blends)	<i>PDF_shading_pattern()</i> , <i>PDF_shfill()</i> , <i>PDF_shading()</i>
soft masks	<i>PDF_load_image()</i> with the <i>masked</i> option referring to an image with more than 1 bit pixel depth
128-bit encryption	<i>PDF_begin_document()</i> with the <i>userpassword</i> , <i>masterpassword</i> , <i>permissions</i> options
extended permission settings	<i>PDF_begin_document()</i> with <i>permissions</i> option, see Table 7.3
certain CMaps for CJK fonts	<i>PDF_load_font()</i> , see Table 4.6
transparency and other graphics state options	<i>PDF_create_gstate()</i> with options <i>alphaishape</i> , <i>blendmode</i> , <i>opacityfill</i> , <i>opacitystroke</i> , <i>textknockout</i>
certain options for actions	<i>PDF_create_action()</i> , see Table 8.46
certain options for annotations	<i>PDF_create_annotation()</i> , see Table 8.48
certain field options	<i>PDF_create_field()</i> and <i>PDF_create_fieldgroup()</i> , see Table 8.49
Tagged PDF	<i>tagged</i> option in <i>PDF_begin_document()</i>

In PDF 1.3 or 1.4 compatibility modes the PDFlib features for PDF 1.5 listed in Table 7.2 will not be available. Trying to use one of these features in PDF 1.3 or PDF 1.4 mode will result in an exception.

Table 7.2 PDFlib features for PDF 1.5 which are not available in PDF 1.3 and 1.4 compatibility modes

Feature	PDFlib API functions and parameters
certain field options	<i>PDF_create_field()</i> and <i>PDF_create_fieldgroup()</i> , see Table 8.49
certain annotation options	<i>PDF_create_annotation()</i> see Table 8.48
extended permission settings	<i>permissions=plainmetadata</i> in <i>PDF_begin_document()</i> , see Table 7.3
certain CMaps for CJK fonts	<i>PDF_load_font()</i> , see Table 4.6
Tagged PDF	certain options for <i>PDF_begin_item()</i> , see Table 8.54 and Table 8.55
Layers	<i>PDF_define_layer()</i> , <i>PDF_begin_layer()</i> , <i>PDF_end_layer()</i> , <i>PDF_layer_dependency()</i>

## 7.2 Encrypted PDF

### 7.2.1 Strengths and Weaknesses of PDF Security

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

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

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

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

- ▶ For PDF versions up to and including 1.3 (i.e., Acrobat 4) the key length is 40 bits.
- ▶ For PDF version 1.4 and above the key length is 128 bits. This requires Acrobat 5 or above. For PDF 1.5 the key length will also be 128 bits, but a slightly different encryption will be used, which requires Acrobat 6.

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

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

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

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



## 7.2.2 Protecting Documents with PDFlib

**Passwords.** Passwords can be set with the *userpassword* and *masterpassword* options in *PDF\_begin\_document()*. PDFlib interacts with the client-supplied passwords in the following ways:

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

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

**Permissions.** Access restrictions can be set with the *permissions* option in *PDF\_begin\_document()*. It contains one or more access restriction keywords. When setting the *permissions* option the *masterpassword* option must also be set, because otherwise Acrobat users could easily remove the permission settings. By default, all actions are allowed. Specifying an access restriction will disable the respective feature in Acrobat. Access restrictions can be applied without any user password. Multiple restriction keywords can be specified as in the following example:

```
PDF_begin_document(p, filename, 0, "permissions {noprint nocopy}");
```

Table 7.3 lists all supported access restriction keywords. As detailed in the table, some keywords require PDF 1.4 or 1.5 compatibility. They will be rejected if the PDF output version is too old.

Table 7.3 Access restriction keywords for the *permissions* option in *PDF\_begin\_document()*

keyword	explanation
<i>noprint</i>	Acrobat will prevent printing the file.
<i>nomodify</i>	Acrobat will prevent users from adding form fields or making any other changes.
<i>nocopy</i>	Acrobat will prevent copying and extracting text or graphics, and will disable the accessibility interface
<i>noannots</i>	Acrobat will prevent adding or changing comments or form fields.
<i>noforms</i>	(PDF 1.4) Acrobat will prevent form field filling, even if <i>noannots</i> hasn't been specified.
<i>noaccessible</i>	(PDF 1.4) Acrobat will prevent extracting text or graphics for accessibility purposes (such as a screenreader program)
<i>noassemble</i>	(PDF 1.4) Acrobat will prevent inserting, deleting, or rotating pages and creating bookmarks and thumbnails, even if <i>nomodify</i> hasn't been specified.
<i>nohiresprint</i>	(PDF 1.4) Acrobat will prevent high-resolution printing. If <i>noprint</i> hasn't been specified printing is restricted to the »print as image« feature which prints a low-resolution rendition of the page.
<i>plainmeta-data</i>	(PDF 1.5) Keep document metadata unencrypted even for encrypted documents.

## 7.3 Web-Optimized (Linearized) PDF

PDFlib can apply a process called linearization to PDF documents (linearized PDF is called *Optimized* in Acrobat 4, and *Fast Web View* in Acrobat 5 and above). Linearization reorganizes the objects within a PDF file and adds supplemental information which can be used for faster access.

While non-linearized PDFs must be fully transferred to the client, a Web server can transfer linearized PDF documents one page at a time using a process called byte-serving. It allows Acrobat (running as a browser plugin) to retrieve individual parts of a PDF document separately. The result is that the first page of the document will be presented to the user without having to wait for the full document to download from the server. This provides enhanced user experience.

Note that the Web server streams PDF data to the browser, not PDFlib. Instead, PDFlib prepares the PDF files for byteserving. All of the following requirements must be met in order to take advantage of byteserving PDFs:

- ▶ The PDF document must be linearized, which can be achieved with the *linearize* option in *PDF\_begin\_document()*. In Acrobat you can check whether a file is linearized by looking at its document properties (»Fast Web View: yes«).
- ▶ The Web server must support byteserving. The underlying byterange protocol is part of HTTP 1.1 and therefore implemented in all current Web servers. In particular, the following Web servers support byteserving:

Microsoft Internet Information Server (IIS) 3.0 and above

Apache 1.2.1 and above; however, Apache 1.3.14 (but not other versions) has a bug which prevents byteserving

- ▶ The user must use Acrobat as a Browser plugin, and have page-at-a-time download enabled in Acrobat (Acrobat 6: *Edit, Preferences, [General...], Internet, Allow fast web view*; Acrobat 5: *Edit, Preferences, General..., Options, Allow Fast Web view*). Note that this is enabled by default.

The larger a PDF file (measured in pages or MB), the more it will benefit from linearization when delivered over the Web.

*Note* Linearizing a PDF document generally slightly increases its file size due to the additional linearization information.

**Temporary storage requirements for linearization.** PDFlib must create the full document before it can be linearized; the linearization process will be applied in a separate step after the document has been created. For this reason PDFlib has additional storage requirements for linearization. Temporary storage will be required which has roughly the same size as the generated document (without linearization). Subject to the *inmemory* option in *PDF\_begin\_document()* PDFlib will place the linearization data either in memory or on a temporary disk file.

## 7.4 PDF/X

### 7.4.1 The PDF/X Family of Standards

The PDF/X formats specified in the ISO 15930 standards family strive to provide a consistent and robust subset of PDF which can be used to deliver data suitable for commercial printing<sup>1</sup>. PDFlib can generate output and process input conforming to the following flavors of PDF/X:

**PDF/X-1:2001 and PDF/X-1a:2001 as defined in ISO 15930-1.** These standards for »blind exchange« (exchange of print data without the requirement for any prior technical discussions) are based on PDF 1.3 and support CMYK and spot color data. RGB and device-independent colors (ICC-based, Lab) are explicitly prohibited. PDF/X-1:2001 supports a mechanism to integrate legacy files (such as TIFF/IT) in a PDF workflow, and is considered obsolete. PDF/X-1a:2001 does not contain this legacy support, and is widely used (especially in North America) for the exchange of publication ads and other applications.

**PDF/X-1a:2003 as defined in ISO 15930-4.** This standard is the successor to PDF/X-1a:2001. It is based on PDF 1.4, with some features (e.g. transparency) prohibited. PDF/X-1a:2003 is a strict subset of PDF/X-3:2003, and supports CMYK and spot color, and CMYK output devices.

**PDF/X-2:2003 as defined in ISO 15930-5.** This standard is targeted at »partial exchange« which requires more discussion between supplier and receiver of a file. PDF documents according to this standard can reference external entities (point to other PDF pages external to the current document). PDF/X-2:2003 is based on PDF 1.4. As a superset of PDF/X-3:2003 it supports device independent colors.

**PDF/X-3:2002 as defined in ISO 15930-3.** This standard is based on PDF 1.3, and supports modern workflows based on device-independent color in addition to grayscale, CMYK, and spot colors. It is especially popular in European countries. Output devices can be monochrome, RGB, or CMYK.

**PDF/X-3:2003 as defined in ISO 15930-6.** This standard is the successor to PDF/X-3:2002. It is based on PDF 1.4, with some features (e.g. transparency) prohibited. When one of the PDF/X standards is referenced below without any standardization year, all versions of the respective standard are meant. For example, *PDF/X-3* means *PDF/X-3:2002* and *PDF-X/3:2003*.

*Note* PANTONE® Colors are not supported in the PDF/X-1:2001, PDF/X-1a:2001, and PDF/X-1a:2003 modes.

1. See [www.pdfx3.org](http://www.pdfx3.org) and [www.pdf-x.com](http://www.pdf-x.com)

## 7.4.2 Generating PDF/X-conforming Output

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

- ▶ PDFlib will automatically take care of several formal settings for PDF/X, such as PDF version number and PDF/X conformance keys.
- ▶ The PDFlib client must explicitly use certain function calls or parameter settings as detailed in Table 7.4.
- ▶ The PDFlib client must refrain from using certain function calls and parameter settings as detailed in Table 7.5.
- ▶ Additional rules apply when importing pages from existing PDF/X-conforming documents (see Section 7.4.3, »Importing PDF/X Documents with PDI«, page 174).

**Required operations.** Table 7.4 lists all operations required to generate PDF/X-compatible output. The items apply to all PDF/X conformance levels unless otherwise noted. Not calling one of the required functions while in PDF/X mode will trigger an exception.

Table 7.4 Operations which must be applied for PDF/X compatibility

Item	PDFlib function and parameter requirements for PDF/X compatibility
conformance level	The <code>pdfx</code> option in <code>PDF_begin_document()</code> must be set to the required PDF/X conformance level.
output condition (output intent)	<code>PDF_load_iccprofile()</code> with <code>usage = outputintent</code> or <code>PDF_process_pdi()</code> with <code>action = copyoutputintent</code> must be called exactly once for each document. If spot colors from one of the built-in color libraries are used an output intent ICC profile must be embedded (using a standard output condition is not allowed in this case). PDF/X-1 and PDF/X-1a: the output device must be a monochrome or CMYK device; PDF/X-3: the output device must be a monochrome, RGB, or CMYK device. If ICC-based colors or Lab colors are used in the file, an output device ICC profile must be embedded.
font embedding	Set the embedding option of <code>PDF_load_font()</code> to true to enable font embedding.
page sizes	The page boxes, which are settable via the <code>CropBox</code> , <code>BleedBox</code> , <code>TrimBox</code> , and <code>ArtBox</code> parameters, must satisfy all of the following requirements: <ul style="list-style-type: none"><li>▶ The <code>TrimBox</code> or <code>ArtBox</code> must be set, but not both of these box entries. If both <code>TrimBox</code> and <code>ArtBox</code> are missing PDFlib will take the <code>CropBox</code> (if present) as the <code>TrimBox</code>, and the <code>MediaBox</code> if the <code>CropBox</code> is also missing.</li><li>▶ The <code>BleedBox</code>, if present, must fully contain the <code>ArtBox</code> and <code>TrimBox</code>.</li><li>▶ The <code>CropBox</code>, if present, must fully contain the <code>ArtBox</code> and <code>TrimBox</code>.</li></ul>
grayscale color	PDF/X-3: the <code>defaultgray</code> option in <code>PDF_begin_page_ext()</code> must be set if grayscale images are used or <code>PDF_setcolor()</code> is used with a gray color space, and the PDF/X output condition is not a CMYK or grayscale device.
RGB color	PDF/X-3: the <code>defaultrgb</code> option in <code>PDF_begin_page_ext()</code> must be set if RGB images are used or <code>PDF_setcolor()</code> is used with an RGB color space, and the PDF/X output condition is not an RGB device.
CMYK color	PDF/X-3: the <code>defaultcmky</code> option in <code>PDF_begin_page_ext()</code> must be set if CMYK images are used or <code>PDF_setcolor()</code> is used with a CMYK color space, and the PDF/X output condition is not a CMYK device.
document info keys	The <code>Creator</code> and <code>Title</code> info keys must be set with <code>PDF_set_info()</code> .

**Prohibited operations.** Table 7.5 lists all operations which are prohibited when generating PDF/X-compatible output. The items apply to all PDF/X conformance levels unless otherwise noted. Calling one of the prohibited functions while in PDF/X mode will trigger an exception. However, images with unacceptable compression (GIF and LZW-compressed TIFF images) will not result in an exception subject to the `imagewarning` param-

eter. Similarly, if an imported PDF page doesn't match the current PDF/X conformance level, the corresponding PDI call will fail without an exception (subject to the *pdi-warning* parameter).

Table 7.5 Operations which must be avoided to achieve PDF/X compatibility

Item	PDFlib functions and parameters to be avoided for PDF/X compatibility
grayscale color	PDF/X-1 and PDF/X-1a: the <i>defaultgray</i> option in <i>PDF_begin_page_ext()</i> must be avoided.
RGB color	PDF/X-1 and PDF/X-1a: RGB images and the <i>defaultrgb</i> option in <i>PDF_begin_page_ext()</i> must be avoided.
CMYK color	PDF/X-1 and PDF/X-1a: the <i>defaultcmymk</i> option in <i>PDF_begin_page_ext()</i> must be avoided.
ICC-based color	PDF/X-1 and PDF/X-1a: the <i>iccbasedgray/rgb/cmyk</i> color space in <i>PDF_setcolor()</i> and the <i>setcolor:iccprofilegray/rgb/cmyk</i> parameters must be avoided.
Lab color	PDF/X-1 and PDF/X-1a: the Lab color space in <i>PDF_setcolor()</i> must be avoided.
annotations and form fields	Annotations inside the <i>BleedBox</i> (or <i>TrimBox</i> / <i>ArtBox</i> if no <i>BleedBox</i> is present) must be avoided: <i>PDF_create_annotation()</i> , <i>PDF_create_field()</i> and related deprecated functions.
actions and JavaScript	All actions including JavaScript must be avoided: <i>PDF_create_action()</i> , and related deprecated functions
images	PDF/X-1 and PDF/X-1a: images with RGB, ICC-based, YCbCr, or Lab color must be avoided. For colorized images the alternate color of the spot color used must satisfy the same conditions.  GIF images and LZW-compressed TIFF images must be avoided.  The OPI-1.3 and OPI-2.0 options in <i>PDF_load_image()</i> must be avoided.
transparency	Soft masks for images must be avoided: the <i>mask</i> option for <i>PDF_load_image()</i> must be avoided unless the mask refers to a 1-bit image.  The <i>opacityfill</i> and <i>opacitystroke</i> options for <i>PDF_create_gstate()</i> must be avoided unless they have a value of 1.
viewer preferences / view and print areas	When the <i>viewarea</i> , <i>viewclip</i> , <i>printarea</i> , and <i>printclip</i> keys are used for <i>PDF_set_parameter()</i> values other than <i>media</i> or <i>bleed</i> are not allowed.
document info keys	Trapped info key values other than <i>True</i> or <i>False</i> for <i>PDF_set_info()</i> must be avoided.
security	PDF/X-1, but not PDF/X-1a: <i>userpassword</i> option and the value <i>noprint</i> for the permissions option in <i>PDF_begin_document()</i> must be avoided;  PDF/X-1a and PDF/X-3: <i>userpassword</i> , <i>masterpassword</i> , and <i>permissions</i> options in <i>PDF_begin_document()</i> must be avoided.
PDF version / compatibility	Using the <i>compatibility</i> option in <i>PDF_begin_document()</i> must be avoided since PDFlib will do this automatically (see Table 7.1 and Table 7.2 for details on features in different PDF versions)  PDF/X-1:2001, PDF/X-1a:2001, and PDF/X-3:2002 are based on PDF 1.3. Operations that require PDF 1.4 or above (such as transparency or soft masks) must be avoided.  PDF/X-1a:2003, PDF/X-2:2003, and PDF/X-3:2003 are based on PDF 1.4. Operations that require PDF 1.5 (such as layers) must be avoided.
PDF import (PDI)	Imported documents must conform to the same PDF/X level as the output document, and must have been prepared according to the same output intent.

**Standard output conditions.** The output condition defines the intended target device, which is mainly useful for reliable proofing. The output intent can either be specified by an ICC profile or by supplying the name of a standard output intent. Table 7.6 lists the names of standard output intents known to PDFlib. Additional standard output intents can be defined using the *StandardOutputIntent* resource category (see Section 3.1.6,

»Resource Configuration and File Searching«, page 51). It is the user’s responsibility to add only those names as standard output intents which can be recognized by PDF/X-processing software.

Table 7.6 Standard output intents for PDF/X

Output intent	description
CGATS TR 001	SWOP (publication) printing in USA
OF COM PO P1 F6o	ISO 12647-2, positive plates, paper type 1 (gloss-coated)
OF COM PO P2 F6o	ISO 12647-2, positive plates, paper type 2 (matte-coated)
OF COM PO-P3 F6o <sup>1</sup>	ISO 12647-2, positive plates, paper type 3 (light weight coated web)
OF COM PO P4 F6o	ISO 12647-2, positive plates, paper type 4 (uncoated white offset)
OF COM NE P1 F6o	ISO 12647-2, negative plates, paper type 1 (gloss-coated)
OF COM NE P2 F6o	ISO 12647-2, negative plates, paper type 2 (matte-coated)
OF COM NE P3 F6o	ISO 12647-2, negative plates, paper type 3 (light weight coated web)
OF COM NE P4 F6o	ISO 12647-2, negative plates, paper type 4 (uncoated white offset)
SC GC2 CO F3o	ISO 12647-5, gamut class 2, conventional UV or water-based air dried
lfra NP_40lcm_neg+CTP_05.00	Coldset offset (computer to plate)

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

### 7.4.3 Importing PDF/X Documents with PDI

Special rules apply when pages from an existing PDF document will be imported into a PDF/X-conforming output document (see Section 5.2, »Importing PDF Pages with PDI (PDF Import Library)«, page 133, for details on the PDF import library PDI). All imported documents must conform to an acceptable PDF/X conformance level according to Table 7.7. As a general rule, input documents conforming to the same PDF/X conformance level as the generated output document, or to an older version of the same level, are acceptable. In addition, certain other combinations are acceptable. If a certain PDF/X conformance level is configured in PDFlib and the imported documents adhere to one of the acceptable levels, the generated output is guaranteed to comply with the selected PDF/X conformance level. Imported documents which do not adhere to one of the acceptable PDF/X levels will be rejected.

Table 7.7 Acceptable PDF/X input levels for various PDF/X output levels; other combinations must be avoided.

PDF/X output level	PDF/X level of the imported document					
	PDF/X-1:2001	PDF/X-1a:2001	PDF/X-1a:2003	PDF/X-2:2003	PDF/X-3:2002	PDF/X-3:2003
PDF/X-1:2001	allowed	allowed				
PDF/X-1a:2001		allowed				
PDF/X-1a:2003		allowed	allowed			
PDF/X-2:2003		allowed	allowed	allowed	allowed	allowed
PDF/X-3:2002		allowed			allowed	
PDF/X-3:2003		allowed	allowed		allowed	allowed

If multiple PDF/X documents are imported, they must all have been prepared for the same output condition. While PDFlib can correct certain items, it is not intended to work as a full PDF/X validator or to enforce full PDF/X compatibility for imported docu-

ments. For example, PDFlib will not embed fonts which are missing from imported PDF pages, and does not apply any color correction to imported pages.

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

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

This statement will retrieve a string designating the PDF/X conformance level of the imported document if it conforms to an ISO PDF/X level, or *none* otherwise. The returned string can be used to set the PDF/X conformance level of the output document appropriately, using the *pdfx* option in *PDF\_begin\_document()*.

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

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

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

# 7.5 Tagged PDF

Tagged PDF is a certain kind of enhanced PDF which enables additional features in PDF viewers, such as accessibility support, text reflow, reliable text extraction and conversion to other document formats such as RTF or XML.

PDFlib supports Tagged PDF generation. However, Tagged PDF can only be created if the client provides information about the document’s internal structure, and obeys certain rules when generating PDF output.

*Note* PDFlib currently doesn’t support custom structure element types (i.e. only standard structure types as defined by PDF can be used), role maps, and structure element attributes.

## 7.5.1 Generating Tagged PDF with PDFlib

**Required operations.** Table 7.8 lists all operations required to generate Tagged PDF output. Not calling one of the required functions while in Tagged PDF mode will trigger an exception.

Table 7.8 Operations which must be applied for generating Tagged PDF

Item	PDFlib function and parameter requirements for Tagged PDF compatibility
Tagged PDF output	The tagged option in PDF_begin_document() must be set to true.
document language	The lang option in PDF_begin_document() must be set to specify the natural language of the document. It must initially be set for the document as a whole, but can later be overridden for individual items on an arbitrary structure level.
structure information	Structure information and artifacts must be identified as such. All content-generating API functions should be enclosed by PDF_begin_item() / PDF_end_item() pairs.

**Unicode-compatible text output.** When generating Tagged PDF, all text output must use fonts which are Unicode-compatible as detailed in Section 4.5.6, »Unicode-compatible Fonts«, page 95. This means that all used fonts must provide a mapping to Unicode. Non Unicode-compatible fonts are only allowed if alternate text is provided for the content via the *ActualText* or *Alt* options in *PDF\_begin\_item()*. PDFlib will throw an exception if text without proper Unicode mapping is used while generating Tagged PDF.

*Note* In some cases PDFlib will not be able to detect problems with wrongly encoded fonts, for example symbol fonts encoded as text fonts. Also, due to historical problems PostScript fonts with certain typographical variations (e.g., expert fonts) are likely to result in inaccessible output.

**Page content ordering.** The ordering of text, graphics, and image operators which define the contents of the page is referred to as the content stream ordering; the content ordering defined by the logical structure tree is referred to as logical ordering. Tagged PDF generation requires that the client obeys certain rules regarding content ordering.

The natural and recommended method is to sequentially generate all constituent parts of a structure element, and then move on to the next element. In technical terms, the structure tree should be created during a single depth-first traversal.

A different method which should be avoided is to output parts of the first element, switch to parts of the next element, return to the first, etc. In this method the structure tree is created in multiple traversals, where each traversal generates only parts of an element.



**Importing Pages with PDI.** Pages from Tagged PDF documents or other PDF documents containing structure information cannot be imported in Tagged PDF mode since the imported document structure would interfere with the generated structure.

Pages from unstructured documents can be imported, however. Note that they will be treated »as is« by Acrobat’s accessibility features unless they are tagged with appropriate *ActualText*.

**Artifacts.** Graphic or text objects which are not part of the author’s original content are called artifacts. Artifacts should be identified as such using the *Artifact* pseudo tag, and classified according to one of the following categories:

- ▶ *Pagination*: features such as running heads and page numbers
- ▶ *Layout*: typographic or design elements such as rules and table shadings
- ▶ *Page*: production aids, such as trim marks and color bars.

Although artifact identification is not strictly required, it is strongly recommended to aid text reflow and accessibility.

**Inline items.** PDF defines block-level structure elements (BLSE) and inline-level structure elements (ILSE) (see Table 8.54 for a precise definition). BLSEs may contain other BLSEs or actual content, while ILSEs always directly contain content. In addition, PDFlib makes the following distinction:

Table 7.9 Regular and inline items

	regular items	inline items
affected items	all grouping elements and BLSEs	all ILSEs and non-structural tags (pseudo tags)
regular/inline status can be changed	no	only for <i>ASpan</i> items
part of the document’s structure tree	yes	no
can cross page boundaries	yes	no
can be interrupted by other items	yes	no
can be suspended and activated	yes	no
can be nested to an arbitrary depth	yes	only with other inline items

The regular vs. inline decision for *ASpan* items is under client control via the *inline* option of *PDF\_begin\_item()*. Forcing an accessibility span to be regular (*inline=false*) is recommended, for example, when a paragraph which is split across several pages contains multiple languages. Alternatively, the item could be closed, and a new item started on the next page. Inline items must be closed on the page where they have been opened.

**Optional operations.** Table 7.10 lists all operations which are optional when generating Tagged PDF output. These features are not strictly required, but will enhance the quality of the generated Tagged PDF output and are therefore recommended.

Table 7.10 Operations which are optional for generating Tagged PDF

Item	Optional PDFlib function and parameter for Tagged PDF compatibility
hyphenation	Word breaks (separating words in two parts at the end of a line) should be presented using a soft hyphen character (U+00A0) as opposed to a hard hyphen (U+002D)
word boundaries	Words should be separated by space characters (U+0020) even if this would not strictly be required for positioning. The autospace parameter can be used for automatically generating space characters after each call to one of the show functions.
artifacts	In order to distinguish real content from page artifacts, artifacts should be identified as such using PDF_begin_item() with tag=Artifact.

**Prohibited operations.** Table 7.11 lists all operations which are prohibited when generating Tagged PDF output. Calling one of the prohibited functions while in Tagged PDF mode will trigger an exception.

Table 7.11 Operations which must be avoided when generating Tagged PDF

Item	PDFlib functions and parameters to be avoided for Tagged PDF compatibility
non-Unicode compatible fonts	Fonts which are not Unicode-compatible according to Section 4.5.6, »Unicode-compatible Fonts«, page 95, must be avoided.
PDF import	Pages from PDF documents which contain structure information (in particular: Tagged PDF documents) must not be imported.

## 7.5.2 Creating Tagged PDF with direct Text Output and Textflows

**Minimal Tagged PDF sample.** The following sample code creates a very simplistic Tagged PDF document. Its structure tree contains only a single *P* element. The code uses the *autospace* feature to automatically generate space characters between fragments of text:

```
if (PDF_begin_document(p, "hello-tagged.pdf", 0, "tagged=true lang=en") == -1)
{
    printf("Error: %s\n", PDF_get_errmsg(p));
    return(2);
}

/* automatically create spaces between chunks of text */
PDF_set_parameter(p, "autospace", "true");

/* open the first structure element as a child of the document structure root (=0) */
id = PDF_begin_item(p, "P", "Title = {Simple Paragraph}");

PDF_begin_page_ext(p, 0, 0, "width=a4.width height=a4.height");
font = PDF_load_font(p, "Helvetica-Bold", 0, "host", "");

PDF_setfont(p, font, 24);
PDF_show_xy(p, "Hello, Tagged PDF!", 50, 700);
PDF_continue_text(p, "This PDF has a very simple");
PDF_continue_text(p, "document structure.");

PDF_end_page_ext(p, "");
PDF_end_item(p, id);
PDF_end_document(p, "");
```

**Generating Tagged PDF with textflows.** The textflow feature (see Section 4.9, »Multi-Line Textflows«, page 111) offers powerful features for text formatting. Since individual text fragments are no longer under client control, but will be formatted automatically by PDFlib, special care must be taken when generating Tagged PDF with textflows:

- ▶ Textflows can not contain individual structure elements, but a textflow may be contained in a structure element.
- ▶ All parts of a textflow (all calls to *PDF\_fit\_textflow()* with a specific textflow handle) should be contained in a single structure element.
- ▶ Since the parts of a textflow could be spread over several pages which could contain other structure items, attention should be paid to choosing the proper parent item (rather than using a parent parameter of -1, which may point to the wrong parent element).

### 7.5.3 Activating Items for complex Layouts

In order to facilitate the creation of structure information with complex non-linear page layouts PDFlib supports a feature called item activation. It can be used to activate a previously created structure element in situations where the developer must keep track of multiple structure branches, where each branch could span one or more pages. Typical situations which will benefit from this technique are the following:

- ▶ multiple columns on a page
- ▶ insertions which interrupt the main text, such as summaries or inserts
- ▶ tables and illustrations which are placed between columns.

The activation feature allows an improved method of generating page content in such situations by switching back and forth between logical branches. This is much more efficient than completing each branch one after the other. Let's illustrate the activation feature using the page layout shown in Figure 7.1. It contains two main text columns, interrupted by a table and an inserted annotation in a box (with dark background) as well as header and footer.

**Generating page contents in logical order.** From the logical structure point of view the page content should be created in the following order: left column, right column (on the lower right part of the page), table, insert, header and footer. The following pseudo code implements this ordering:

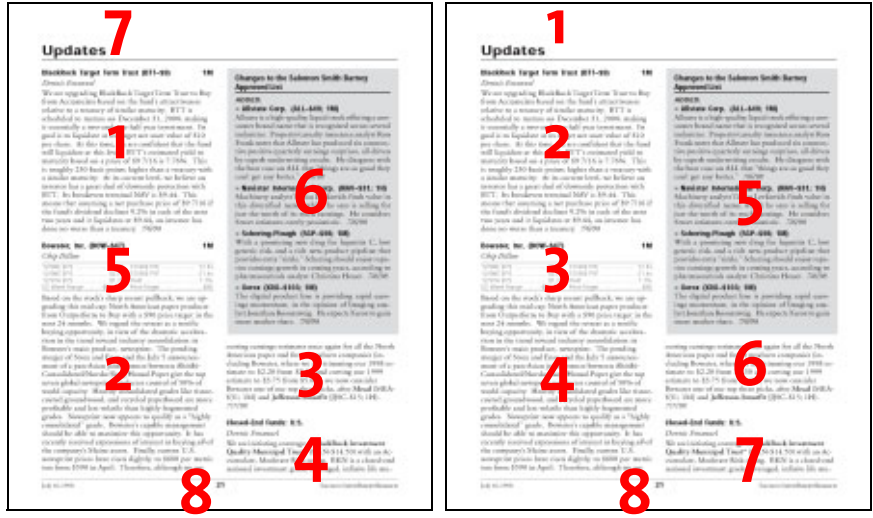
```
/* create page layout in logical structure order */

id_art = PDF_begin_item(p, "Art", "Title = Article");

id_sect1 = PDF_begin_item(p, "Sect", "Title = {First Section}");
/* 1 create top part of left column */
PDF_set_text_pos(p, x1_left, y1_left_top);
...
/* 2 create bottom part of left column */
PDF_set_text_pos(p, x1_left, y1_left_bottom);
...
/* 3 create top part of right column */
PDF_set_text_pos(p, x1_right, y1_right_top);
...
PDF_end_item(p, id_sect1);

id_sect2 = PDF_begin_item(p, "Sect", "Title = {Second Section}");
```

Fig. 7.1  
Creating a complex  
page layout in logical  
structure order (left)  
and in visual order  
(right). The right vari-  
ant uses item activa-  
tion for the first sec-  
tion before continuing  
fragments 4 and 6.



```

/* 4 create bottom part of right column */
PDF_set_text_pos(p, x2_right, y2_right);
...
/* second section may be continued on next page(s) */
PDF_end_item(p, id Sect2);

sprintf(optlist, "Title=Table parent=%d", id_art);
id_table = PDF_begin_item(p, "Table", optlist);
/* 5 create table structure and content */
PDF_set_text_pos(p, x_start_table, y_start_table);
...
PDF_end_item(p, id_table);

sprintf(optlist, "Title=Insert parent=%d", id_art);
id_insert = PDF_begin_item(p, "P", optlist);
/* 6 create insert structure and content */
PDF_set_text_pos(p, x_start_table, y_start_table);
...
PDF_end_item(p, id_insert);

id_artifact = PDF_begin_item(p, "Artifact", "");
/* 7+8 create header and footer */
PDF_set_text_pos(p, x_header, y_header);
...
PDF_set_text_pos(p, x_footer, y_footer);
...
PDF_end_item(p, id_artifact);

/* article may be continued on next page(s) */
...
PDF_end_item(p, id_art);

```

**Generating page contents in visual order.** The »logical order« approach forces the creator to construct the page contents in logical order even if it might be easier to create it in visual order: header, left column upper part, table, left column lower part, insert,

right column, footer. Using *PDF\_activate\_item()* this ordering can be implemented as follows:

```
/* create page layout in visual order */

id_header = PDF_begin_item(p, "Artifact", "");
/* 1 create header */
PDF_set_text_pos(p, x_header, y_header);
...
PDF_end_item(p, id_header);

id_art = PDF_begin_item(p, "Art", "Title = Article");

id_sect1 = PDF_begin_item(p, "Sect", "Title = {First Section}");
/* 2 create top part of left column */
PDF_set_text_pos(p, x1_left, y1_left_top);
...

sprintf(optlist, "Title=Table parent=%d", id_art);
id_table = PDF_begin_item(p, "Table", optlist);
/* 3 create table structure and content */
PDF_set_text_pos(p, x_start_table, y_start_table);
...
PDF_end_item(p, id_table);

/* continue with first section */
PDF_activate_item(p, id_sect1);
/* 4 create bottom part of left column */
PDF_set_text_pos(p, x1_left, y1_left_bottom);
...

sprintf(optlist, "Title=Insert parent=%d", id_art);
id_insert = PDF_begin_item(p, "P", optlist);
/* 5 create insert structure and content */
PDF_set_text_pos(p, x_start_table, y_start_table);
...
PDF_end_item(p, id_insert);

/* still more contents for first section */
PDF_activate_item(p, id_sect1);
/* 6 create top part of right column */
PDF_set_text_pos(p, x1_right, y1_right_top);
...
PDF_end_item(p, id_sect1);

id_sect2 = PDF_begin_item(p, "Sect", "Title = {Second Section}");
/* 7 create bottom part of right column */
PDF_set_text_pos(p, x2_right, y2_right);
...
/* second section may be continued on next page(s) */
PDF_end_item(p, id_sect2);

id_footer = PDF_begin_item(p, "Artifact", "");
/* 8 create footer */
PDF_set_text_pos(p, x_footer, y_footer);
...
PDF_end_item(p, id_footer);
```

```
/* article may be continued on next page(s) */  
...  
PDF_end_item(p, id_art);
```

With this ordering of structure elements the main text (which spans one and a half columns) is interrupted twice for the table and the insert. Therefore it must also be activated twice using *PDF\_activate\_item()*.

The same technique can be applied if the content spans multiple pages. For example, the header or other inserts could be created first, and then the main page content element is activated again.

## 7.5.1 Using Tagged PDF in Acrobat

This section mentions observations which we made while testing Tagged PDF output in Adobe Acrobat 6.0. They are mostly related to bugs or inconsistent behavior in Acrobat. A workaround is provided in cases where we found one.

**Acrobat's Reflow Feature.** Acrobat allows Tagged PDF documents to reflow, i.e. to adjust the page contents to the current window size. While testing Tagged PDF we made several observations regarding the reflow feature in Acrobat:

- ▶ The order of content on the page should follow the desired reflow order.
- ▶ Symbol (non-Unicode fonts) can cause Acrobat's reflow feature to crash. For this reason it is recommended to put the text in a *Figure* element.
- ▶ BLSEs may contain both structure children and direct content elements. In order for the reflow feature (as well as Accessibility checker and Read Aloud) to work it is recommended to put the direct elements before the first child elements.
- ▶ The *BBox* option should be provided for tables and illustrations. The *BBox* should be exact; however, for tables only the lower left corner has to be set exactly. As an alternative to supplying a *BBox* entry, graphics could also be created within a BLSE tag, such as *P*, *H*, etc. However, vector graphics will not be displayed when Reflow is active. If the client does not provide the *BBox* option (and relies on automatic *BBox* generation instead) all table graphics, such as cell borders, should be drawn outside the table element.
- ▶ Table elements should only contain table-related elements (*TR*, *TD*, *TH*, *THHead*, *TBody*, etc.) as child elements, but not any others. For example, using a *Caption* element within a table could result in reflow problems, although it would be correct Tagged PDF.
- ▶ Content covered by the *Private* tag will not be exported to other formats. However, they are subject to reflow and Read Aloud, and illustrations within the *Private* tag must therefore have alternate text.
- ▶ Imported images should be covered by a *Figure* element; imported PDF pages should be covered by a *Form* item. Item type *Formula* should be avoided in order to avoid problems with reflow.
- ▶ Reflow seems to have problems with PDF documents generated with the *topdown* option.
- ▶ Structure items with mixed types of children (i.e., both page content sequences and non-inline structure elements) should be avoided since otherwise Reflow could fail.
- ▶ If an activated item contains only content, but no structure children, Reflow could fail, especially if the item is activated on another page. This problem can be avoided by wrapping the activated item with a non-inline *Span* tag.

**Acrobat's Accessibility Checker.** Acrobat's accessibility checker can be used to determine the suitability of Tagged PDF documents for consumption with assisting technology such as a screenreader.

- ▶ Elements containing an imported image should use the *Alt* property. The *ActualText* property could cause the accessibility checker to crash. Another reason to prefer *Alt* over *ActualText* is that the Read Aloud feature will catch the real text.
- ▶ If a *Form* tag covering an imported PDF page is the very first item on the page it can cause problems with the accessibility checker.
- ▶ If the *Lbl* tag is set within the *TOCI* tag (as actually described in the PDF reference) the Accessibility Checker will warn that the *Lbl* tag is not set within an *LI* tag.

**Export to other formats with Acrobat.** Tagged PDF can significantly improve the result of exporting PDF documents to other formats in Acrobat.

- ▶ If an imported PDF page has the *Form* tag, the text provided with the *ActualText* option will be exported to other formats in Acrobat, while the text provided with the *Alt* tag will be ignored. However, the Read Aloud feature works for both options.
- ▶ Elements containing an imported image should use the *Alt* property instead of *ActualText* so that the Export feature will catch the real text.
- ▶ The content of a *NonStruct* tag will not be exported to HTML 4.01 CSS 1.0 (but it will be used for HTML 3.2 export).
- ▶ Alternate text should be supplied for ILSEs (such as *Code*, *Quote*, or *Reference*). If the *Alt* option is used, Read Aloud will read the provided text, but the real content will be exported to other formats. If the *ActualText* option is used, the provided text will be used both for reading and exporting.

**Acrobat's Read Aloud Feature.** Tagged PDF will enhance Acrobat's capability to read text aloud.

- ▶ When supplying *Alt* or *ActualText* it is useful to include a space character at the beginning. This allows the Read Aloud feature to distinguish the text from the preceding sentence. For the same reason, including a ' ' character at the end may also be useful. Otherwise Read Aloud will try to read the last word of the preceding sentence in combination with the first word of the alternate text.





# 8 API Reference for PDFlib, PDI, and PPS

The API reference documents all supported functions of PDFlib, PDI (PDF Import) and PPS (PDFlib Personalization Server).

## 8.1 Data Types and Naming Conventions

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

Table 8.1 details the use of the PDF document type and the string type in all language bindings. The data types *integer*, *long*, and *double* are not mentioned since there is an obvious mapping of these types in all bindings. Please refer to the respective language section and the examples in Chapter 2 for more language-specific details.

Table 8.1 Data types in the language bindings

language binding	p parameter?	PDF_ prefix?	string data type	binary data type
C (also used in this API reference)	yes	yes	const char * <sup>1</sup>	const char *
C++	no	no	string <sup>2</sup>	char *
Cobol <sup>3</sup>	yes	no <sup>4</sup>	STRING	STRING
Java	no	no	String	byte[ ]
Perl	yes	yes	string	string
PHP	yes	yes	string	string
Python	yes	yes	string	string
RPG	yes	yes	string, but must add x'oo'	data
Tcl	yes	yes	string	byte array

1. C language NULL string values and empty strings are considered equivalent.  
2. NULL string values must not be used in the C++ binding.  
3. See Section 2.2.1, »Special Considerations for Cobol«, page 20 for more information on Cobol data types.  
4. Cobol programs must use abbreviated names for the PDFlib functions.

**Unicode Strings.** PDFlib accepts Unicode strings in all relevant areas and supports various formats and settings related to Unicode. Please review Section 4.5.2, »Content Strings, Hypertext Strings, and Name Strings«, page 90, for details, and take care of the following string types which are used in this chapter:

- Content strings
- Hypertext strings
- Name strings

**Naming conventions for PDFlib Functions.** In the C binding, all PDFlib functions live in a global namespace and carry the common *PDF\_* prefix in their name in order to minimize namespace pollution. In contrast, several language bindings hide the PDF document parameter in an object created by the language wrapper. For these bindings, the

function name given in this API reference must be changed by omitting the *PDF\_* prefix and the *PDF \** parameter used as first argument. For example, the C-like API description

```
PDF *p;  
PDF_begin_document(PDF *p, const char *filename, const char *optlist);
```

translates to the following when the function is used from Java:

```
pdflib p;  
p.begin_document(String filename, String optlist);
```

# 8.2 General Functions

## 8.2.1 Setup

Table 8.2 lists relevant parameters and values for this section.

Table 8.2 Parameters and values for the setup functions

function	key	explanation
set_parameter	compatibility	Deprecated, use the compatibility option for PDF_begin_document( ).
set_parameter	pdfx	Deprecated, use the pdfx option for PDF_begin_document( ).
set_parameter	flush	Deprecated, use the flush option for PDF_begin_document_callback( ).
set_parameter	SearchPath	(Not supported on MVS) Relative or absolute path name of a directory containing files to be read. The SearchPath can be set multiply; the entries will be accumulated and used in least-recently-set order (see Section 3.1.6, »Resource Configuration and File Searching«, page 51). Scope: any
set_parameter	resourcefile	Relative or absolute file name of the PDFlib UPR resource file. The resource file will be loaded immediately. Existing resources will be kept; their values will be overridden by new ones if they are set again. Scope: any
set_parameter	asciifile	(Only supported on iSeries and zSeries). Expect text files (PFA, AFM, UPR, encodings) in ASCII encoding. Default: true on iSeries; false on zSeries. Scope: any
set_parameter	license	Set the license key for PDFlib, PDFlib+PDI, or PPS. The key can be set (even multiply to accumulate keys) before the first call to PDF_begin_document( ). Scope: object
set_parameter	licensefile	Set the name of a file containing the license key. The license file can only be set once before the first call to PDF_begin_document( ). Scope: object.
set_value	compress	Set the compression level. This parameter does not affect image data handled in pass-through mode. Default: 6. Scope: page, document 0 no compression 1 best speed 9 best compression
get_value	major, minor revision	Return the major, minor, or revision number of PDFlib, respectively. Scope: any, null <sup>1</sup> .
get_parameter	version	Return the full PDFlib version string in the format <major>.<minor>.<revision>, possibly suffixed with additional qualifiers such as beta, rc, etc. Scope: any, null <sup>1</sup> .
get_parameter	scope	Return the name of the current scope (see Table 3.1). Scope: any
set_parameter	trace	If true, all API function calls will be logged to a trace file. The contents of the trace file may be useful for debugging purposes, or may be requested by PDFlib support. Scope: any. Default: false
set_parameter	tracefile	Set trace file name. Scope: any, but before enabling tracing. Default: PDFlib.trace.
set_parameter	tracemsg	If tracing is enabled, the supplied message text will be written to the trace file in addition to API calls. This may be useful for debugging client code. Scope: any

1. May be called with a PDF \* argument of NULL or 0.

```
void PDF_boot(void)
void PDF_shutdown(void)
```

Boot and shut down PDFlib, respectively.

Scope null

Bindings C: Recommended for the C language binding, although currently not required.

Other bindings: For all other language bindings booting and shutting down is accomplished automatically by the wrapper code, and these functions are not available.

---

**PDF \*PDF\_new(void)**

---

Create a new PDFlib object with default settings.

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

**Returns** A handle to a PDFlib object which is to be used in subsequent PDFlib calls. If this function doesn't succeed due to unavailable memory it will return NULL (in C) or throw an exception.

**Scope** *null*; this function starts object scope, and must always be paired with a matching *PDF\_delete()* call.

**Bindings** The data type used for the opaque PDFlib object handle varies among language bindings. This doesn't really affect PDFlib clients, since all they have to do is pass the PDF handle as the first argument to all functions.

C: In order to load the PDFlib DLL dynamically at runtime use *PDF\_new\_dl()* instead (see Section 2.4.3, »Using PDFlib as a DLL loaded at Runtime«, page 25). *PDF\_new\_dl()* will return a pointer to a *PDFlib\_api* structure filled with pointers to all PDFlib API functions. If the DLL cannot be loaded, or a mismatch of major or minor version number is detected, NULL will be returned.

C++: this function is not available since it is hidden in the PDF constructor.

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

PHP: this function is automatically called by the wrapper code of the object-oriented PHP 5 interface when an object is created.

---

**PDF \*PDF\_new2(void (\*errorhandler)(PDF \*p, int errortype, const char \*msg),  
void\* (\*allocproc)(PDF \*p, size\_t size, const char \*caller),  
void\* (\*reallocproc)(PDF \*p, void \*mem, size\_t size, const char \*caller),  
void (\*freeproc)(PDF \*p, void \*mem),  
void \*opaque)**

---

Create a new PDFlib object with client-supplied error handling and memory allocation routines.

**errorhandler** Pointer to a user-supplied error-handling function. The error handler will be ignored in *PDF\_TRY/PDF\_CATCH* blocks.

**allocproc** Pointer to a user-supplied memory allocation function.

**reallocproc** Pointer to a user-supplied memory reallocation function.

**freeproc** Pointer to a user-supplied free function.

**opaque** Pointer to some user data which may be retrieved later with *PDF\_get\_opaque()*.

- Returns** A handle to a PDFlib object which is to be used in subsequent PDFlib calls. If this function doesn't succeed due to unavailable memory it will return NULL (in C) or throw an exception.
- Details** This function creates a new PDFlib object with client-supplied error handling and memory allocation routines. Unlike *PDF\_new()*, the caller may optionally supply own procedures for error handling and memory allocation. The function pointers for the error handler, the memory procedures, or both may be NULL. PDFlib will use default routines in these cases. Either all three memory routines must be provided, or none.
- Scope** *null*; this function starts *object* scope, and must always be paired with a matching *PDF\_delete()* call. No other PDFlib function with the same PDFlib object must be called after calling this function.
- Bindings** C++: this function is indirectly available via the PDF constructor. Not all function arguments must be given since default values of NULL are supplied. All supplied functions must be »C« style functions, not C++ methods.

---

**void PDF\_delete(PDF \*p)**

---

Delete a PDFlib object and free all internal resources.

- Details** This function deletes a PDF object and frees all document-related PDFlib-internal resources. Although not necessarily required for single-document generation, deleting the PDF object is heavily recommended for all server applications when they are done producing PDF. This function must only be called once for a given PDF object. *PDF\_delete()* should also be called for cleanup when an exception occurred. *PDF\_delete()* itself is guaranteed to not throw any exception. If more than one PDF document will be generated it is not necessary to call *PDF\_delete()* after each document, but only when the complete sequence of PDF documents is done.
- Scope** *any*; this function starts *null* scope, i.e., no more API function calls are allowed.
- Bindings** C: If the PDFlib DLL has been loaded dynamically at runtime with *PDF\_new\_dl()*, use *PDF\_delete\_dl()* to delete the PDFlib object.
- C++: this function is indirectly available via the PDF destructor.
- Java: this function is automatically called by the wrapper code. However, it can explicitly be called from client code in order to overcome shortcomings in Java's finalizer system.
- PHP: this function will automatically be called for the object-oriented PHP 5 interface when the PDFlib object goes out of scope.

8.2.2 Document and Page

Table 8.3 lists relevant parameters and values for this section.

Table 8.3 Parameters and values for the document and page functions

function	key	explanation
set_parameter	openwarning	Deprecated, use <i>PDF_get_errmsg()</i> to find out the reason of failed attempts at opening a document.
set_value	pagewidth pageheight	Deprecated, use the width and height parameters for <i>PDF_begin_page_ext()</i> or the mediabox option for <i>PDF_begin_page_ext()</i> or <i>PDF_end_page_ext()</i> .

Table 8.3 Parameters and values for the document and page functions

function	key	explanation
set_parameter	topdown	If true, the origin of the coordinate system at the beginning of a page, pattern, or template will be assumed in the top left corner of the page, and y coordinates will increase downwards; otherwise the default coordinate system will be used (see Section 3.2.1, »Coordinate Systems«, page 57). Scope: document. Default: false
get_value	pagewidth pageheight	Get the page size of the current page (dimensions of the MediaBox). Scope: page, path
set_value	ArtBox BleedBox CropBox TrimBox	Deprecated, use the artbox, bleedbox, cropbox, and trimbox options for PDF_begin_page_ext() or PDF_end_page_ext().
set_parameter	userpassword master- password permissions	Deprecated, use the userpassword/masterpassword/permissions options for PDF_begin_document().

---

```
int PDF_begin_document(PDF *p, const char *filename, int len, const char *optlist)
```

---

---

```
int PDF_begin_document_callback(PDF *p, const char *filename, int len,  
size_t (*writeproc) (PDF *p, void *data, size_t size), const char *optlist)
```

---

Create a new PDF file subject to various options.

**filename** (Name string, but Unicode file names are only supported on Windows) Absolute or relative name of the PDF output file to be generated. If *filename* is empty, the PDF document will be generated in memory instead of on file, and the generated PDF data must be fetched by the client with the *PDF\_get\_buffer()* function. The special file name »-« can be used for generating PDF on the *stdout* channel. On Windows it is OK to use UNC paths or mapped network drives.

**len** (Only for the C binding.) Length of *filename* (in bytes) for UTF-16 strings. If *len* = 0 a null-terminated string must be provided.

**writeproc** (Only for C and C++) C callback function which will be called by PDFlib in order to submit (portions of) the generated PDF data.

**optlist** An option list specifying document options according to Table 8.4 or Table 8.5. Options specified in *PDF\_end\_document()* have precedence over identical options specified in *PDF\_begin\_document()*.

**Returns** -1 (in PHP: 0) on error, and 1 otherwise. If *filename* is empty this function will always succeed, and never return the -1 (in PHP: 0) error value.

**Details** This function creates a new PDF file using the supplied *filename*. PDFlib will attempt to open a file with the given name, and close the file when the PDF document is finished. *PDF\_begin\_document\_callback()* opens a new PDF document in memory, without writing to a disk file. The callback function supplied as *writeproc* must return the number of bytes written. If the return value doesn't match the *size* argument supplied by PDFlib, an exception will be thrown. The frequency of *writeproc* calls is configurable with the *flush* option.

**Scope** *object*; this function starts *document* scope if the file could successfully be opened, and must always be paired with a matching `PDF_end_document()` call.

**Bindings** C, C++, Java, JavaScript: take care of properly escaping the backslash path separator. For example, the following denotes a file on a network drive: `\\\\malik\\rp\\foo.pdf`. `PDF_begin_document_callback()` is only available in C and C++. The supplied *writeproc* must be a C-style function, not a C++ method.

Table 8.4 Document options for `PDF_begin_document()` and `PDF_begin_document_callback()`

option	type	description
compatibility	keyword	Set the document's PDF version to one of the strings »1.3«, »1.4«, or »1.5« for Acrobat 4, 5, or 6. See Section 7.1, »Acrobat and PDF Versions«, page 167, for details. This option will be ignored if the <code>pdfx</code> parameter is used. Default: »1.5«
flush	keyword	(Only for <code>PDF_begin_document_callback()</code> ) Set the flushing strategy; See Section 3.1.7, »Generating PDF Documents in Memory«, page 54, for details (default: page): <div> <div>none</div> <div>flush only once at the end of the document</div> <div>page</div> <div>flush at the end of each page</div> <div>content</div> <div>flush after all fonts, images, file attachments, and pages</div> <div>heavy</div> <div>always flush when the internal 64 KB document buffer is full</div> </div>
groups	list of strings	Define the names and ordering of the page groups used in the document.
inmemory	boolean	(Only for <code>PDF_begin_document()</code> ) If true and the <code>linearize</code> option is true as well, PDFlib will not create any temporary files for linearization, but will process the file in memory. This can result in tremendous performance gains on some systems (especially MVS), but requires memory twice the size of the document. If false, a temporary file will be created for linearization. Default: false
lang	string	(Required if <code>tagged=true</code> ) Set the natural language of the document as a two-character ISO 639 language code (examples: DE, EN, FR, JA), optionally followed by a hyphen and a two-character ISO 3166 country code (examples: EN-US, EN-GB, ES-MX). Case is not significant.  The language specification can be overridden for individual items on all levels of the structure tree, but must be set initially for the document as a whole.
linearize	boolean	(Only for <code>PDF_begin_document()</code> ) If true, the output document will be linearized (see Section 7.3, »Web-Optimized (Linearized) PDF«, page 170). On MVS systems this option cannot be combined with in-core generation (i.e. an empty filename). Default: false
master-password	string	The master password for the document. If it is empty no master password will be applied. Default: empty
permissions	keyword list	The access permission list for the output document. It contains any number of the <code>noprint</code> , <code>nomodify</code> , <code>nocopy</code> , <code>noannots</code> , <code>noassemble</code> , <code>noforms</code> , <code>noaccessible</code> , <code>nohiresprint</code> , and <code>plainmetadata</code> keywords (see Table 7.3). Default: empty
pdfx	keyword	Set the PDF/X conformance level to one of »PDF/X-1:2001«, »PDF/X-1a:2001«, »PDF/X-1a:2003«, »PDF/X-2:2003«, »PDF/X-3:2002«, »PDF/X-3:2003«, or »none« (see Section 7.4, »PDF/X«, page 171). Default: none
recordsize	integer	(MVS only) The record size of the output file. Default: 0 (unblocked output)
tagged	boolean	(PDF 1.4 and above) If true, generate Tagged PDF output. Proper structure information must be provided by the client in Tagged PDF mode (see Section 8.10, »Structure Functions for Tagged PDF«, page 278). Default: false
tempdirname	string	(Only for <code>PDF_begin_document()</code> ) Name of a directory where temporary files needed for linearization will be created. If empty, PDFlib will generate temporary files in the current directory. This option will be ignored if the <code>tempfilename</code> option has been supplied. Default: empty

Table 8.4 Document options for `PDF_begin_document()` and `PDF_begin_document_callback()`

option	type	description
tempfilenames	list of two strings	(Only on MVS and for <code>PDF_begin_document()</code> ) Full file names for two temporary files needed for PDFlib's internal processing. If empty, PDFlib will generate unique temporary file names. The user is responsible for deleting the temporary files after <code>PDF_end_document()</code> . If this option is supplied the filename parameter must not be empty. Default: empty
user-password	string	The user password for the document. If it is empty no user password will be applied. Default: empty

**void PDF\_end\_document(PDF \*p, const char \*optlist)**

Close the generated PDF file and apply various options.

**optlist** An option list specifying document options according to Table 8.5. Options specified in `PDF_end_document()` have precedence over identical options specified in `PDF_begin_document()`.

**Details** This function finishes the generated PDF document, frees all document-related resources, and closes the output file if the PDF document has been opened with `PDF_begin_document()`. This function must be called when the client is done generating pages, regardless of the method used to open the PDF document.

When the document was generated in memory (as opposed to on file), the document buffer will still be kept after this function is called (so that it can be fetched with `PDF_get_buffer()`), and will be freed in the next call to `PDF_begin_document()`, or when the PDFlib object goes out of scope in `PDF_delete()`.

**Scope** *document*; this function terminates *document* scope, and must always be paired with a matching call to one of the `PDF_begin_document()` or `PDF_begin_document_callback()` functions.

Table 8.5 Document options for `PDF_begin_document()` and `PDF_end_document()`

option	type	description
action	action list	(PDF 1.4 except <i>open</i> , which is available in PDF 1.3) List of document actions for one or more of the following events (default: empty list): <i>open</i> Actions to be performed when the document is opened. <i>didprint</i> JavaScript actions to be performed after printing the document. <i>didsave</i> JavaScript actions to be performed after saving the document. <i>willclose</i> JavaScript actions to be performed before closing the document. <i>willprint</i> JavaScript actions to be performed before printing the document. <i>willsave</i> JavaScript actions to be performed before saving the document.
destination	option list	An option list specifying the document open action according to Table 8.47. The open action will be dominant over this option. Default: the handle supplied in the open action, or {type fitwindow page o} if no open action was supplied.
labels	list of option lists	A list containing one or more option lists according to Table 8.7 specifying symbolic page names. The page name will be displayed as a page label (instead of the page number) in Acrobat's status line. The combination of style/prefix/start values must be unique within a document. Default: none



Table 8.5 Document options for PDF\_begin\_document() and PDF\_end\_document()

option	type	description
openmode	keyword	Set the appearance when the document is opened. Default: bookmarks if the document contains any bookmarks, otherwise none:
		none                   Open with no additional panel visible.
		bookmarks           Open with the bookmark panel visible.
		thumbnails          Open with the thumbnail panel visible
		fullscreen          Open in fullscreen mode (does not work in the browser).
		layers               (PDF 1.5) Open with the layer panel visible.
pagelayout	keyword	The page layout to be used when the document is opened (default: singlepage):
		singlepage          Display one page at a time.
		onecolumn           Displays the pages continously in one column.
		twocolumnleft      Display the pages in two columns, odd pages on the left.
		twocolumnright     Display the pages in two columns, odd pages on the right.
uri	string	Set the document's base URL. This is useful when a document with relative Web links to other documents is moved to a different location. Setting the base URL to the »old« location makes sure that relative links will still work. Default: none
viewer-preferences	option list	An option list specifying various viewer preferences according to Table 8.6. Default: empty
metadata	option list	(PDF 1.4) Supply metadata for the document. PDFlib will not synchronize meta-data and document info fields. The option list may contain the following options:
		inputencoding (keyword)           The encoding to interpret the supplied data. Default: unicode
		inputformat (keyword)           The format of the supplied data. Default: utf8 (ebcdicutf8 on EBCDIC-based systems), but bytes if inputencoding is an 8-bit encoding
		filename           (name string, required) The name of a disk-based or virtual file containing the metadata. The file must contain well-formed XMP metadata which will be copied to the output uncompressed. PDFlib will automatically generate the XDP packet header and trailer.
		outputformat (keyword)           The format in which the data will be written to the PDF output (the output encoding will always be unicode). Possible values are utf8, utf16be, utf16le. Default: utf8 if inputformat=bytes, otherwise inputformat

Table 8.6 Suboptions for the viewerpreference option in PDF\_begin\_document() and PDF\_end\_document()

option	type	description
centerwindow	boolean	Specifies whether to position the document's window in the center of the screen . Default: false
direction	keyword	The reading order of the document, which affects the scroll ordering in double-page view. (default l2r):
		l2r                   Left to right
		r2l                   Right to left (including vertical writing systems)
displaydoctitle	boolean	Specifies whether to display the Title document info field in Acrobat's title bar (true) or the file name (false). Default: false
fitwindow	boolean	Specifies whether to resize the document's window to the size of the first page . Default: false
hidemenubar	boolean	Specifies whether to hide Acrobat's menu bar. Default: false
hidetoolbar	boolean	Specifies whether to hide Acrobat's tool bars. Acrobat ignores this setting when viewing PDFs in a browser. Default: false
hidewindowui	boolean	Specifies whether to hide Acrobat's window controls. Default: false

Table 8.6 Suboptions for the viewerpreference option in PDF\_begin\_document() and PDF\_end\_document()

option	type	description
nonfullscreen- pagemode	keyword	(Only relevant if the openmode option is set to fullscreen) Specifies how to display the document on exiting full-screen mode (default: none): bookmarks      display page and bookmark pane thumbnails      display page and thumbnail pane layers      display page and layer pane none      display page only
viewarea viewclip printarea printclip	keyword	The type of the page boundary box representing the area of a page to be displayed or clipped when viewing the document on screen or printing it. Acrobat ignores this setting, but it may be useful for other applications (default: crop): art      Use the ArtBox bleed      Use the BleedBox crop      Use the CropBox media      Use the MediaBox trim      Use the TrimBox  PDF/X: values other than media or bleed are not allowed.

Table 8.7 Suboptions for the labels option in PDF\_begin/end\_document() and label option in PDF\_begin/end\_page\_ext()

option	type	description
group	string	(Only for PDF_begin_document(); required if the document uses page groups, but not allowed otherwise.) The label will be applied to all pages in the specified group and all pages in all subsequent groups until a new label is applied.
hypertext- encoding	keyword	Specifies the encoding for the prefix option (see Section 4.5.4, »String Handling in non-Unicode-capable Languages«, page 91). An empty string is equivalent to unicode. Default: value of the global hypertextencoding parameter.
pagenumber	integer	(Only for PDF_end_document(); required if the document does not use page groups, but not allowed otherwise) The label will be applied to the specified page and subsequent pages until a new label is applied.
prefix	hypertext string	The label prefix for all labels in the range. Default: none
start	integer >= 1	Numeric value for the first label in the range. Subsequent pages in the range will be numbered sequentially starting with this value. Default: 1
style	keyword	The numbering style to be used (default: none): none      no page number; labels will only consist of the prefix. D      decimal arabic numerals (1, 2, 3, ...) R      uppercase roman numerals (I, II, III, ...) r      lowercase roman numerals (i, ii, iii, ...) A      uppercase letters (A, B, C, ..., AA, BB, CC, ...) a      lowercase letters (a, b, c, ..., aa, bb, cc, ...)

---

**void PDF\_open\_mem(PDF \*p, size\_t (\*writeproc) (PDF \*p, void \*data, size\_t size))**

---

Deprecated, use *PDF\_begin\_document\_callback()*.

---

**const char \* PDF\_get\_buffer(PDF \*p, long \*size)**

---

Get the contents of the PDF output buffer.

**size** (C language binding only) C-style pointer to a memory location where the length of the returned data in bytes will be stored.

*Returns* A buffer full of binary PDF data for consumption by the client. It returns a language-specific data type for binary data according to Table 8.1. The returned buffer must be used by the client before calling any other PDFlib function.

*Details* Fetch the full or partial buffer containing the generated PDF data. If this function is called between page descriptions, it will return the PDF data generated so far. If generating PDF into memory, this function must at least be called after *PDF\_end\_document()*, and will return the remainder of the PDF document. It can be called earlier to fetch partial document data. If there is only a single call to this function which happens after *PDF\_end\_document()* the returned buffer is guaranteed to contain the complete PDF document in a contiguous buffer.

Since PDF output contains binary characters, client software must be prepared to accept non-printable characters including null values.

*Scope* *object, document* (in other words: after *PDF\_end\_page\_ext()* and before *PDF\_begin\_page\_ext()*, or after *PDF\_end\_document()* and before *PDF\_delete()*). This function can only be used if an empty filename has been supplied to *PDF\_begin\_document()*.

If the *linearize* option in *PDF\_begin\_document()* has been set to *true*, the scope is restricted to *document*, i.e. this function can only be called after *PDF\_end\_document()*.

*Bindings* C and C++: the *size* parameter is only used for C and C++ clients.

Other bindings: an object of appropriate length will be returned, and the *size* parameter must be omitted.

---

**void PDF\_begin\_page\_ext(PDF \*p, double width, double height, const char \*optlist)**

---

Add a new page to the document, and specify various options.

**width, height** The *width* and *height* parameters are the dimensions of the new page in points. They can be overridden by the options with the same name (the dummy value o can be used for the parameters in this case). A list of commonly used page formats can be found in Table 3.4. See also Table 8.9 for more details (options width and height).

**optlist** An option list according to Table 8.8 and Table 8.9. These options have lower priority than identical options specified in *PDF\_end\_page\_ext()*.

*Details* This function will reset all text, graphics, and color state parameters for the new page to their defaults.

*Scope* *document*; this function starts *page* scope, and must always be paired with a matching *PDF\_end\_page\_ext()* call.

**Params** The following deprecated parameters will be ignored when using this function:  
*pagewidth, pageheight, ArtBox, BleedBox, CropBox, TrimBox.*

Table 8.8 Options for `PDF_begin_page_ext()`

option	type	description
group	string	(Required if the document uses page groups, but not allowed otherwise.) Name of the page group to which the page will belong. This name can be used to keep pages together in a page group and to address pages with <code>PDF_resume_page()</code> .
pagenumber	integer	If this option is specified with a value <i>n</i> , the page will be inserted before the existing page <i>n</i> within the page group specified in the group option (or the document if the document doesn't use page groups). If this option is not specified the page will be inserted at the end of the group.
separation-info	option list	An option list containing color separation details for the current page. This will be ignored in Acrobat, but may be useful in third-party software for identifying and correctly previewing separated pages in a pre-separated workflow:  pages (integer; required for the first page of a set of separation pages, but not allowed for subsequent pages of the same set) The number of pages which belong to the same set of separation pages comprising the color data for a single composite page. All pages in the set must appear sequentially in the file.  spotname (string; required unless spotcolor has been supplied) The name of the colorant for the current page.  spotcolor (spot color handle) A color handle describing the colorant for the current page.
topdown	boolean	If true, the origin of the coordinate system at the beginning of the page will be assumed in the top left corner of the page, and y coordinates will increase downwards; otherwise the default coordinate system will be used (see Section 3.2.1, «Coordinate Systems», page 57). Default: false

**void PDF\_end\_page\_ext(PDF \*p, const char \*optlist)**

Finish a page, and apply various options.

**optlist** An option list according to Table 8.9. Options specified in `PDF_end_page_ext()` have priority over identical options specified in `PDF_begin_page_ext()`.

**Scope** *page*; this function terminates *page* scope, and must always be paired with a matching `PDF_begin_page_ext()` call.

Table 8.9 Options for `PDF_begin_page_ext()` and `PDF_end_page_ext()`

option	type	description
action	action list	List of page actions for one or more of the following events. Default: empty list: open Actions to be performed when the page is opened. close Actions to be performed when the page is closed.
artbox bleedbox cropbox mediabox trimbox	rectangle	(The mediabox option is not allowed if the topdown option or parameter is true) Change the page box parameters of the current page. The coordinates of the respective box are specified in the default coordinate system (see Section 3.2.2, «Page Sizes and Coordinate Limits», page 59 for details). By default, only the MediaBox will be created by using the width and height parameters. The mediabox option will override the width and height options and parameters.
defaultgray defaultrgb defaultcmyk	icc handle	Set a default gray, RGB, or CMYK color space for the page according to the supplied profile handle.

Table 8.9 Options for PDF\_begin\_page\_ext() and PDF\_end\_page\_ext()

option	type	description
duration	float	Set the page display duration in seconds for the current page if openmode=full-screen (see Table 8.5). Default: 1
label	option list	An option list according to Table 8.7 specifying symbolic page names. specifying symbolic pages. The page name will be displayed as a page label (instead of the page number) in Acrobat's status line. The specified numbering scheme will be used for the current and subsequent pages until it is changed again. The combination of style/prefix/start values must be unique within a document.
rotate	integer	The page rotation value. The rotation will affect page display, but does not modify the coordinate system. Possible values are 0, 90, 180, 270. Default: 0
transition	keyword	Set the page transition for the current page in order to achieve special effects which may be useful when displaying the PDF in Acrobat's fullscreen mode as presentations if openmode=fullscreen (see Table 8.5). Default: replace split Two lines sweeping across the screen reveal the page blinds Multiple lines sweeping across the screen reveal the page box A box reveals the page wipe A single line sweeping across the screen reveals the page dissolve The old page dissolves to reveal the page glitter The dissolve effect moves from one screen edge to another replace The old page is simply replaced by the new page fly (PDF 1.5) The new page flies into the old page. push (PDF 1.5) The new page pushes the old page off the screen cover (PDF 1.5) The new page slides on to the screen and covers the old page. uncover (PDF 1.5) The old page slides off the screen and uncovers the new page. fade (PDF 1.5) The new page gradually becomes visible through the old one.
width	float or	(Not allowed if the toptdown option or parameter is true) The dimensions of the new page in points. Acrobat's page size limits are documented in Section 3.2.2, »Page Sizes and Coordinate Limits«, page 59. In order to produce landscape pages use width > height or the rotate option. PDFlib uses width and height to construct the page's MediaBox, but the MediaBox can also explicitly be set using the mediabox option. The width and height options will override the parameters with the same name.
height	keyword	
		The following symbolic page size names can be used as keywords by appending .width or .height (e.g. a4.width, a4.height). See Table 3.4 for the numerical values: ao, a1, a2, a3, a4, a5, a6, b5, letter, legal, ledger, 11x17

**void PDF\_suspend\_page(PDF \*p, const char \*optlist)**

Suspend the current page so that it can later be resumed.

**optlist** An option list for future use.

*Details* The full state of the current page (graphics, color, text, etc.) will be saved internally. It can later be resumed with *PDF\_resume\_page()* to add more content. Suspended pages must be resumed before they can be closed.

*Scope* *page*; this function starts *document* scope, and must always be paired with a matching *PDF\_resume\_page()* call. This function must not be used in Tagged PDF mode.

---

**void PDF\_resume\_page(PDF \*p, const char \*optlist)**

---

Resume a page to add more content to it.

**optlist** An option list according to Table 8.10.

*Details* The page must have been suspended with *PDF\_suspend\_page()*. It will be opened again so that more content can be added. All suspended pages must be resumed before they can be closed, even if no more content has been added.

*Scope* *document*; this function starts *page* scope, and must always be paired with a matching *PDF\_suspend\_page()* call.

Table 8.10 Options for *PDF\_resume\_page()*

option	type	description
group	string	(Required if the document uses page groups, but not allowed otherwise.) Name of the page group of the resumed page. The group name must have been defined with the groups option in <i>PDF_begin_document()</i> .
pagenumber	integer	If this option is supplied, the page with the specified number within the page group chosen in the group option (or in the document if the document doesn't use page groups) will be resumed. If this option is missing the last page in the group will be resumed.

---

**int PDF\_open\_file(PDF \*p, const char \*filename)****void PDF\_close(PDF \*p)**

---

Deprecated, use *PDF\_begin\_document()* and *PDF\_end\_document()*.

---

**void PDF\_begin\_page(PDF \*p, double width, double height)****void PDF\_end\_page(PDF \*p)**

---

Deprecated, use *PDF\_begin\_page\_ext()* and *PDF\_end\_page\_ext()*.

### 8.2.3 Parameter Handling

PDFlib maintains a number of internal parameters which are used for controlling PDFlib's operation and the appearance of the PDF output. Four functions are available for setting and retrieving both numerical and string parameters. All parameters (both keys and values) are case-sensitive. The descriptions of available parameters can be found in the respective sections in this chapter.

---

**double PDF\_get\_value(PDF \*p, const char \*key, double modifier)**

---

Get the value of some PDFlib parameter with numerical type.

**key** The name of the parameter to be queried.

**modifier** An optional modifier to be applied to the parameter. Whether a modifier is required and what it relates to is explained in the various parameter tables. If the modifier is unused it must be o.

*Returns* The numerical value of the parameter.

*Scope* Depends on *key*.

*See also* `PDF_get_pdi_value()`

---

**void PDF\_set\_value(PDF \*p, const char \*key, double value)**

---

Set the value of some PDFlib parameter with numerical type.

**key** The name of the parameter to be set.

**value** The new value of the parameter to be set.

*Scope* Depends on *key*.

---

**const char \* PDF\_get\_parameter(PDF \*p, const char \*key, double modifier)**

---

Get the contents of some PDFlib parameter with string type.

**key** The name of the parameter to be queried.

**modifier** An optional modifier to be applied to the parameter. Whether a modifier is required and what it relates to is explained in the various parameter tables. If the modifier is unused it must be 0.

*Returns* The string value of the parameter as a hypertext string. The returned string can be used until the end of the surrounding *document* scope. If no information is available an empty string will be returned.

*Scope* Depends on *key*.

*Bindings* C and C++: C and C++ clients must not free the returned string. PDFlib manages all string resources internally.

*See also* `PDF_get_pdi_parameter()`

---

**void PDF\_set\_parameter(PDF \*p, const char \*key, const char \*value)**

---

Set some PDFlib parameter with string type.

**key** The name of the parameter to be set.

**value** (Name string) The new value of the parameter to be set.

*Scope* Depends on *key*.

### 8.2.4 PDFlib Virtual File System (PVF) Functions

---

**void PDF\_create\_pvf(PDF \*p,  
const char \*filename, int len, const void \*data, size\_t size, const char \*optlist)**

---

Create a named virtual read-only file from data provided in memory.

**filename** (Name string) The name of the virtual file. This is an arbitrary string which can later be used to refer to the virtual file in other PDFlib calls.

**len** (Only for the C binding.) Length of *filename* (in bytes) for UTF-16 strings. If *len* = 0 a null-terminated string must be provided.

**data** A pointer to a memory buffer containing the data comprising the virtual file.

**size** (C language binding only) The length in bytes of the memory block containing the data.

**optlist** An option list according to Table 8.11.

*Details* The virtual file name can be supplied to any API function which uses input files (virtual files cannot be used for the generated PDF output; use an empty filename in *PDF\_delete()* to achieve this). Some of these functions may set a lock on the virtual file until the data is no longer needed. Virtual files will be kept in memory until they are deleted explicitly with *PDF\_delete\_pvf()*, or automatically in *PDF\_delete()*.

If *filename* refers to an existing virtual file an exception will be thrown. This function does not check whether *filename* is already in use for a regular disk file.

Unless the *copy* option has been supplied, the caller must not modify or free (delete) the supplied data before a corresponding successful call to *PDF\_delete\_pvf()*. Not obeying to this rule will most likely result in a crash.

*Scope* any

Table 8.11 Options for *PDF\_create\_pvf()*

option	type	description
copy	boolean	PDFlib will immediately create an internal copy of the supplied data. In this case the caller may dispose of the supplied data immediately after this call. The copy option will automatically be set to true in the COM, .NET, and Java bindings (default for other bindings: false). In other language bindings the data will not be copied unless the copy option is supplied.

---

**int *PDF\_delete\_pvf*(PDF \*p, const char \*filename, int len)**

---

Delete a named virtual file and free its data structures (but not the contents).

**filename** (Name string) The name of the virtual file as supplied to *PDF\_create\_pvf()*.

**len** (Only for the C binding.) Length of *filename* (in bytes) for UTF-16 strings. If *len* = 0 a null-terminated string must be provided.

*Returns* -1 (in PHP: 0) if the corresponding virtual file exists but is locked, and 1 otherwise.

*Details* If the file isn't locked, PDFlib will immediately delete the data structures associated with *filename*. If *filename* does not refer to a valid virtual file this function will silently do nothing. After successfully calling this function *filename* may be reused. All virtual files will automatically be deleted in *PDF\_delete()*.

The detailed semantics depend on whether or not the *copy* option has been supplied to the corresponding call to *PDF\_create\_pvf()*: If the *copy* option has been supplied, both the administrative data structures for the file and the actual file contents (data) will be freed; otherwise, the contents will not be freed, since the client is supposed to do so.

*Scope* any



## 8.2.5 Exception Handling

Table 8.12 lists relevant parameters and values for this section.

Table 8.12 Parameters and values for exception handling

function	key	explanation
set_parameter	warning	Enable or suppress warnings (nonfatal exceptions). Possible values are true and false. Scope: any. Default: true

---

### int PDF\_get\_errnum(PDF \*p)

---

Get the number of the last thrown exception or the reason for a failed function call.

- Returns** The number of an exception, or the reason code of the the most recently called function which failed with an error code.
- Scope** Between an exception thrown by PDFlib and *PDF\_delete()*. Alternatively, this function may be called after a function returned a -1 (in PHP: o) error code, but before calling any other function except those listed in this section.
- Bindings** In C++, Java, and PHP 5 this function is also available as *get\_errnum()* in the *PDFlibException* object.

---

### const char \*PDF\_get\_errmsg(PDF \*p)

---

Get the text of the last thrown exception or the reason for a failed function call.

- Returns** Text containing the description of the last exception thrown, or the reason why the most recently called function failed with an error code.
- Scope** Between an exception thrown by PDFlib and *PDF\_delete()*. Alternatively, this function may be called after a function returned a -1 (in PHP: o) error code, but before calling any other function except those listed in this section.
- Bindings** In C++, Java, and PHP 5 this function is also available as *get\_errmsg()* in the *PDFlibException* object.

---

### const char \*PDF\_get\_apiname(PDF \*p)

---

Get the name of the API function which threw the last exception or failed.

- Returns** The name of the function which threw an exception, or the name of the most recently called function which failed with an error code.
- Scope** Between an exception thrown by PDFlib and *PDF\_delete()*. Alternatively, this function may be called after a function returned a -1 (in PHP: o) error code, but before calling any other function except those listed in this section.
- Bindings** In C++, Java, and PHP 5 this function is also available as *get\_apiname()* in the *PDFlibException* object.

---

**void \*PDF\_get\_opaque(PDF \*p)**

---

Fetch the opaque application pointer stored in PDFlib.

**Details** This function returns the opaque application pointer stored in PDFlib which has been supplied in the call to *PDF\_new2()*. PDFlib never touches the opaque pointer, but supplies it unchanged to the client. This may be used in multi-threaded applications for storing private thread-specific data within the PDFlib object. It is especially useful for thread-specific exception handling.

**Scope** any

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

## 8.2.6 Utility Functions

These functions may be useful if the functionality is not available in the environment.

---

**const char \*PDF\_utf16\_to\_utf8(PDF \*p, const char \*utf16string, int len, int \*size)**

---

Convert a string from UTF-16 format to UTF-8.

**utf16string** The string to be converted. A Byte Order Mark (BOM) in the string will be evaluated. If it is missing the platform's native byte ordering is assumed.

**len** (Only for the C binding.) Length of *utf16string* (in bytes).

**size** (Only for the C binding.) C-style pointer to a memory location where the length of the returned string (in bytes) will be stored. If the pointer is NULL it will be ignored.

**Returns** The converted UTF-8 string. The generated UTF-8 string will start with a BOM (*\xEF\xBB\xBF*). On EBCDIC platforms the conversion result including the BOM will finally be converted to EBCDIC. The returned string is valid until the next call to any PDFlib function other than *PDF\_utf16\_to\_utf8()* and *PDF\_utf8\_to\_utf16()*, or until an exception is thrown. Clients must copy the string if they need it longer. The memory used for the converted string will be managed by PDFlib.

**Scope** any

**Bindings** This function is not available in Unicode-capable language bindings.

---

**const char \*PDF\_utf8\_to\_utf16(PDF \*p, const char \*utf8string, const char \*ordering, int \*size)**

---

Convert a string from UTF-8 format to UTF-16.

**utf8string** The string to be converted, which must contain a valid UTF-8 sequence (on EBCDIC platforms it must be encoded in EBCDIC). If a Byte Order Mark (BOM) is present, it will be removed.

**ordering** Specifies the byte ordering of the result string:

- ▶ *utf16* or an empty string: The converted string will not have a BOM, and will be stored in the platform's native byte order.
- ▶ *utf16le*: The converted string will be formatted in little endian format, and will be prefixed with the LE BOM (*\xFF\xFE*).

- *utf16be*: The converted string will be formatted in big endian format, and will be prefixed with the BE BOM (`\xFE\xFF`).

**size** (Only for the C binding.) C-style pointer to a memory location where the length of the returned string (in bytes) will be stored.

**Returns** The converted UTF-16 string. The returned string is valid until the next call to any PDFlib function other than *PDF\_utf16\_to\_utf8()* and *PDF\_utf8\_to\_utf16()*, or until an exception is thrown. Clients must copy the string if they need it longer. The memory used for the converted string will be managed by PDFlib.

**Scope** *any*

**Bindings** This function is not available in Unicode-capable language bindings.

# 8.3 Text Functions

## 8.3.1 Font Handling

Table 8.13 lists relevant parameters and values for this section.

Table 8.13 Parameters and values for the font functions (see Section 8.2.3, »Parameter Handling«, page 198)

function	key	explanation
set_parameter	FontAFM FontPFM FontOutline Encoding HostFont SearchPath	The corresponding resource file line as it would appear for the respective category in a UPR file (see Section 3.1.6, »Resource Configuration and File Searching«, page 51). Multiple calls add new entries to the internal list. (See also resourcefile in Table 8.2). Scope: any
get_value	font	Returns the identifier of the current font which has been set with PDF_setfont(), or -1 (in PHP: 0) if no font is set. Scope: page, pattern, template, glyph
get_value	fontmaxcode	Returns the number of valid glyph ids for the font in the modifier. Scope: any
get_parameter	fontname	The name of the current font which must have been previously set with PDF_setfont(). Scope: page, pattern, template, glyph
get_parameter	fontencoding	The name of the encoding or CMap used with the current font. A font must have been previously set with PDF_setfont(). Scope: page, pattern, template, glyph
get_value	fontsize	Returns the size of the current font which must have been previously set with PDF_setfont(). Scope: page, pattern, template, glyph
get_parameter	fontstyle	The style of the current font, which resembles the fontstyle option (normal, bold, italic, or bolditalic). Scope: page, pattern, template, glyph
get_value	capheight ascender descender	Returns metrics information for the font identified by the modifier. See Section 4.6, »Text Metrics and Text Variations«, page 97 for more details. The values are measured in fractions of the font size, and must therefore be multiplied by the desired font size. Scope: any
set_parameter	fontwarning	If false, PDF_load_font() returns -1 (in PHP: 0) if the font/encoding combination cannot be loaded (instead of throwing an exception). Default: true. Scope: any
get_value	monospace	Returns the value of the monospace option for the current font if it has been set, and 0 otherwise. Scope: page, pattern, template, glyph
set_value	subsetlimit	Disables font subsetting if the document uses more than the given percentage of characters in the font. Default value: 100 percent. Scope: any
set_value	subsetminsize	Subsetting will only be applied to fonts above this size in Kilobyte (see Section 4.3, »Font Embedding and Subsetting«, page 78). Default: 100 KB. Scope: any
set_parameter	auto-subsetting	Controls automatic activation of subsetting for TrueType and OpenType fonts . Default: true. Scope: any
set_parameter	autocidfont	Controls automatic conversion of TrueType fonts with encodings other than macroman and winansi to CID fonts (see Section 4.3, »Font Embedding and Subsetting«, page 78). Default: true. Scope: any
set_parameter	unicodemap	Controls generation of ToUnicode CMaps (see Section 4.5.1, »Unicode for Page Content and Hypertext«, page 89). This parameter will be ignored in Tagged PDF mode Default: true. Scope: any

---

```
int PDF_load_font(PDF *p,  
    const char *fontname, int len, const char *encoding, const char *optlist)
```

---

Search for a font and prepare it for later use.

**fontname** (Name string) The real or alias name of the font. It will be used to find font data according to the description in Section 4.3.1, »How PDFlib Searches for Fonts«, page 78. Case is significant.

**len** (C language binding only) Length of *fontname* in bytes for UTF-16 strings. If *len* = 0 a null-terminated string must be provided.

**encoding** The encoding to be used with the font, which must be one of the following:

- ▶ one of the predefined 8-bit encodings *winansi*, *macroman*, *macroman\_apple*, *macroman\_euro*, *ebcdic*, *pdfdoc*, *iso8859-X*, *cpXXXX*, or *U+XXXX*;
- ▶ *host* or *auto* for an automatically selected encoding;
- ▶ the name of a user-defined encoding loaded from file or defined via *PDF\_encoding\_set\_char()*;
- ▶ *unicode* for Unicode-based addressing;
- ▶ *glyphid* for glyph id addressing;
- ▶ *builtin* to select the font's internal encoding;
- ▶ the name of a standard CMap. In this case *fontname* must be the name of a standard CJK font; custom CJK fonts are only supported with *unicode* encoding (see Section 4.7, »Chinese, Japanese, and Korean Text«, page 101);
- ▶ an encoding name known to the operating system (not available on all platforms).

Care must be taken to choose an encoding which is compatible with the font, and matches the available input and desired output. Review Section 4.4, »Encoding Details«, page 83, for more details. Case is significant for *encoding*.

**optlist** An option list according to Table 8.14.

**Returns** A font handle for later use with *PDF\_setfont()*. The behavior of this function changes when the *fontwarning* parameter or option is set to *false*. In this case the function returns an error code of -1 (in PHP: 0) if the requested font/encoding combination cannot be loaded, and does not throw an exception. However, exceptions will still be thrown when bad parameters are supplied.

The returned number – the font handle – doesn't have any significance to the user other than serving as an argument to *PDF\_setfont()* and related functions. In particular, requesting the same font/encoding combination in different documents may result in different font handles.

When calling this function again with the same font name the same font handle as in the first call will be returned unless a different *encoding* parameter or *fontstyle* option has been supplied.

Conflicting options: when a font is loaded via *PDF\_load\_font()* or requested via *PDF\_fill\_textblock()* without embedding, kerning, or subsetting, these options will be ignored if the same font is loaded again later.

**Details** This function prepares a font for later use with *PDF\_setfont()*. The metrics will be loaded from memory or from a (virtual or disk-based) metrics file. If the requested font/encoding combination cannot be used due to configuration problem (e.g., a font, metrics, or encoding file could not be found, or a mismatch was detected), an exception will be

thrown unless the *fontwarning* parameter is set to false. Otherwise, the value returned by this function can be used as font argument to other font-related functions.

*Scope* document, page, pattern, template, glyph

*Params* See Table 8.13.

*PDF/X* The *embedding* option must be true.

Table 8.14 Options for *PDF\_load\_font()*

option	type	description
auto-subsetting	boolean	Dynamically decide whether or not the font will be subset, subject to the <i>subsetlimit</i> and <i>subsetminsize</i> parameters and the actual usage of glyphs. This option will be ignored when the subsetting option has been supplied. Default: the value of the global <i>autosubsetting</i> parameter.
autocidfont	boolean	If true, TrueType fonts with 8-bit encoding except <i>winansi</i> , <i>macroman</i> , <i>builtin</i> and OpenType fonts without glyph names will automatically be stored as CID fonts. This avoids problems with certain non-accessible glyphs outside <i>winansi</i> encoding. Default: the value of the global <i>autocidfont</i> parameter.
embedding	boolean	Controls whether or not the font will be embedded. This does not have any effect on Type 3 fonts. If a font is to be embedded, the font outline file must be available in addition to the metrics information (this is irrelevant for TrueType and OpenType fonts), and the actual font outline definition will be included in the PDF output. If a font is not embedded, only general information about the font is included in the PDF output. Default: false
fontstyle	keyword	Controls the creation of artificial font styles. These work only for TrueType and OpenType fonts which are not embedded (see Section 4.6.3, »Text Variations«, page 99). Possible keywords are <i>normal</i> , <i>bold</i> , <i>italic</i> , <i>bolditalic</i> . Default: <i>normal</i> .
fontwarning	boolean	If true, an exception will be thrown when the requested font/encoding combination cannot be loaded; If false an error code will be returned. (The encoding search is under control of the <i>fontwarning</i> parameter, but not under control of the <i>fontwarning</i> option.) Default: the value of the global <i>fontwarning</i> parameter.
kerning	boolean	Controls whether or not kerning values will be read from the font (see Section 4.6, »Text Metrics and Text Variations«, page 97). Default: false
monospace	integer 1...2048	Forces all glyphs in the font to use the specified width (in the font coordinate system: 1000 units equal the font size). For Type 3 fonts all glyph widths which are different from 0 will be modified. This option is only recommended for standard CJK fonts, and not supported for core fonts; it will be ignored if the font is embedded. Default: absent (metrics from the font will be used)
subsetlimit	float or percentage	Font subsetting will be disabled if the percentage of glyphs used in the document related to the total number of glyphs in the font exceeds the provided percentage. Default: the value of the global <i>subsetlimit</i> parameter.
subsetminsize	float	Font subsetting will be disabled if the size of the original font file is less than the provided value in KB. Default: the value of the global <i>subsetminsize</i> parameter.
subsetting	boolean	Controls whether or not the font will be subset, subject to the total number of glyphs used in the document and the <i>subsetlimit</i> and <i>subsetminsize</i> options (see Section 4.3, »Font Embedding and Subsetting«, page 78). Default: false
unicodemap	boolean	Controls the generation of ToUnicode CMaps (see Section 4.5.1, »Unicode for Page Content and Hypertext«, page 89). This option will be ignored in Tagged PDF mode. Default: true

---

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

---

Set the current font in the specified size.

**font** A font handle returned by *PDF\_load\_font()*.

**fontsize** Size of the font, measured in units of the current user coordinate system. The font size must not be 0; negative font size will result in mirrored text relative to the current transformation matrix.

*Details* The font must be set on each page before drawing any text. Font settings will not be retained across pages. The current font can be changed an arbitrary number of times per page.

*Scope* *page, pattern, template, glyph*

*Params* See Table 8.13. This function automatically sets the *leading* parameter to *fontsize*.

---

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

---

Deprecated, use *PDF\_load\_font()*.

### 8.3.2 User-defined (Type 3) Fonts

---

**void PDF\_begin\_font(PDF \*p, char \*fontname, int reserved,  
double a, double b, double c, double d, double e, double f, const char \*optlist)**

---

Start a Type 3 font definition.

**fontname** (Name string) The name under which the font will be registered, and can later be used with *PDF\_load\_font()*.

**reserved** (C language binding only.) Reserved, must be 0.

**a, b, c, d, e, f** The elements of the font matrix. This matrix defines the coordinate system in which the glyphs will be drawn. The six values make up a matrix in the same way as in PostScript and PDF (see references). In order to avoid degenerate transformations,  $a*d$  must not be equal to  $b*c$ .

A typical font matrix for a 1000 x 1000 coordinate system is  $[0.001, 0, 0, 0.001, 0, 0]$ .

**optlist** An option list according to Table 8.15.

*Details* This function will reset all text, graphics, and color state parameters to their defaults. The font may contain an arbitrary number of glyphs, but only 256 glyphs can be accessed via an encoding. The defined font can be used until the end of the current *document* scope.

*Scope* *document, page*; this function starts *font* scope, and must always be paired with a matching *PDF\_end\_font()* call.

Table 8.15 Options for `PDF_begin_font()`

option	type	description
<i>colorized</i>	<i>boolean</i>	If <i>true</i> , the font may explicitly specify the color of individual characters. If <i>false</i> , all characters will be drawn with the current color (at the time the font is used, not when it is defined), and the glyph definitions must not contain any color operators or images other than masks. Default: <i>false</i>

---

**void PDF\_end\_font(PDF \*p)**

---

Terminate a Type 3 font definition.

*Scope* *font*; this function terminates *font* scope, and must always be paired with a matching `PDF_begin_font()` call.

---

**void PDF\_begin\_glyph(PDF \*p,  
char \*glyphname, double wx, double llx, double lly, double urx, double ury)**

---

Start a glyph definition for a Type 3 font.

**glyphname** The name of the glyph. This name must be used in any encoding which will be used with the font. Glyph names within a font must be unique.

**wx** The width of the glyph in the glyph coordinate system, as specified by the font's matrix.

**llx, lly, urx, ury** If the font's *colorized* option is *false* (which is default), the coordinates of the lower left and upper right corners of the glyph's bounding box. The bounding box values must be correct in order to avoid problems with PostScript printing. If the font's *colorized* option is *true*, all four values must be 0.

*Details* The glyphs in a font can be defined using text, graphics, and image functions. Images, however, can only be used if the font's *colorized* option is *true*, or the image has been opened with the *mask* option. It is strongly suggested to use the inline image feature (see Section 5.1.1, »Basic Image Handling«, page 125) for defining bitmaps in Type 3 fonts.

Since the complete graphics state of the surrounding page will be inherited for the glyph definition when the *colorized* option is *true*, the glyph definition should explicitly set any aspect of the graphics state which is relevant for the glyph definition (e.g., line-width).

*Scope* *page, font*; this function starts *glyph* scope, and must always be paired with a matching `PDF_end_glyph()` call.

---

**void PDF\_end\_glyph(PDF \*p)**

---

Terminate a glyph definition for a Type 3 font.

*Scope* *glyph*; this function terminates *glyph* scope, and must always be paired with a matching `PDF_begin_glyph()` call.



### 8.3.3 Encoding Definition

```
void PDF_encoding_set_char(PDF *p,
    const char *encoding, int slot, const char *glyphname, int uv)
```

Add a glyph name and/or Unicode value to a custom encoding.

**encoding** The name of the encoding. This is the name which must be used with `PDF_load_font()`. The encoding name must be different from any built-in encoding and all previously used encodings.

**slot** The position of the character in the encoding to be defined, with  $0 \leq slot \leq 255$ . A particular slot must only be filled once within a given encoding.

**glyphname** The character's name.

**uv** The character's Unicode value.

*Details* This function can be called multiply to define up to 256 character slots in an encoding. More characters may be added to a particular encoding until it has been used for the first time; otherwise an exception will be raised. Not all code points must be specified; undefined slots will be filled with `.notdef`.

- There are three possible combinations of glyph name and Unicode value:
- ▶ `glyphname` supplied, `uv = 0`: this parallels an encoding file without Unicode values;
  - ▶ `uv` supplied, but no `glyphname` supplied: this parallels a codepage file;
  - ▶ `glyphname` and `uv` supplied: this parallels an encoding file with Unicode values;

The defined encoding can be used until the end of the current *object* scope.

*Scope* `object, document, page, pattern, template, path, font, glyph`

### 8.3.4 Simple Text Output

*Note* All text supplied to the functions in this section must match the encoding selected with `PDF_load_font()`. This applies to 8-bit text as well as Unicode or other encodings selected via a CMap. Due to restrictions in Acrobat, text strings must not exceed 32 KB in length.

Table 8.16 lists relevant parameters and values for this section.

Table 8.16 Parameters and values for the text functions

function	key	explanation
set_parameter	autospace	If true and the current font is Unicode-compatible, PDFlib will automatically add a space character (0x20) after each text output generated with a show operation. This may be useful for generating Tagged PDF (see Section 7.5, »Tagged PDF«, page 176). Note that adding spaces changes the current text position after the show operation. Default: false. Scope: any
set_parameter	charref	(Not for UTF-8 text formats) If true, enable substitution of numeric and character entity references (see Section 4.5.5, »Character References«, page 94). Default: false
set_value get_value	charspacing	Set or get the character spacing, i.e., the shift of the current point after placing individual characters in a string. It is specified in units of the user coordinate system, and is reset to the default of 0 at the beginning and end of each page. In order to spread characters apart use positive values for horizontal writing mode, and negative values for vertical writing mode. Scope: page, pattern, template, glyph, document.

Table 8.16 Parameters and values for the text functions

function	key	explanation
set_parameter	glyphwarning	If true, an exception will be thrown when a glyph cannot be shown because the font does not contain the corresponding glyph description. If false, missing glyphs will be replaced with a space character or glyph ID 0. Default: false. Scope: any
set_value get_value	horizscaling	Set or get the horizontal text scaling to the given percentage. Text scaling shrinks or expands the text by a given percentage. It is set to the default of 100 at the beginning and end of each page. Text scaling always relates to the horizontal coordinate. Scope: page, pattern, template, glyph, document.
set_value get_value	italicangle	Specifies the italic (slant) angle of text in degrees (between -90° and 90°). Negative values can be used to simulate italic text when only a regular font is available, especially for CJK fonts (see Section 4.6.3, «Text Variations», page 99). Default: 0 (this parameter will be reset at the beginning and end of each page). Scope: page, pattern, template, glyph, document
set_parameter get_parameter	kerning	If true, enable kerning for fonts which have been opened with the kerning option; disable if false. (see Section 4.6, «Text Metrics and Text Variations», page 97). Default: true. Scope: any
set_value get_value	leading	Set or get the leading, which is the distance between baselines of adjacent lines of text. The leading is used for PDF_continue_text(). It is set to the value of the font size when a new font is selected using PDF_setfont(). Setting the leading equal to the font size results in dense line spacing (leading = 0 will result in overprinting lines). However, ascenders and descenders of adjacent lines will generally not overlap. Scope: page, pattern, template, glyph.
set_parameter get_parameter	textformat	Specifies the format in which the text output functions will expect the client-supplied strings. Possible values are bytes, utf8, ebcdicutf8 (only on iSeries and zSeries), utf16, utf16le, utf16be, and auto (see Section 4.5.2, «Content Strings, Hypertext Strings, and Name Strings», page 90). Default: auto. Scope: any
set_value get_value	textrendering	Set or get the current text rendering mode. It is set to the default of 0 at the beginning of each page. Scope: page, pattern, template, glyph. 0 fill text 1 stroke text (outline) 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
set_value get_value	textrise	Set or get the text rise parameter, which specifies the distance between the desired text position and the default baseline. Positive values of text rise move the text up. The text rise always relates to the vertical coordinate. This may be useful for superscripts and subscripts. The text rise is set to the default value of 0 at the beginning of each page. Scope: page, pattern, template, glyph.
get_value	textx texty	Get the x or y coordinate, respectively, of the current text position. Scope: page, pattern, template, glyph.
set_parameter get_parameter	underline overline strikeout	Set or get the current underline, overline, and strikeout modes, which are retained until they are explicitly changed, or a new page is started. These modes can be set independently from each other, and are reset to false at the beginning of each page (see Section 4.6, «Text Metrics and Text Variations», page 97). Scope: page, pattern, template, glyph. true underline/overline/strikeout text false do not underline/overline/strikeout text

Table 8.16 Parameters and values for the text functions

function	key	explanation
set_value get_value	wordspacing	Set or get the word spacing, i.e., the shift of the current point after placing individual words in a line. In other words, the current point is moved horizontally after each space character (0x20). The spacing value is given in text space units, and is reset to the default of 0 at the beginning and end of each page. Scope: page, pattern, template, glyph, document.

---

**void PDF\_set\_text\_pos(PDF \*p, double x, double y)**

---

Set the position for text output on the page.

**x, y** The current text position to be set.

*Details* The text position is set to the default value of (0, 0) at the beginning of each page. The current point for graphics output and the current text position are maintained separately.

*Scope* page, pattern, template, glyph

*Params* See Table 8.16.

---

**void PDF\_show(PDF \*p, const char \*text)**  
**void PDF\_show2(PDF \*p, const char \*text, int len)**

---

Print text in the current font and size at the current text position.

**text** (Content string) The text to be printed. In C *text* must not contain null characters when using *PDF\_show()*, since it is assumed to be null-terminated; use *PDF\_show2()* for strings which may contain null characters.

**len** (Only for *PDF\_show2()*) Length of *text* (in bytes) for UCS-2 strings. If *len* = 0 a null-terminated string must be provided.

*Details* The font must have been set before with *PDF\_setfont()*. The current text position is moved to the end of the printed text.

*Scope* page, pattern, template, glyph

*Params* See Table 8.16.

*Bindings* *PDF\_show2()* is only available in C since in all other bindings arbitrary string contents can be supplied with *PDF\_show()*.

---

**void PDF\_xshow(PDF \*p, const char \*text, int len, const double \*xadvancelist)**

---

Print text in the current font and size, using individual horizontal positions.

**text** (Content string) The text to be printed.

**len** (Only for the C language binding) Length of *text* (in bytes) for UCS-2 strings. If *len* = 0 a null-terminated string must be provided.

**xadvancelist** An array of x advance values for the glyphs in text. Each value specifies the relative horizontal displacement (in user coordinates) after a glyph has been placed.

The array length must be equal to the number of glyphs in text (not necessarily equal to *len*, which is the the number of bytes!).

*Details* The font must have been set before with *PDF\_setfont()*.

*Scope* *page, pattern, template, glyph*

*Params* See Table 8.16.

*Bindings* Only available in the C and C++ language binding. Other bindings can use the *xadvance-list* option in *PDF\_fit\_textline()* to achieve the same functionality.

---

**void PDF\_show\_xy(PDF \*p, const char \*text, double x, double y)**  
**void PDF\_show\_xy2(PDF \*p, const char \*text, int len, double x, double y)**

---

Print text in the current font.

**text** (Content string) The text to be printed. In C *text* must not contain null characters when using *PDF\_show\_xy()*, since it is assumed to be null-terminated; use *PDF\_show\_xy2()* for strings which may contain null characters.

**x, y** The position in the user coordinate system where the text will be printed.

**len** (Only for *PDF\_show\_xy2()*) Length of *text* (in bytes) for UCS-2 strings. If *len = 0* a null-terminated string must be provided.

*Details* The font must have been set before with *PDF\_setfont()*. The current text position is moved to the end of the printed text.

*Scope* *page, pattern, template, glyph*

*Params* See Table 8.16.

*Bindings* *PDF\_show\_xy2()* is only available in C since in all other bindings arbitrary string contents can be supplied with *PDF\_show\_xy()*.

---

**void PDF\_continue\_text(PDF \*p, const char \*text)**  
**void PDF\_continue\_text2(PDF \*p, const char \*text, int len)**

---

Print text at the next line.

**text** (Content string) The text to be printed. If this is an empty string, the text position will be moved to the next line anyway. In C *text* must not contain null characters when using *PDF\_continue\_text()*, since it is assumed to be null-terminated; use *PDF\_continue\_text2()* for strings which may contain null characters.

**len** (Only for *PDF\_continue\_text2()*) Length of *text* (in bytes) for UCS-2 strings. If *len = 0* a null-terminated string must be provided as in *PDF\_continue\_text()*.

*Details* The positioning of text (*x* and *y* position) and the spacing between lines is determined by the *leading* parameter and the most recent call to *PDF\_fit\_textline()*, *PDF\_show\_xy()* or *PDF\_set\_text\_pos()*. The current point will be moved to the end of the printed text; the *x* position for subsequent calls of this function will not be changed.

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

*Params* See Table 8.16.

*Bindings* `PDF_continue_text2()` is only available in C since in all other bindings arbitrary string contents can be supplied with `PDF_continue_text()`.

**void PDF\_fit\_textline(PDF\*p, const char \*text, int len, double x, double y, const char \*optlist)**

Place a single line of text at position (x, y) subject to various options.

**text** (Content string) The text to be printed.

**len** (C binding only) Length of *text* (in bytes) for UCS-2 strings. If *len* = 0 a null-terminated string must be provided.

**x, y** The coordinates of the reference point in the user coordinate system where the text will be placed, subject to various options.

**optlist** An option list specifying formatting options according to Table 8.17 and appearance options according to Table 8.18.

*Details* The current graphics state will not be modified by this function. In particular, the current font will be unaffected. However, the current text position will be adjusted to point to the end of the generated text output.

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

*Params* See Table 8.16.

Table 8.17 Formatting options for `PDF_fit_textline()`

key	type	explanation
<i>boxsize</i>	<i>list of floats</i>	Two values specifying the width and height of a box, relative to which the text will be placed and possibly scaled. The lower left corner of the box coincides with the reference point (x, y). Placing the text and fitting it into the box is controlled by the position and fitmethod options. If width = 0, only the height is considered; If height = 0, only the width is considered. In these cases the text will be placed relative to the vertical line from (x, y) to (x, y+height), or the horizontal line from (x, y) to (x+width, y), respectively. Default: {0 0}.

Table 8.17 Formatting options for PDF\_fit\_textline()

key	type	explanation
fitmethod	keyword	Specifies the method used to fit the text into the specified box. This option will be ignored if no box has been specified. Default: nofit.
		nofit Position the text only, without any scaling or clipping.
		clip Position the text, and clip it at the edges of the box.
		meet Position the text according to the position option, and scale it such that it entirely fits into the box while preserving its aspect ratio.
		Generally at least two edges of the text will meet the corresponding edges of the box. The dpi and scale options are ignored.
		auto This method tries to fit the text into the box automatically. In detail: Same as nofit if the text fits into the box. Otherwise a scaling factor is calculated such that the text fits into the box. If this factor is larger than the shrinklimit option the text is distorted to fit into the box, otherwise the meet method is applied.
		slice Position the text according to the position option, and scale it such that it entirely covers the box, while preserving the aspect ratio and making sure that at least one dimension of the text is fully contained in the box. Generally parts of the text's other dimension will extend beyond the box, and will therefore be clipped.
locallink	option list	entire Position the text according to the position option, and scale it such that it entirely covers the box. Generally this method will distort the text. The scale option will be ignored.
		If this option is provided, a functional local link will be created from the text, i.e. an annotation with type=Link will be created with default options or those provided in the option list. The following options can be provided (for details see Table 8.48): annotcolor, borderstyle, dasharray, highlight (the action and usercoordinates options will be set automatically).
margin	list of floats	One or two float values describing additional horizontal and vertical extensions of the text box. Default: 0.
orientate	keyword	Specifies the desired orientation of the text when it is placed. Default: north.
		north upright
		east pointing to the right
		south upside down
		west pointing to the left
position	list of floats	(Alignment control) One or two values specifying the position of the reference point (x, y) within the text's bounding box with {0 0} being the lower left corner of the text box, and {100 100} the upper right corner. If the boxsize option has been specified, the position option also specifies the positioning of the target box. The values are expressed as percentages of the text's width and height. If both percentages are equal it is sufficient to specify a single float value. Some examples: {0 50} results in left-justified text; {50 50} results in centered text; {100 50} results in right-justified text. Default: 0 (lower left corner)
rotate	float	Rotate the coordinate system, using the reference point as center and the specified value as rotation angle in degrees. This results in the box and the text being rotated. The rotation will be reset when the text has been placed. Default: 0.
weblink	option list	If this option is provided, a functional weblink will be created from the text, i.e. an annotation with type=Link will be created with default options or those provided in the option list. The following options can be provided (for details see Table 8.48): annotcolor, borderstyle, dasharray, highlight (the action and usercoordinates options will be set automatically).

Table 8.17 Formatting options for PDF\_fit\_textline()

key	type	explanation
xadvancelist	list of floats	Specifies the advance width of all glyphs in the text in user coordinates. The length of the list must be less or equal than the number of glyphs in the text. If the length is less than the number of glyphs a warning will be thrown if glyphwarning is true. The xadvance values will be used instead of the standard glyph widths. Other effects, such as kerning and character spacing, are unaffected.

Table 8.18 Appearance options for PDF\_fit\_textline() and direct or inline options for PDF\_create\_textflow()

key	type	explanation
charref	boolean	(Not for UTF-8 text formats) If true, enable substitution of numeric and character entity references (see Section 4.5.5, »Character References«, page 94). Default: false
charspacing	float or percentage	The character spacing (see Table 8.16). Percentages are based on fontsize. Default: the global charspacing parameter.
fillcolor	color	Fill color of the text. Default: the current fill color
font	font handle	A font handle returned by PDF_load_font(). Default: the current font
fontsize	float	(Required if the font option is provided) Size of the font, measured in units of the current user coordinate system. Default: the current font size
glyphwarning	boolean	If true, an exception will be thrown when a glyph cannot be shown because the font does not contain the corresponding glyph description. If false, glyphs missing from a font will be replaced with a space character or glyph ID 0. Default: the global glyphwarning parameter.
horizscaling	float or percentage	The horizontal text scaling (see Table 8.16). Default: the global horizscaling parameter
italicangle	float	Specifies the italic (slant) angle of text in degrees (see Section 4.6.3, »Text Variations«, page 99). Default: the global italicangle parameter
kerning	boolean	Kerning behavior (see Table 8.16). Default: the global kerning parameter
overline	boolean	Overline mode (see Table 8.16). Default: the global overline parameter
strikeout	boolean	Strikeout mode (see Table 8.16). Default: the global strikeout parameter
strokecolor	color	Stroke color of the text. Default: the current stroke color
shrinklimit	float or percentage	The lower limit of the shrinkage factor which will be applied to fit text. Default: 0.75
textformat	keyword	The format used to interpret the supplied text (see Section 4.5.2, »Content Strings, Hypertext Strings, and Name Strings«, page 90). Default: the global textformat parameter.
textrendering	integer	The text rendering mode (see Table 8.16). Default: the global textrendering parameter
textrise	float or percentage	The text rise mode (see Table 8.16). Percentages are based on fontsize. Default: the global text rise parameter
underline	boolean	Underline mode (see Table 8.16). Default: the global underline parameter
wordspacing	float or percentage	The word spacing (see Table 8.16). Percentages are based on fontsize. Default: global wordspacing parameter

---

**double PDF\_stringwidth(PDF \*p, const char \*text, int font, double fontsize)**  
**double PDF\_stringwidth2(PDF \*p, const char \*text, int len, int font, double fontsize)**

---

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

**text** (Content string) The text for which the width will be queried. In C *text* must not contain null characters when using *PDF\_stringwidth()*, since it is assumed to be null-terminated; use *PDF\_stringwidth2()* for strings which may contain null characters.

**len** (Only for *PDF\_stringwidth2()*) Length of *text* (in bytes) for UCS-2 strings. If *len = 0* a null-terminated must be provided.

**font** A font handle returned by *PDF\_load\_font()*. The corresponding font must not be a CJK font with a non-Unicode CMap. If *font* refers to such a font, this function returns 0 regardless of the *text* and *fontsize* parameters (unless the *monospace* option has been supplied when loading the font).

**fontsize** Size of the font, measured in units of the user coordinate system (see *PDF\_setfont()*).

**Returns** The width of *text* in an arbitrary font which has been selected with *PDF\_load\_font()* and the supplied *fontsize*. The returned width value may be negative (e.g., when negative horizontal scaling has been set).

**Details** The width calculation takes the current values of the following text parameters into account: horizontal scaling, kerning, character spacing, and word spacing.

**Scope** *page, pattern, template, path, glyph, document*

**Params** See Table 8.16.

**Bindings** *PDF\_stringwidth2()* is only available in C since in all other bindings arbitrary string contents can be supplied with *PDF\_stringwidth()*.

---

**int PDF\_show\_boxed(PDF \*p, const char \*text, double x, double y,  
double width, double height, const char \*mode, const char \*feature)**

---

Deprecated, use *PDF\_fit\_textline()* for single lines, or the *PDF\_\*\_textflow()* functions for multi-line formatting. Using *minspacing=100%*, *maxspacing=10000%*, *nofitlimit= 100%*, and *shrinklimit= 100%* in the latter case will achieve similar results as *PDF\_show\_boxed()*. The number of characters remaining after formatting (the value that would be returned by *PDF\_show\_boxed()*) can be retrieved by using the *remainchars* option in *PDF\_info\_textflow()*.

### 8.3.5 Multi-Line Text Output with Textflows

---

**int PDF\_create\_textflow(PDF \*p, const char \*text, int len, const char \*optlist)**

---

Preprocess text for later formatting and create a textflow object.

**text** (Content string) The contents of the textflow. It may contain text in various encodings and inline option lists according to Table 8.19 and Table 8.21.



**len** (C language binding only) The length of text in bytes, or 0 for null-terminated strings.

**optlist** An option list specifying textflow options according to Table 8.19 or Table 8.21. Options specified in *optlist* will be evaluated before those in inline option lists contained in *text* so that inline options have precedence over options provided in the *optlist* parameter.

*Details* A textflow handle which can be used in calls to *PDF\_fit\_textflow()*, *PDF\_info\_textflow()*, and *PDF\_delete\_textflow()*. The handle is valid until the end of the enclosing *document* scope, or until *PDF\_delete\_textflow()* is called with this handle. In case of an error the function returns an error code of -1 (in PHP: 0) if the *textwarning* parameter or option is *false*. If it is *true* the function will throw an exception in case of an error.

*Details* This function processes the supplied text and creates an internal data structure from it. It determines text portions (e.g. words) which will later be used by the formatter, processes inline option lists, converts the text to Unicode if possible, determines potential line breaks, and calculates the width of text portions based on font and text options. Searching for inline option lists can be disabled for parts or all of the text by supplying the *textlen* option in the *optlist* parameter.

This function does not create any output in the generated PDF document, but only prepares the text. Use *PDF\_fit\_textflow()* to create output with the preprocessed text-flow handle.

By default, a new line will be forced by the characters VT, LS, LF, CR, CRLF, NEL, PS, and FF (see Table 4.8 for a description of these characters). All of these except VT and LS force a new paragraph (which means that the *parindent* option will be effective). FF immediately stops the process of fitting text to the current fitbox (the function *PDF\_fit\_textflow()* will be exited with a return string of *\_nextpage*).

A horizontal tab character (HT) sets a new start position for subsequent text. The details of this are controlled by the *hortabmethod* and *hortabsize* options.

Soft hyphen characters (SHY) will be replaced with the character specified in the *hyphenchar* option if there is a line break after the soft hyphen. See Section 4.9.8, »Controlling the Linebreak Algorithm«, page 122 for more details.

*Scope* any except *object*

Table 8.19 Option for *PDF\_create\_textflow()*

option	type	explanation
<i>textwarning</i>	boolean	If true, an exception will be thrown when an error is found in an option list or the text (e.g. a character value is found which cannot be represented with the chosen font). Otherwise the function will return an error code of -1 (in PHP: 0), and (for unavailable glyphs) replace the glyph with a space character. Default: true
<i>fixedtext-format</i>	boolean	(Will be ignored in the Unicode-aware languages Java and Tcl) If true, all text fragments and inline options lists will use the same textformat, which must be one of utf8, utf16, utf16be, or utf16le.  If false, inline option lists including the delimiters must be encoded in winansi (or ebcdic on EBCDIC-based platforms). As an exception to this rule, the begoptlist-char must be encoded in the encoding of the preceding text fragment if this fragment uses a Unicode-compatible 8-bit encoding and the textlen option is not supplied.  Default: false

---

```
const char *PDF_fit_textflow(PDF *p,  
    int textflow, double llx, double lly, double urx, double ury, const char *optlist)
```

---

Format the next portion of a textflow into a rectangular area.

**textflow** A textflow handle returned by a call to *PDF\_create\_textflow()*.

**llx, lly, urx, ury** *x* and *y* coordinates of the lower left and upper right corners of the target rectangle (the *fit box*) in user coordinates. The corners can also be specified in reverse order.

**optlist** An option list specifying processing options according to Table 8.2o.

**Returns** A string which specifies the reason for returning from the function:

- ▶ *\_stop*: all text in the textflow has been processed.
- ▶ *\_nextpage*: Waiting for the next page (caused by a form feed character U+000C). Another call to *PDF\_fit\_textflow()* is required for processing the remaining text.
- ▶ *\_boxfull*: No more space available in the fit box. Another call to *PDF\_fit\_textflow()* is required for processing the remaining text.
- ▶ Any other string: The string supplied to the *return* command in an inline option list.

If there are multiple simultaneous reasons for returning, the first in the list (from top to bottom) will be reported. The returned string is valid until the next call to this function.

**Details** This function leaves the text state unchanged. However, the current text position will be adjusted to point to the end of the generated text output.

**Scope** *page, pattern, template, glyph*

Table 8.2o Options for *PDF\_fit\_textflow()*

option	type	explanation
<i>blind</i>	<i>boolean</i>	No output will be generated, but all calculations will be performed and the formatting results can be checked with <i>PDF_info_textflow()</i> . Default: <i>false</i>
<i>rewind</i>	<i>integer:</i> -2, -1, 0, or 1	The state of the supplied textflow is reset to the state before some other call to <i>PDF_fit_textflow()</i> . Currently the following values are supported (default: 0): 1      Rewind to the state before the first call to <i>PDF_fit_textflow()</i> . 0      Don't reset the textflow. -1     Rewind to the state before the last call to <i>PDF_fit_textflow()</i> . -2     Rewind to the state before the second last call to <i>PDF_fit_textflow()</i> .
<i>showborder</i>	<i>boolean</i>	If true, the border of the fitbox will be stroked (using the current graphics state). This may be useful for development and debugging. Default: <i>false</i>

---

```
double PDF_info_textflow(PDF *p, int textflow, const char *keyword)
```

---

Query the current state of a textflow.

**textflow** A textflow handle returned by a call to *PDF\_create\_textflow()*.

**keyword** A keyword specifying the requested information:

- ▶ *boxlinecount*: Number of lines in the last fit box.
- ▶ *firstparalinecount*: The number of lines in the first paragraph of the fit box.
- ▶ *lastmark*: The number of the last mark found in the processed part of the textflow in the last fit box. Marks can be set with the *mark* option.



Table 8.21 Options for the `optlist` parameter of `PDF_create_textflow()` and inline option lists in the text

option	type	explanation
fontname	string	(Must be used with the encoding option) The name of the font.
autocidfont		These options will only be used if both of the fontname and encoding options are supplied. Notes: fontwarning Will be initialized with the value of the textwarning option.
autosubsetting		
embedding		
fontstyle		
fontwarning		
monospace		
subsetlimit		
subsetminsize		
subsetting		
unicodemap		

**All appearance options for `PDF_fit_textline()` (see Table 8.18):**

charref		These options will be initialized with the value of the corresponding parameter.
charspacing	font	
fillcolor	font	Will be ignored if both fontname and encoding are specified.
font	font	This option is required.
fontsize	glyphwarning	Will be initialized with the value of the textwarning option.
glyphwarning		
horizscaling		
Kerning		
overline		
strikeout		
strokecolor		
textformat		
textrendering		
textrise		
underline		
wordspacing		

**Options for controlling processing of inline options lists:**

begoptlistchar	integer or keyword	Unicode value of the character which starts inline option lists. Use the textlen option if you must use the start character literally in the text. The keyword none can be used to completely disable the search for option lists. Default: < (U+003C)
endoptlistchar	integer	Unicode value of the character which terminates inline option lists. The character } (U+007D) is not allowed. Default: > (U+003F)
textlen	integer or keyword	(Required for text in fonts which are not Unicode-compatible, or if text contains the begoptlistchar, or for text fragments with fixedtextformat=false and text-format=utf16xx in non-Unicode aware languages) Number of bytes or (in Unicode-aware languages) characters before the next inline option list. Default: 0

**Options for text semantics:**

charmapping <sup>1</sup>	list with pairs of two characters or a character and an integer	Replace individual character codes with one or more instances of another code. The option list contains one or more pairs of Unicode values. The first value of each pair will be replaced with the second value. In addition to numerical values character entity names (see Section 4.5.5, »Character References«, page 94) and the following symbolic names can be used (see Table 4.8): hortab, HT, linefeed, LF, verttab, VT, formfeed, FF, return, CR, newline, NEL, nbsp, NBSP, shy, SHY, linesep, LS, parasep, PS, CRLF. Instead of one-to-one mapping the second element in each pair may be an option list containing a character and a count: count > 0 The replacement code will be repeated count times. count < 0 A sequence of multiple instances of the character will be reduced to the absolute value of the specified number. count = 0 The code will be deleted.
--------------------------	---	---

Table 8.21 Options for the `optlist` parameter of `PDF_create_textflow()` and inline option lists in the text

option	type	explanation
<code>hyphenchar</code> <sup>1</sup>	integer	Unicode value of the character which replaces a soft hyphen at line breaks. Default: U+00AD (SOFT HYPHEN) if available in the font, U+002D (HYPHEN-MINUS) otherwise
<code>tabalignchar</code> <sup>1</sup>	integer	Unicode value of the character at which decimal tabs will be aligned. Default: the ' ' character (U+002E)
<b>Options for controlling the text layout:</b>		
<code>alignment</code>	keyword	Specifies formatting for lines in a paragraph (default: left): left left-aligned, starting at <code>leftindent</code> center centered between <code>leftindent</code> and <code>rightindent</code> right right-aligned, ending at <code>rightindent</code> justify left- and right-aligned
<code>avoid-emptybegin</code>	boolean	If true, empty lines at the beginning of a fitbox will be deleted. Default: false
<code>fixedleading</code>	boolean	If true, the first leading value found in each line will be used. Otherwise the maximum of all leading values in the line will be used. Default: false
<code>horthabsize</code> <sup>1</sup>	float or percentage	Width of a horizontal tab <sup>2</sup> . The interpretation depends on the <code>horthabmethod</code> option. Default: 7.5%
<code>horthab-method</code> <sup>1</sup>	keyword	Treatment of horizontal tabs in the text. If the determined position is to the left of the current text position, the tab will be ignored (default: relative): relative The position will be advanced by the amount specified in <code>horthabsize</code> . typewriter The position will be advanced to the next multiple of <code>horthabsize</code> . ruler The position will be advanced to the n-th tab value in the ruler option, where n is the number of tabs found in the line so far. If n is larger than the number of tab positions the relative method will be applied.
<code>lastalignment</code>	keyword	Formatting for the last line in a paragraph. All keywords of the alignment option are supported, plus the following (default: auto): auto Use the value of the alignment option unless it is justify. In the latter case left will be used.
<code>leading</code>	float or percentage	Distance between adjacent text baselines in user coordinates, or as a percentage of the font size. Default: 100%
<code>minlinecount</code>	integer	The minimum number of lines in the last paragraph in the fit box. If there are fewer lines they will be placed in the next fit box. The value 2 can be used to prevent single lines of a paragraph at the end of a fit box («orphans»). Default: 1
<code>parindent</code>	float or percentage	Left indent of the first line of a paragraph <sup>2</sup> . The amount will be added to <code>leftindent</code> . Specifying this option within a line will act like a tab. Default: 0
<code>rightindent</code> <code>leftindent</code>	float or percentage	Right or left indent of all text lines <sup>2</sup> . If <code>leftindent</code> is specified within a line and the determined position is to the left of the current text position, this option will be ignored for the current line. Default: 0
<code>ruler</code> <sup>1</sup>	list of floats or percentages	List of absolute tab positions for <code>horthabmethod=ruler</code> <sup>2</sup> . The list may contain up to 32 non-negative entries in ascending order. Default: integer multiples of <code>horthabsize</code>
<code>tabalignment</code> <sup>1</sup>	list of keywords	Alignment for tab stops. Each entry in the list defines the alignment for the corresponding entry in the ruler option (default: left) center Text will be centered at the tab position. decimal The first instance of <code>tabalignchar</code> will be left-aligned at the tab position. If no <code>tabalignchar</code> is found, right alignment will be used instead. left Text will be left-aligned at the tab position. right Text will be right-aligned at the tab position.

Table 8.21 Options for the `optlist` parameter of `PDF_create_textflow()` and inline option lists in the text

option	type	explanation
<b>Options for controlling the line-breaking algorithm:</b>		
<code>adjust-method</code>	keyword	Method used to adjust a line when a text portion doesn't fit into a line after compressing or expanding the distance between words subject to the limits specified by the <code>minspacing</code> and <code>maxspacing</code> options. Default: <code>auto</code>
	<code>auto</code>	The following methods are applied in order: <code>shrink</code> , <code>spread</code> , <code>nofit</code> , <code>split</code> .
	<code>clip</code>	Same as <code>nofit</code> , except that the long part at the right edge of the fit box (taking into account the <code>rightindent</code> option) will be clipped.
	<code>nofit</code>	The last word will be moved to the next line provided the remaining (short) line will not be shorter than the percentage specified in the <code>nofitlimit</code> option. Even justified paragraphs may look slightly ragged.
	<code>shrink</code>	If a word doesn't fit in the line the text will be compressed subject to <code>shrinklimit</code> . If it still doesn't fit the <code>nofit</code> method will be applied.
	<code>split</code>	The last word will not be moved to the next line, but will forcefully be hyphenated. For text fonts a hyphen character will be inserted, but not for symbol fonts.
	<code>spread</code>	The last word will be moved to the next line and the remaining (short) line will be justified by increasing the distance between characters in a word, subject to <code>spreadlimit</code> . If justification still cannot be achieved the <code>nofit</code> method will be applied.
<code>avoidbreak</code> <sup>1</sup>	boolean	If true, try to avoid any line breaks until <code>avoidbreak</code> is reset to false. Default: false
<code>maxspacing</code> <sup>1</sup> <code>minspacing</code> <sup>1</sup>	float or percentage	The maximum or minimum distance between words (in user coordinates, or as a percentage of the width of the space character). The calculated word spacing is limited by the provided values (but the <code>wordspacing</code> option will still be added). Defaults: <code>minspacing=50%</code> , <code>maxspacing=500%</code>
<code>nofitlimit</code>	float or percentage	Lower limit for the length of a line with the <code>nofit</code> method (in user coordinates or as a percentage of the width of the fitbox). Default: 75%.
<code>shrinklimit</code>	percentage	Lower limit for compressing text with the <code>shrink</code> method; the calculated shrinking factor is limited by the provided value, but will be multiplied with the value of the <code>horzscaling</code> option. Default: 85%
<code>spreadlimit</code> <sup>1</sup>	float or percentage	Upper limit for the distance between two characters for the <code>spread</code> method (in user coordinates or as a percentage of the font size); the calculated character distance will be added to the value of the <code>charspacing</code> option. Default: 0
<b>Options which work as commands:</b>		
<code>comment</code>	string	Arbitrary text which will be ignored; useful for commenting option lists or macros
<code>mark</code>	integer	Store the supplied number internally as a mark. The mark which has been stored least recently can later be retrieved with <code>PDF_info_textflow()</code> . This may be useful for determining which portions of text have already been placed on the page.
<code>nextline</code> <code>nextparagraph</code>	boolean	Force a new line or paragraph, even in fonts which are not Unicode-compatible.
<code>resetfont</code>	boolean	Reset the font and fontsize options to their previous values. This may be useful to reset the font after inserts, such as italic text. The font option has precedence over this option. Initially this command doesn't have any effect.
<code>return</code>	string	Exit <code>PDF_create_textflow()</code> with the supplied string as return value. The string must not start with an underscore <code>_</code> character.
<code>space</code>	float or percentage	The text position will be advanced by the provided value specified in user coordinates, or as a percentage of the font size. This also works in fonts which are not Unicode-compatible.

1. This option does not affect text in fonts which are not Unicode-compatible according to Section 4.5.6, »Unicode-compatible Fonts«, page 95.

2. In user coordinates, or as a percentage of the width of the fit box

**Macros for textflow options.** Option lists for textflows (either inline in the text, or directly in the call to `PDF_create_textflow()`) may contain macro definitions and macro calls according to Table 8.22. Macros may be useful for having a central definition of multiply used option values, such as font names, indentation amounts, etc. Before parsing an option list each contained macros will be substituted with the contents of the corresponding option list provided in the macro definition. The resulting option list will then be parsed. The following example demonstrates a macro definition for two macros:

```
<macro {  
    H1 {fontname=Helvetica-Bold encoding=winansi fontsize=14 }  
    body {fontname=Helvetica encoding=winansi fontsize=12 }  
}>
```

These macros could be used as follows in an option list:

```
<&H1>Chapter 1  
<&body>This chapter talks about...
```

The following rules apply to macro definition and use:

- ▶ Macros may be nested to an arbitrary depth (macro definitions may contain calls to other macros).
- ▶ Macros can not be used directly in the same option list where they are defined except within other macro definitions. However, a new option list which uses the macro can be started immediately after the end of the option list in which the macro is defined.
- ▶ Macro names are case-insensitive.
- ▶ Undefined macros will result in an exception.
- ▶ Macros can be redefined at any time.

Table 8.22 Option list macro definitions and calls for `PDF_fit_textflow()`

option	type	explanation
macro	list of pairs	Each pair describes the name and definition of a macro as follows: name (string) The name of the macro which can later be used for macro calls. Macros which have already been defined can be redefined later. The special name comment will be ignored. suboptlist An option list which will literally replace the macro name when the macro is called. Leading and trailing whitespace will be ignored.
&name	-	The macro with the specified name will be expanded, and the macro name (including the & character) will be replaced by the macro's contents, i.e. the suboptlist which has been defined for the macro (without the surrounding braces). The macro name is terminated by whitespace, {, }, =, or &. Therefore, these characters can not be used as part of a macro name.  Nested macros will be expanded without any nesting limit. Macros contained in string options will also be expanded. Macro substitution must result in a valid option list.

## 8.4 Graphics Functions

### 8.4.1 Graphics State Functions

All graphics state parameters are restored to their default values at the beginning of a page. The default values are documented in the respective function descriptions. Functions related to the text state are listed in Section 8.3, »Text Functions«, page 204.

*Note* None of the graphics state functions must be used in path scope (see Section 3.2, »Page Descriptions«, page 57).

---

**void PDF\_setdash(PDF \*p, double b, double w)**

---

Set the current dash pattern.

**b, w** The number of alternating black and white units. *b* and *w* must be non-negative numbers.

*Details* In order to produce a solid line, set *b* = *w* = 0. The dash parameter is set to solid at the beginning of each page.

*Scope* page, pattern, template, glyph

---

**void PDF\_setdashpattern(PDF \*p, const char \*optlist)**

---

Set a dash pattern defined by an option list.

**optlist** An option list according to Table 8.23. An empty list will generate a solid line.

Table 8.23 Options for PDF\_setdashpattern()

option	type	description
dasharray	list of floats	A list of 2-8 alternating values for the lengths of dashes and gaps for stroked paths (measured in the user coordinate system) . The array values must be non-negative, and not all zero. The array values will be cyclicly reused until the complete path is stroked.
dashphase	float	Distance into the dash pattern at which to start the dash. Default: 0

*Details* The dash parameter is set to a solid line at the beginning of each page.

*Scope* page, pattern, template, glyph

---

**void PDF\_setflat(PDF \*p, double flatness)**

---

Set the flatness parameter.

**flatness** A positive number which describes the maximum distance (in device pixels) between the path and an approximation constructed from straight line segments.

*Details* The flatness parameter is set to the default value of 1 at the beginning of each page.

*Scope* page, pattern, template, glyph



**void PDF\_setlinejoin(PDF \*p, int linejoin)**




Set the linejoin parameter.

**linejoin** Specifies the shape at the corners of paths that are stroked, see Table 8.24.

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

*Scope* *page, pattern, template, glyph*

Table 8.24 Values of the linejoin parameter

value	description (from the PDF reference)	examples
0	Miter joins: the outer edges of the strokes for the two segments are continued until they meet. If the extension projects too far, as determined by the miter limit, a bevel join is used instead.	
1	Round joins: a circular arc with a diameter equal to the line width is drawn around the point where the segments meet and filled in, producing a rounded corner.	
2	Bevel joins: the two path segments are drawn with butt end caps (see the discussion of linecap parameter), and the resulting notch beyond the ends of the segments is filled in with a triangle.	

**void PDF\_setlinecap(PDF \*p, int linecap)**




Set the linecap parameter.

**linecap** Controls the shape at the end of a path with respect to stroking, see Table 8.25.

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

*Scope* *page, pattern, template, glyph*

Table 8.25 Values of the linecap parameter

value	description (from the PDF reference)	examples
0	Butt end caps: the stroke is squared off at the endpoint of the path.	
1	Round end caps: a semicircular arc with a diameter equal to the line width is drawn around the endpoint and filled in.	
2	Projecting square end caps: the stroke extends beyond the end of the line by a distance which is half the line width and is squared off.	

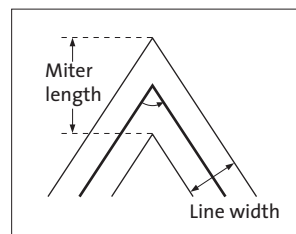
---

**void PDF\_setmiterlimit(PDF \*p, double miter)**

---

Set the miter limit.

**miter** A value greater than or equal to 1 which controls the spike produced by miter joins.



**Details** If the `linejoin` parameter is set to 0 (miter join), two line segments joining at a small angle will result in a sharp spike. This spike will be replaced by a straight end (i.e., the miter join will be changed to a bevel join) when the ratio of the miter length and the line width exceeds the miter limit. The miter limit is set to the default value of 10 at the beginning of each page. This corresponds to an angle of roughly 11.5 degrees.

**Scope** *page, pattern, template, glyph*

---

**void PDF\_setlinewidth(PDF \*p, double width)**

---

Set the current line width.

**width** The line width in units of the current user coordinate system.

**Details** The `width` parameter is set to the default value of 1 at the beginning of each page.

**Scope** *page, pattern, template, glyph*

---

**void PDF\_initgraphics(PDF \*p)**

---

Reset all color and graphics state parameters to their defaults.

**Details** The color, linewidth, linecap, linejoin, miterlimit, dash parameter, and the current transformation matrix (but not the text state parameters) are reset to their respective defaults. The current clipping path is not affected.

This function may be useful in situations where the program flow doesn't allow for easy use of `PDF_save()`/`PDF_restore()`.

**Scope** *page, pattern, template, glyph*

## 8.4.2 Saving and Restoring Graphics States

---

**void PDF\_save(PDF \*p)**

---

Save the current graphics state to a stack.

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

- ▶ graphics parameters: clipping path, coordinate system, current point, flatness, line cap style, dash pattern, line join style, line width, miter limit;

- ▶ color parameters: fill and stroke colors;
- ▶ text position and most text-related parameters, see list below;
- ▶ some PDFlib parameters, see list below.

Pairs of `PDF_save()` and `PDF_restore()` may be nested. Although the PDF specification doesn't limit the nesting level of save/restore pairs, applications must keep the nesting level below 26 in order to avoid printing problems caused by restrictions in the Post-Script output produced by PDF viewers, and to allow for additional save levels required by PDFlib internally.

- Scope

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

The following parameters are subject to save/restore: *charspacing, wordspacing, horz-scaling, italicangle, leading, font, fontsize, textrendering, textrise*;  
The following parameters are not subject to save/restore: *fillrule, kerning, underline, overline, strikeout*.

---

**void PDF\_restore(PDF \*p)**


---

Restore the most recently saved graphics state from the stack.

- Details

The corresponding graphics state must have been saved on the same page, pattern, or template.
- Scope

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

### 8.4.3 Coordinate System Transformation Functions

All transformation functions (`PDF_translate()`, `PDF_scale()`, `PDF_rotate()`, `PDF_skew()`, `PDF_concat()`, `PDF_setmatrix()`, and `PDF_initgraphics()`) change the coordinate system used for drawing subsequent objects. They do not affect existing objects on the page at all.

---

**void PDF\_translate(PDF \*p, double tx, double ty)**


---

Translate the origin of the coordinate system.

**tx, ty** The new origin of the coordinate system is the point *(tx, ty)*, measured in the old coordinate system.

- Scope

*page, pattern, template, glyph*

---

**void PDF\_scale(PDF \*p, double sx, double sy)**


---

Scale the coordinate system.

**sx, sy** Scaling factors in *x* and *y* direction.

*Details* This function scales the coordinate system by *sx* and *sy*. It may also be used for achieving a reflection (mirroring) by using a negative scaling factor. One unit in the *x* direction in the new coordinate system equals *sx* units in the *x* direction in the old coordinate system; analogous for *y* coordinates.

*Scope* *page, pattern, template, glyph*

---

**void PDF\_rotate(PDF \*p, double phi)**

---

Rotate the coordinate system.

**phi** The rotation angle in degrees.

*Details* Angles are measured counterclockwise from the positive *x* axis of the current coordinate system. The new coordinate axes result from rotating the old coordinate axes by *phi* degrees.

*Scope* *page, pattern, template, glyph*

---

**void PDF\_skew(PDF \*p, double alpha, double beta)**

---

Skew the coordinate system.

**alpha, beta** Skewing angles in *x* and *y* direction in degrees.

*Details* Skewing (or shearing) distorts the coordinate system by the given angles in *x* and *y* direction. *alpha* is measured counterclockwise from the positive *x* axis of the current coordinate system, *beta* is measured clockwise from the positive *y* axis. Both angles must be in the range  $-360^\circ < \alpha, \beta < 360^\circ$ , and must be different from  $-270^\circ$ ,  $-90^\circ$ ,  $90^\circ$ , and  $270^\circ$ .

*Scope* *page, pattern, template, glyph*

---

**void PDF\_concat(PDF \*p, double a, double b, double c, double d, double e, double f)**

---

Concatenate a matrix to the current transformation matrix.

**a, b, c, d, e, f** Elements of a transformation matrix. The six values make up a matrix in the same way as in PostScript and PDF (see references). In order to avoid degenerate transformations,  $a*d$  must not be equal to  $b*c$ .

*Details* This function concatenates a matrix to the current transformation matrix (CTM) for text and graphics. It allows for the most general form of transformations. Unless you are familiar with the use of transformation matrices, the use of *PDF\_translate()*, *PDF\_scale()*, *PDF\_rotate()*, and *PDF\_skew()* is suggested instead of this function. The CTM is reset to the identity matrix  $[1, 0, 0, 1, 0, 0]$  at the beginning of each page.

*Scope* *page, pattern, template, glyph*

void PDF\_setmatrix(PDF \*p, double a, double b, double c, double d, double e, double f)

Explicitly set the current transformation matrix.

**a, b, c, d, e, f** See *PDF\_concat()*.

*Details* This function is similar to *PDF\_concat()*. However, it disposes of the current transformation matrix, and completely replaces it with the new matrix.

*Scope* *page, pattern, template, glyph*

8.4.4 Explicit Graphics States

int PDF\_create\_gstate(PDF \*p, const char \*optlist)

Create a graphics state object subject to various options.

**optlist** An option list containing options for graphics states according to Table 8.26.

*Returns* A gstate handle that can be used in subsequent calls to *PDF\_set\_gstate()* during the enclosing *document* scope.

*Details* The option list may contain any number of graphics state parameters. Not all parameters are allowed for all PDF versions. The table lists the minimum required PDF version.

*Scope* *document, page, pattern, template, glyph*

*PDF/X* The *opacityfill* and *opacitystroke* options must be avoided unless they have a value of 1.

Table 8.26 Options for *PDF\_create\_gstate()*

key	type	explanation and possible values
alphaishshape	boolean	(PDF 1.4) sources of alpha are treated as shape (true) or opacity (false). Default: false
blendmode	keyword list	(PDF 1.4) Name of the blend mode. Multiple blend modes can be specified. Possible values: None, Normal, Multiply, Screen, Overlay, Darken, Lighten, ColorDodge, ColorBurn, HardLight, SoftLight, Difference, Exclusion, Hue, Saturation, Color, Luminosity. Default: None
flatness	float	maximum distance between a path and its approximation (see <i>PDF_setflat()</i> ), must be > 0.
linecap	integer	shape at the end of a path(see <i>PDF_setlinecap()</i> ); must be 0, 1, or 2.
linejoin	integer	shape at the corners of paths (see <i>PDF_setlinejoin()</i> ); must be 0, 1, or 2
linewidth	float	line width (see <i>PDF_setlinewidth()</i> ); must be > 0
miterlimit	float	controls the spike produced by miter joins, which must be >= 1 (see <i>PDF_setmiterlimit()</i> )
opacityfill	float	(PDF 1.4) constant alpha for fill operations; must be >= 0 and <= 1.
opacitystroke	float	(PDF 1.4) constant alpha for stroke operations; must be >=0 and <=1
overprintfill	boolean	overprint for operations other than stroke. Default: false
overprintmode	integer	overprint mode. 0 (default) means that each color component replaces previously placed marks; mode 1 (called »overprinting default is nonzero overprinting« in Acrobat) means that a color component of 0 leaves the corresponding component unchanged.
overprintstroke	boolean	overprint for stroke operations. Default: false

Table 8.26 Options for `PDF_create_gstate()`

key	type	explanation and possible values
<code>renderingintent</code>	keyword	color rendering intent used for gamut compression; possible keywords: <i>Auto</i> , <i>AbsoluteColorimetric</i> , <i>RelativeColorimetric</i> , <i>Saturation</i> , <i>Perceptual</i>
<code>smoothness</code>	float	maximum error of a linear interpolation for a shading; must be $\geq 0$ and $\leq 1$
<code>strokeadjust</code>	boolean	whether or not to apply automatic stroke adjustment. Default: <i>false</i>
<code>textknockout</code>	boolean	(PDF 1.4) with respect to compositing, glyphs in a text object will be treated as separate objects ( <i>false</i> ) or as a single object ( <i>true</i> ). Default: <i>true</i>

---

**void PDF\_set\_gstate(PDF \*p, int gstate)**

---

Activate a graphics state object.

**gstate** A handle for a graphics state object retrieved with `PDF_create_gstate()`.

**Details** All options contained in the graphics state object will be set. Graphics state options accumulate when this function is called multiply. Options which are not explicitly set in the graphics state object will keep their current values. All graphics state options will be reset to their default values at the beginning of a page.

**Scope** *page, pattern, template, glyph*

### 8.4.5 Path Construction

Table 8.27 lists relevant parameters and values for this section.

Table 8.27 Parameters and values for path segment functions (see Section 8.2.3, »Parameter Handling«, page 198)

function	key	explanation
<code>get_value</code>	<code>currentx</code> <code>currenty</code>	The x or y coordinate (in units of the current coordinate system), respectively, of the current point. Scope: <i>page, pattern, template, path</i>

**Note** Make sure to call one of the functions in Section 8.4.6, »Path Painting and Clipping«, page 233, after using the functions in this section, or the constructed path will have no effect, and subsequent operations may raise a PDFlib exception.

---

**void PDF\_moveto(PDF \*p, double x, double y)**

---

Set the current point for graphics output.

**x, y** The coordinates of the new current point.

**Details** The current point is set to the default value of *undefined* at the beginning of each page. The current points for graphics and the current text position are maintained separately.

**Scope** *page, pattern, template, path, glyph*; this function starts *path* scope.

**Params** *currentx, currenty*

---

**void PDF\_lineto(PDF \*p, double x, double y)**

---

Draw a line from the current point to another point.

**x, y** The coordinates of the second endpoint of the line.

*Details* This function adds a straight line from the current point to  $(x, y)$  to the current path. The current point must be set before using this function. The point  $(x, y)$  becomes the new current point.

The line will be centered around the »ideal« line, i.e. half of the linewidth (as determined by the value of the linewidth parameter) will be painted on each side of the line connecting both endpoints. The behavior at the endpoints is determined by the value of the linecap parameter.

*Scope* *path*

*Params* *currentx, currenty*

---

**void PDF\_curveto(PDF \*p, double x1, double y1, double x2, double y2, double x3, double y3)**

---

Draw a Bézier curve from the current point, using three more control points.

**x1, y1, x2, y2, x3, y3** The coordinates of three control points.

*Details* A Bézier curve is added to the current path from the current point to  $(x_3, y_3)$ , using  $(x_1, y_1)$  and  $(x_2, y_2)$  as control points. The current point must be set before using this function. The endpoint of the curve becomes the new current point.

*Scope* *path*

*Params* *currentx, currenty*

---

**void PDF\_circle(PDF \*p, double x, double y, double r)**

---

Draw a circle.

**x, y** The coordinates of the center of the circle.

**r** The radius of the circle.

*Details* This function adds a circle to the current path as a complete subpath. The point  $(x + r, y)$  becomes the new current point. The resulting shape will be circular in user coordinates. If the coordinate system has been scaled differently in  $x$  and  $y$  directions, the resulting curve will be elliptical.

*Scope* *page, pattern, template, path, glyph*; this function starts *path* scope.

*Params* *currentx, currenty*

---

**void PDF\_arc(PDF \*p, double x, double y, double r, double alpha, double beta)**

---

Draw a counterclockwise circular arc segment.

**x, y** The coordinates of the center of the circular arc segment.

**r** The radius of the circular arc segment.  $r$  must be nonnegative.

**alpha, beta** The start and end angles of the circular arc segment in degrees.

*Details* This function adds a counterclockwise circular arc segment to the current path, extending from *alpha* to *beta* degrees. For both *PDF\_arc()* and *PDF\_arcn()*, angles are measured counterclockwise from the positive x axis of the current coordinate system. If there is a current point an additional straight line is drawn from the current point to the starting point of the arc. The endpoint of the arc becomes the new current point.

The arc segment will be circular in user coordinates. If the coordinate system has been scaled differently in x and y directions the resulting curve will be elliptical.

*Scope* *page, pattern, template, path, glyph*; this function starts *path* scope.

*Params* *currentx, currenty*

---

**void PDF\_arcn(PDF \*p, double x, double y, double r, double alpha, double beta)**

---

Draw a clockwise circular arc segment.

*Details* Except for the drawing direction, this function behave exactly like *PDF\_arc()*. In particular, the angles are still measured *counterclockwise* from the positive x axis.

---

**void PDF\_rect(PDF \*p, double x, double y, double width, double height)**

---

Draw a rectangle.

**x, y** The coordinates of the lower left corner of the rectangle.

**width, height** The size of the rectangle.

*Details* This function adds a rectangle to the current path as a complete subpath. Setting the current point is not required before using this function. The point (x, y) becomes the new current point. The lines will be centered around the »ideal« line, i.e. half of the linewidth (as determined by the value of the linewidth parameter) will be painted on each side of the line connecting the respective endpoints.

*Scope* *page, pattern, template, path, glyph*; this function starts *path* scope.

*Params* *currentx, currenty*

---

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

---

Close the current path.

*Details* This function closes the current subpath, i.e., adds a line from the current point to the starting point of the subpath.

*Scope* *path*

*Params* *currentx, currenty*



# 8.4.6 Path Painting and Clipping

Table 8.28 lists relevant parameters and values for this section.

Table 8.28 Parameters and values for path painting and clipping functions

function	key	explanation
set_parameter	fillrule	Set the current fill rule to winding or evenodd. The fill rule is used by PDF viewers to determine the interior of shapes for the purpose of filling or clipping. Since both algorithms yield the same result for simple shapes, most applications won't have to change the fill rule. The fill rule is reset to the default of winding at the beginning of each page. Scope: page, pattern, template, glyph.

*Note* Most functions in this section clear the path, and leave the current point undefined. Subsequent drawing operations must therefore explicitly set the current point (e.g., using PDF\_moveto()) after one of these functions has been called.

void PDF\_stroke(PDF \*p)

Stroke the path with the current line width and current stroke color, and clear it.

*Scope* path; this function terminates path scope.

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

Close the path, and stroke it.

*Details* This function closes the current subpath (adds a straight line segment from the current point to the starting point of the path), and strokes the complete current path with the current line width and the current stroke color.

*Scope* path; this function terminates path scope.

void PDF\_fill(PDF \*p)

Fill the interior of the path with the current fill color.

*Details* This function fills the interior of the current path with the current fill color. The interior of the path is determined by one of two algorithms (see the fillrule parameter). Open paths are implicitly closed before being filled.

*Scope* path; this function terminates path scope.

*Params* fillrule

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

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

*Scope* path; this function terminates path scope.

*Params* fillrule

---

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

---

Close the path, fill, and stroke it.

*Details* This function closes the current subpath (adds a straight line segment from the current point to the starting point of the path), and fills and strokes the complete current path.

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

*Params* *fillrule*

---

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

---

Use the current path as clipping path, and terminate the path.

*Details* This function uses the intersection of the current path and the current clipping path as the clipping path for subsequent operations. The clipping path is set to the default value of the page size at the beginning of each page. The clipping path is subject to *PDF\_save()*/*PDF\_restore()*. It can only be enlarged by means of *PDF\_save()*/*PDF\_restore()*.

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

---

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

---

End the current path without filling or stroking it.

*Details* This function doesn't have any visible effect on the page. It generates an invisible path on the page.

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

## 8.4.7 Layer Functions

---

**int PDF\_define\_layer(PDF \*p, const char \*name, int len, const char \*optlist)**

---

Create a new layer definition (requires PDF 1.5).

**name** (Hypertext string) The name of the layer.

**len** (C binding only) Length of *name* (in bytes) for UCS-2 strings. If *len* = 0 a null-terminated string must be provided.

**optlist** An option list specifying layer settings according to Table 8.29.

*Returns* A layer handle which can be used in calls to *PDF\_begin\_layer()* and *PDF\_set\_layer\_dependency()* until the end of the enclosing *document* scope.

*Details* PDFlib will issue a warning if a layer was defined but hasn't been used in the document.

*Scope* *document, page*

Table 8.29 Options for PDF\_define\_layer()

option	type	explanation
creatorinfo	option list	An option list describing the content and the creating application. Both of the following entries are required if this option is used: creator (Hypertext string) The name of the application which created the layer subtype (string) The type of content. Suggested values are Artwork and Technical.
defaultstate	boolean	Default: true
hypertext-encoding	keyword	Specifies the encoding for the name parameter and the creator option (see Section 4.5.4, »String Handling in non-Unicode-capable Languages«, page 91). An empty string is equivalent to unicode. Default: the global hypertextencoding parameter
hypertext-format	keyword	Set the format for the name parameter. Possible values are bytes, utf8, utf16, utf16le, utf16be, and auto. Default: the value of the hypertextformat parameter
initial-exportstate	boolean	Specifies the layer's recommended export state. If true, Acrobat will include the layer when converting/exporting to older PDF versions or other document formats. Default: true
initial-printstate	boolean	The layer's recommended printing state. If true, Acrobat will include the layer when printing the document. Default: true
initial-viewstate	boolean	The layer's recommended viewing state. If true, Acrobat will display the layer when opening the document. Default: true
intent	keyword	View or Design
language	option list	Specifies the language of the layer: lang (string; required) The language and possibly locale in the format described in Table 8.4 for the lang option preferred (boolean) If true this layer is used if there is only a partial match between the layer and the system language. Default: false
onpanel	boolean	If false, the layer name will not be visible in Acrobat's layer panel, and therefore cannot be manipulated by the user. Default: true
pageelement	keyword	Specifies that the layer contains a pagination artifact: one of HF (header/footer), FG (foreground image or graphic), BG (background image or graphic), or L (logo).
printsyntax	option list	Specifies whether the layer is intended for printing: subtype (keyword) One of Trapping, PrintersMarks, or Watermark specifying the kind of content in the layer. printstate (boolean) If true, Acrobat will activate the layer contents upon printing.
zoom	list of floats or percentages	One or two values specifying the layer's visibility depending on the zoom factor (1.0 means a zoom factor of 100 percent). If one value is provided, it will be used as the maximum zoom factor at which the layer should be visible; if two values are provided they specify the minimum and maximum zoom factor. The keyword maxzoom can be used to specify the largest possible zoom factor.

**void PDF\_set\_layer\_dependency(PDF \*p, const char \*type, const char \*optlist)**

Define hierarchical and group relationships among layers (requires PDF 1.5).

- type** The type of dependency, which must be one of the following:
- ▶ *GroupAllOn*: The layer specified in the *depend* option will be visible if all layers specified in the *group* option are visible.
  - ▶ *GroupAnyOn*: The layers specified in the *depend* option will be visible if any layer specified in the *group* option is visible.
  - ▶ *GroupAllOff*: The layer specified in the *depend* option will be visible if all layers specified in the *group* option are invisible.

- ▶ *GroupAnyOff*: The layer specified in the *depend* option will be visible if any layer specified in the *group* option is invisible.
- ▶ *Parent*: Specify a hierarchical relationship between the layer specified in the *parent* option and the layers specified in the *children* option. A layer can not belong to more than one parent layer.
- ▶ *Radiobtn*: Specify a radiobutton relationship between the layers specified in the *group* option. This means that at most one layer in the group will be visible at a time, which is particularly useful for multiple language layers.
- ▶ *Title*: The layer handle specified in the *parent* option does not control any page contents directly, but serves as the parent layer node for the layers specified in the *children* option.

**optlist** An option list specifying layer dependencies according to Table 8.30.

Scope *document, page*

Table 8.30 Options for `PDF_set_layer_dependency()`

option	type	explanation
parent	layer handle	(Only for type=Parent and Title) The layer which is the parent of the layers specified in the children option.
children	list of layer handles	(Only for type=Parent and Title) One or more layer handles specifying the layers subordinate to the provided parent layer.
depend	layer handle	(Only for type=GroupAllOn, GroupAnyOn, GroupAllOff, and GroupAnyOff) The layer which is controlled by the layers specified in the group option.
group	list of layer handles	(Only for type=GroupAllOn, GroupAnyOn, GroupAllOff, GroupAnyOff, and Radiobtn) One or more layer handles comprising the group.

---

**void PDF\_begin\_layer(PDF \*p, int layer)**

---

Start a layer for subsequent output on the page (requires PDF 1.5).

**layer** The layer’s handle, which must have been retrieved with `PDF_define_layer()`.

*Details* All content placed on the page after this call, but before any subsequent call to `PDF_begin_layer()` or `PDF_end_layer()` will be part of the specified layer. The content’s visibility depends on the layer’s settings.

This function activates the specified layer, and deactivates any layer which may be currently active.

Scope *page*

---

**void PDF\_end\_layer(PDF \*p)**

---

Deactivate all active layers (requires PDF 1.5).

*Details* Content placed on the page after this call will not belong to any layer. All layers must be closed at the end of a page.

Scope *page*

# 8.5 Color Functions

## 8.5.1 Setting Color and Color Space

Table 8.31 lists relevant parameters and values for this section.

Table 8.31 Parameter for color functions

function	key	explanation
set_parameter	preserveold-pantonenames	If false, old-style Pantone spot color names will be converted to the corresponding new color names, otherwise they will be preserved. Default: false. Scope: any
set_parameter	spotcolor-lookup	If false, PDFlib will not use its internal database of spot color names. This can be used to provide custom definitions of known spot colors, which may be required as a workaround to match the definitions used by other applications. This feature should be used with care, and is not recommended. Default: true. Scope: any

```
void PDF_setcolor(PDF *p,  
    const char *fstype, const char *colorspace, double c1, double c2, double c3, double c4)
```

Set the current color space and color.

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

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

**c1, c2, c3, c4** Color components for the chosen color space (see Section 3.3, »Working with Color«, page 63):

- ▶ If *colorspace* is *gray*, *c1* specifies a gray value;
- ▶ If *colorspace* is *rgb*, *c1*, *c2*, *c3* specify red, green, and blue values;
- ▶ If *colorspace* is *cmyk*, *c1*, *c2*, *c3*, *c4* specify cyan, magenta, yellow, and black values;
- ▶ If *colorspace* is *spot*, *c1* specifies a spot color handle returned by *PDF\_makespotcolor()*, and *c2* specifies a tint value between 0 and 1;
- ▶ If *colorspace* is *pattern*, *c1* specifies a pattern handle returned by *PDF\_begin\_pattern()* or *PDF\_shading\_pattern()*.
- ▶ If *colorspace* is *iccbasedgray*, *c1* specifies a gray value;
- ▶ If *colorspace* is *iccbasedrgb*, *c1*, *c2*, *c3* specify red, green, and blue values;
- ▶ If *colorspace* is *iccbasedcmyk*, *c1*, *c2*, *c3*, *c4* specify cyan, magenta, yellow, and black values;
- ▶ If *colorspace* is *lab*, *c1*, *c2*, and *c3* specify color values in the CIE L\*a\*b\* color space, interpreted with the D50 illuminant. *c1* specifies the L\* (luminance) in the range 0 to 100, and *c2*, *c3* specify the a\*, b\* (chrominance) values in the range -127 to 128.

**Details** All color values for the *gray*, *rgb*, and *cmyk* color spaces and the *tint* value for the *spot* color space must be numbers in the inclusive range 0–1. Unused parameters should be set to 0.

Grayscale, RGB values and spot color tints are interpreted according to additive color mixture, i.e., 0 means no color and 1 means full intensity. Therefore, a gray value of 0 and RGB values with  $(r, g, b) = (0, 0, 0)$  mean black; a gray value of 1 and RGB values with  $(r, g, b) = (1, 1, 1)$  mean white. CMYK values, however, are interpreted according to subtrac-

tive color mixture, i.e.,  $(c, m, y, k) = (0, 0, 0, 0)$  means white and  $(c, m, y, k) = (0, 0, 0, 1)$  means black. Color values in the range 0–255 must be scaled to the range 0–1 by dividing by 255.

The fill and stroke color values for the *gray*, *rgb*, and *cmymk* color spaces are set to a default value of black at the beginning of each page. There are no defaults for spot and pattern colors.

If the *iccbasedgray/rgb/cmyk* color spaces are used, the corresponding ICC profile must have been set before using the *setcolor:iccprofilegray/rgb/cmyk* parameters (see Table 8.33).

*Scope* *page*, *pattern* (only if the pattern’s paint type is 1), *template*, *glyph* (only if the Type 3 font’s *colorized* option is true), *document*; a pattern color can not be used within its own definition. Setting the color in *document* scope may be useful for defining spot colors with *PDF\_makespotcolor()*.

*PDF/X* PDF/X-1 and PDF/X-1a: *colorspace* = *rgb*, *iccbasedgray/rgb/cmyk*, and *lab* are not allowed. PDF/X-3: *colorspace* = *gray* requires that the *defaultgray* option in *PDF\_begin\_page\_ext()* has been set unless the PDF/X output intent is a grayscale or CMYK device. *colorspace* = *rgb* requires that the *defaultrgb* option in *PDF\_begin\_page\_ext()* has been set unless the PDF/X output intent is an RGB device. *colorspace* = *cmyk* requires that the *defaultcmyk* option in *PDF\_begin\_page\_ext()* has been set unless the PDF/X output intent is a CMYK device. Using *iccbasedgray/rgb/cmyk* and *lab* color requires an ICC profile in the output intent (a standard name is not sufficient in this case).

*Params* *setcolor:iccprofilegray/rgb/cmyk*

---

**int PDF\_makespotcolor(PDF \*p, const char \*spotname, int reserved)**

---

Find a built-in spot color name, or make a named spot color from the current fill color.

**spotname** The name of a built-in spot color, or an arbitrary name for the spot color to be defined. This name is restricted to a maximum length of 126 bytes. Only 8-bit characters are supported in the spot color name; Unicode or embedded null characters are not supported.

**reserved** (C language binding only.) Reserved, must be 0.

*Returns* A color handle which can be used in subsequent calls to *PDF\_setcolor()* throughout the document. Spot color handles can be reused across all pages, but not across documents. There is no limit for the number of spot colors in a document.

*Details* If *spotname* is known in the internal color tables and the *spotcolorlookup* parameter is *true* (which is default), the supplied spot color name will be used. Otherwise the (CMYK or other) color values of the current fill color will be used to define the appearance of a new spot color. These alternate values will only be used for screen preview and low-end printing. The supplied spot color name will be used for producing color separations instead of the alternate values.

If *spotname* has already been used in a previous call to *PDF\_makespotcolor()*, the return value will be the same as in the earlier call, and will not reflect the current color.

The special spot color name *All* can be used to apply color to all color separations, which is useful for painting registration marks. A spot color name of *None* will produce no visible output on any color separation.

- Scope** *page, pattern, template, glyph* (only if the Type 3 font's *colorized* option is true), *document*; the current fill color must not be a spot color or pattern if a custom color is to be defined.
- PDF/X** PANTONE® colors are not supported in the PDF/X-1 and PDF/X-1a modes.
- Params** *spotcolorlookup, preserveoldpantonenames*

---

**int PDF\_load\_iccprofile(PDF \*p, const char \*profilename, int len, const char \*optlist)**

---

Search for an ICC profile, and prepare it for later use.

**profilename** (Name string) The name of an *ICCProfile* resource, a disk-based or virtual file name, or a standard output condition name for PDF/X. The latter is only allowed if the *usage* option is set to *outputintent*.

**len** (Only for the C binding) Length of *profilename* (in bytes) for UTF-16 strings. If *len* = 0 a null-terminated string must be provided.

**optlist** An option list describing aspects of the profile handling according to Table 8.32.

Table 8.32 Options for *PDF\_load\_iccprofile()*

key	type	explanation and possible values
usage	keyword	Describes the intended use of the ICC profile (default: <i>iccbased</i> ):
		<i>iccbased</i> the ICC profile will be used to define an ICC-based color space, or will be applied to an image.
		<i>outputintent</i> the ICC profile will be used to define a PDF/X output intent.
description	string	This option is only used if <i>usage</i> = <i>outputintent</i> . It contains the human-readable description of the ICC profile which will be used along with the PDF/X output intent. Default: if <i>profilename</i> refers to a standard output intent, the description will be taken from an internal list; otherwise there will be no description.
embedprofile	boolean	This option is only used if <i>usage</i> = <i>outputintent</i> . Force an embedded ICC profile even if a standard output intent has been provided as <i>profilename</i> . Default: <i>false</i>

- Returns** A profile handle which can be used in subsequent calls to *PDF\_load\_image()* or for setting profile-related parameters. The return value must be checked for -1 (in PHP: 0) which signals an error. In order to get more detailed information about the nature of a profile-related problem (file not found, unsupported format, etc.) set the *iccwarning* parameter to *true*. The returned profile handle can not be reused across multiple PDF documents. Also, the returned handle can not be applied to an image if the *usage* option is *outputintent*. There is no limit to the number of ICC profiles in a document.
- Details** If the *usage* option is *iccbased* the named profile will be searched according to the search strategy discussed in Section 3.3.4, »Color Management and ICC Profiles«, page 67. If the profile is found, it will be checked whether it is suitable (e.g., number of color components). The *sRGB* profile is always available internally, and must not be configured.
- If *usage* option is *outputintent* the named profile is first searched in an internal list of standard output intents. If this search was unsuccessful, the name will be searched in the list of user-configured output intents. If the supplied name was found to be a standard output intent according to the built-in or user-configured list, no ICC profile will be searched, and the name supplied with the description option will be embedded in

the PDF output as the PDF/X output intent. If the name was not found to be a standard output intent identifier, it is treated as a profile name and the corresponding ICC profile will be embedded in the PDF as the PDF/X output intent.

**Scope** *document*; if the *usage* option is *iccbased* the following scopes are also allowed: *page*, *pattern*, *template*, *glyph*.

**Params** See Table 8.33.

**PDF/X** The output intent for the generated document must be set either using this function, or by copying an imported document's output intent using *PDF\_process\_pdi()*.

Table 8.33 Parameters and values for ICC profiles

function	key	explanation
set_parameter	ICCProfile Standard- OutputIntent	The corresponding resource file line as it would appear for the respective category in a UPR file (see Section 3.1.6, »Resource Configuration and File Searching«, page 51). Multiple calls add new entries to the internal list. (See also resourcefile in Table 8.2). Scope: any
set_parameter	iccwarning	Enable or suppress warnings (nonfatal exceptions) related to ICC profiles. Possible values are true and false, default value is false. Scope: any
get_value	iccomponents	Return the number of color components in the ICC profile referenced by the handle provided in the modifier.
set_value	setcolor:icc- profilegray	Set an ICC profile which specifies a Gray color space for use with PDF_setcolor(). Scope: document, page, pattern, template, glyph
set_value	setcolor:icc- profilergb	Set an ICC profile which specifies an RGB color space for use with PDF_setcolor(). Scope: document, page, pattern, template, glyph
set_value	setcolor:icc- profilecmyk	Set an ICC profile which specifies a CMYK color space for use with PDF_setcolor(). Scope: document, page, pattern, template, glyph
set_value	defaultgray defaultrgb defaultcmyk	Deprecated; use the defaultgray, defaultrgb, and defaultcmyk options in PDF_begin/end_page_ext()

## 8.5.2 Patterns and Shadings

---

**int PDF\_begin\_pattern(PDF \*p,  
double width, double height, double xstep, double ystep, int painttype)**

---

Start a pattern definition.

**width, height** The dimensions of the pattern's bounding box in points.

**xstep, ystep** The offsets when repeatedly placing the pattern to stroke or fill some object. Most applications will set these to the pattern *width* and *height*, respectively.

**painttype** If *painttype* is 1 the pattern must contain its own color specification which will be applied when the pattern is used; if *painttype* is 2 the pattern must not contain any color specification but instead the current fill or stroke color will be applied when the pattern is used for filling or stroking.

**Returns** A pattern handle that can be used in subsequent calls to *PDF\_setcolor()* during the enclosing *document* scope.



*Details* This function will reset all text, graphics, and color state parameters to their defaults. Hypertext functions and functions for opening images must not be used during a pattern definition, but all text, graphics, and color functions (with the exception of the pattern which is in the process of being defined) can be used.

*Scope* *document, page*; this function starts *pattern* scope, and must always be paired with a matching *PDF\_end\_pattern()* call.

**void PDF\_end\_pattern(PDF \*p)**

Finish a pattern definition.

*Scope* *pattern*; this function terminates *pattern* scope, and must always be paired with a matching *PDF\_begin\_pattern()* call.

**int PDF\_shading\_pattern(PDF \*p, int shading, const char \*optlist)**

Define a shading pattern using a shading object (requires PDF 1.4 or above).

**shading** A shading handle returned by *PDF\_shading()*.

**optlist** An option list describing aspects of the shading pattern according to Table 8.34.

*Returns* A pattern handle that can be used in subsequent calls to *PDF\_setcolor()* during the enclosing *document* scope.

*Details* This function can be used to fill arbitrary objects with a shading. To do so, a shading handle must be retrieved using *PDF\_shading()*, then a pattern must be defined based on this shading using *PDF\_shading\_pattern()*. Finally, the pattern handle can be supplied to *PDF\_setcolor()* to set the current color to the shading pattern.

*Scope* *document, page, font*

Table 8.34 Options for *PDF\_shading\_pattern()*

key	type	explanation and possible values
<i>gstate</i>	<i>handle</i>	A graphics state handle

**void PDF\_shfill(PDF \*p, int shading)**

Fill an area with a shading, based on a shading object (requires PDF 1.4 or above).

**shading** A shading handle returned by *PDF\_shading()*.

*Details* This function allows shadings to be used without involving *PDF\_shading\_pattern()* and *PDF\_setcolor()*. However, it works only for simple shapes where the geometry of the object to be filled is the same as that of the shading itself. Since the current clip area will be shaded (subject to the *extendo* and *extend1* options of the shading) this function will generally be used in combination with *PDF\_clip()*.

*Scope* *page, pattern* (only if the pattern's paint type is 1), *template, glyph* (only if the Type 3 font's *colorized* option is *true*), *document*

---

```
int PDF_shading(PDF *p, const char *shtype, double xo, double yo, double x1, double y1,  
double c1, double c2, double c3, double c4, const char *optlist)
```

---

Define a blend from the current fill color to another color (requires PDF 1.4 or above).

**shtype** The type of the shading; must be *axial* for linear shadings or *radial* for circle-like shadings.

**xo, yo, x1, y1** For axial shadings,  $(x_0, y_0)$  and  $(x_1, y_1)$  are the coordinates of the starting and ending points of the shading. For radial shadings these points specify the centers of the starting and ending circles.

**c1, c2, c3, c4** Color values of the shading's endpoint, interpreted in the current fill color space in the same way as the color parameters in *PDF\_setcolor()*. If the current fill color space is a spot color space *c1* will be ignored, and *c2* contains the tint value.

**optlist** An option list describing aspects of the shading according to Table 8.35.

**Returns** A shading handle that can be used in subsequent calls to *PDF\_shading\_pattern()* and *PDF\_shfill()* during the enclosing *document* scope.

**Details** The current fill color will be used as the starting color; it must not be based on a pattern.

**Scope** *document, page, font*

Table 8.35 Options for *PDF\_shading()*

key	type	explanation and possible values
<i>N</i>	<i>float</i>	The exponent for the color transition function; must be > 0. Default: 1
<i>ro</i>	<i>float</i>	(Only for radial shadings, and required in this case) Radius of the starting circle.
<i>r1</i>	<i>float</i>	(Only for radial shadings, and required in this case) The radius of the ending circle.
<i>extendo</i>	<i>boolean</i>	Specifies whether to extend the shading beyond the starting point. Default: false
<i>extend1</i>	<i>boolean</i>	Specifies whether to extend the shading beyond the endpoint. Default: false
<i>antialias</i>	<i>boolean</i>	Specifies whether to activate antialiasing for the shading. Default: false

# 8.6 Image and Template Functions

Table 8.36 lists relevant parameters and values for this section.

Table 8.36 Parameters and values for the image functions (see Section 8.2.3, »Parameter Handling«, page 198)

function	key	explanation
get_value	imagewidth imageheight	Get the width or height, respectively, of an image in pixels. The modifier is the integer handle of the selected image. Scope: page, pattern, template, glyph, document, path.
get_value	resx resy	Get the horizontal or vertical resolution of an image, respectively. The modifier is the integer handle of the selected image. Scope: page, pattern, template, glyph, document, path.  If the value is positive, the return value is the image resolution in pixels per inch (dpi). If the return value is negative, resx and resy can be used to find the aspect ratio of non-square pixels, but don't have any absolute meaning. If the return value is zero, the resolution of the image is unknown.
set_parameter	honoriccprofile	Read ICC color profiles embedded in images, and apply them to the image data. Default: true
set_parameter	imagewarning	This parameter can be used in order to obtain more detailed information about why an image couldn't be opened successfully with PDF_load_image(.). Scope: any true      Raise an exception when the image function fails, and return -1 (in PHP: o). The message supplied with the exception may be useful for debugging. false     Do not raise an exception when the image function fails. Instead, the function simply returns -1 (in PHP: o) on error. This is the default.
set_value	image:iccprofile	Handle for an ICC profile which will be applied to all respective images unless the iccprofile option is supplied.
get_value	image:iccprofile	Return a handle for the ICC profile embedded in the image referenced by the image handle provided in the modifier.
set_value	renderingintent	The rendering intent for images. See Table 3.7 for a list of possible keywords and their meaning. Default: Auto.

## 8.6.1 Images

```
int PDF_load_image(PDF *p,  
    const char *imagetype, const char *filename, int len, const char *optlist)
```

Open a disk-based or virtual image file subject to various options.

**imagetype**    The string *auto* instructs PDFlib to automatically detect the image file type (this is not possible for CCITT and raw images). Explicitly specifying the image format with the strings *bmp*, *ccitt*, *gif*, *jpeg*, *png*, *raw*, or *tiff* offers slight performance advantages (for details see Section 5.1.2, »Supported Image File Formats«, page 126). Type *ccitt* is different from a TIFF file which contains CCITT-compressed image data.

**filename**    (Name string) Generally the name of the image file to be opened. This must be the name of a disk-based or virtual file; PDFlib will not pull image data from URLs.

**len** (Only for the C binding.) Length of *filename* (in bytes) for UTF-16 strings. If *len* = 0 a null-terminated string must be provided.

**optlist** An option list specifying image-related properties according to Table 8.37.

**Returns** An image handle which can be used in subsequent image-related calls. The return value must be checked for -1 (in PHP: 0) which signals an error. In order to get more detailed information about the nature of an image-related problem (wrong image file name, unsupported format, bad image data, etc.), set the *imagewarning* parameter or option to *true* (see Table 8.36 and Table 8.37). The returned image handle can not be reused across multiple PDF documents.

**Details** This function opens and analyzes a raster graphics file in one of the supported formats as determined by the *imagetype* parameter, and copies the relevant image data to the output document. This function will not have any visible effect on the output. In order to actually place the imported image somewhere in the generated output document, *PDF\_fit\_image()* must be used. Opening the same image more than once is not recommended because the actual image data will be copied to the output document more than once.

PDFlib will open the image file with the provided *filename*, process the contents, and close the file before returning from this call. Although images can be placed multiply within a document (see *PDF\_fit\_image()*), the actual image file will not be kept open after this call.

If *imagetype* = *raw* or *ccitt*, the *width*, *height*, *components*, and *bpc* options must be supplied since PDFlib cannot deduce those from the image data. The user is responsible for supplying option values which actually match the image. Otherwise corrupt PDF output may be generated, and Acrobat may respond with the message *Insufficient data for an Image*.

If *imagetype* = *raw*, the length of the supplied image data must be equal to  $[width \times components \times bpc / 8] \times height$  bytes, with the bracketed term adjusted upwards to the next integer. The image samples are expected in the standard PostScript/PDF ordering, i.e., top to bottom and left to right (assuming no coordinate transformations have been applied). 16-bit samples must be provided with the most significant byte first (big-endian or »Mac« byte order). The polarity of the pixel values is as discussed in Section 3.3.1, »Color and Color Spaces«, page 63. If *bpc* is smaller than 8, each pixel row begins on a byte boundary, and color values must be packed from left to right within a byte. Image samples are always interleaved, i.e., all color values for the first pixel are supplied first, followed by all color values for the second pixel, and so on.

**Scope** If the *inline* option is not provided, the scope is *document*, *page*, *font*, and this function must always be paired with a matching *PDF\_close\_image()* call. Loading images in *document* or *font* scope instead of *page* scope offers slight output size advantages. If the *inline* option is provided, the scope is *page*, *pattern*, *template*, *glyph*, and *PDF\_close\_image()* must not be called.

**PDF/X** All PDF/X conformance levels:

- ▶ GIF and LZW-compressed TIFF images are not allowed.
- ▶ The *OPI-1.3* and *OPI-2.0* options are not allowed.
- ▶ The *masked* option is only allowed if the mask refers to a 1-bit image.

PDF/X-1 and PDF/X-1a: RGB images are not allowed.

PDF/X-3: Grayscale images require that the *defaultgray* option in *PDF\_begin\_page\_ext()* must have been set unless the PDF/X output intent is a grayscale or CMYK device. RGB images require that the *defaultrgb* option in *PDF\_begin\_page\_ext()* must have been set unless the PDF/X output intent is an RGB device. CMYK images require that the *defaultcmky* option in *PDF\_begin\_page\_ext()* must have been set unless the PDF/X output intent is a CMYK device.

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

Table 8.37 Options for *PDF\_load\_image()*

key	type	explanation
<i>bitreverse</i>	boolean	(Only for <i>imagetype</i> = <i>ccitt</i> ) If true, do a bitwise reversal of all bytes in the compressed data. Default: <i>false</i>
<i>bpc</i>	integer	(Only for <i>imagetype</i> = <i>raw</i> ; required in this case) The number of bits per component; must be 1, 2, 4, or 8. In PDF 1.5 <i>bpc</i> =16 is also allowed.
<i>colorize</i>	spot color handle	Colorize the image with a spot color handle, which must have been retrieved with <i>PDF_makespotcolor()</i> . The image must be a grayscale image.
<i>components</i>	integer	(Only for <i>imagetype</i> = <i>raw</i> ; required in this case) The number of image components (channels); must be 1, 3, or 4.
<i>height</i>	integer	(Only for <i>imagetype</i> = <i>raw</i> and <i>ccitt</i> ; required in this case) Image height in pixels.
<i>honor-iccprofile</i>	boolean	(Only for <i>imagetype</i> = <i>jpeg</i> , <i>png</i> , and <i>tiff</i> ) Read an embedded ICC profile (if any) and apply it to the image. Default: the value of the <i>honoriccprofile</i> parameter.
<i>iccprofile</i>	icc handle	(Only for <i>imagetype</i> = <i>jpeg</i> , <i>png</i> , and <i>tiff</i> ) Handle of an ICC profile which will be applied to the image. Default: the value of the global <i>image:iccprofile</i> parameter.
<i>ignoremask</i>	boolean	Ignore transparency information in the image. Default: <i>false</i>
<i>ignore-orientation</i>	boolean	(Only for <i>imagetype</i> = <i>tiff</i> ) Ignore any orientation tag in an image. This may be useful for compensating wrong orientation information. Default: <i>false</i>
<i>image-warning</i>	boolean	Enable exceptions when the image cannot be opened. Default: the value of the global <i>imagewarning</i> parameter.
<i>inline</i>	boolean	(Only for <i>imagetype</i> = <i>ccitt</i> , <i>jpeg</i> , and <i>raw</i> ) If true, the image will be written directly into the content stream of the page, pattern, template, or glyph description (see Section 5.1.1, »Basic Image Handling«, page 125).
<i>interpolate</i>	boolean	Enable image interpolation to improve the appearance on screen and paper. This is useful for bitmap images for glyph descriptions in Type 3 fonts. Default: <i>false</i>
<i>invert</i>	boolean	Invert the image (swap light and dark colors). This can be used as a workaround for images which are interpreted differently by applications. Default: <i>false</i>
<i>K</i>	integer	(Only for <i>imagetype</i> = <i>ccitt</i> ) CCITT compression parameter for encoding scheme selection. Default: 0 -1 G4 compression 0 one-dimensional G3 compression (G3-1D) 1 mixed one- and two-dimensional compression (G3, 2-D)
<i>mask</i>	boolean	The image is going to be used as a mask (see Section 5.1.3, »Image Masks and Transparency«, page 128). This is required for 1-bit masks, but optional for masks with more than 1 bit per pixel. However, masks with more than 1 bit require PDF 1.4. This option is only allowed for images with one color component (including indexed color). Default: <i>false</i> . There are two uses for masks: Masking another image The returned image handle may be used in subsequent calls for opening another image and can be supplied for the »masked« option. Placing a colorized transparent image Treat the 0-bit pixels in the image as transparent, and colorize the 1-bit pixels with the current fill color.

Table 8.37 Options for PDF\_load\_image()

key	type	explanation
masked	image handle	An image handle for an image which will be applied as a mask to the current image. The image handle has been returned by a previous call to PDF_load_image() (with the »mask« option if it is a 1-bit mask and PDF 1.3 is generated), and has not yet been closed. In PDF 1.3 compatibility mode the image handle must refer to a 1-bit image since soft masks are only supported in PDF 1.4.
OPI-1.3	option list	An option list containing OPI 1.3 PostScript comments as option names; the following entries are required:  ALDImageFilename (string), ALDImageDimensions (list of integers), ALDImageCropRect (rectangle with integers), ALDImagePosition (list of floats)  The following entries are optional: ALDImageID (string), ALDObjectComments (string), ALDImageCropFixed (rectangle), ALDImageResolution (list of floats), ALDImageColorType (keyword; one of Process, Spot, Separation; default: Spot), ALDImageColorType (list of four color values in the range 0...1 and a color name), ALDImageTint (float), ALDImageOverprint (boolean), ALDImageType (list of integers), ALDImageGrayMap (list of integers), ALDImageTransparency (boolean), ALDImageAsciiTag (list of integer/string pairs)
OPI-2.0	option list	An option list containing OPI 2.0 PostScript comments as option names; the following entry is required: ImageFilename (string)  The following entries should either both be present or absent: ImageCropRect (rectangle), ImageDimensions (list of floats)  The following entries are optional: MainImage (string), TIFFASCIITag (list of integer/string pairs), ImageOverprint (boolean), ImageInks (the string full_color, the string registration, or a list containing the string monochrome and string/float pairs for each colorant name and tint), IncludedImageDimensions (list of integers), IncludedImageQuality (integer with one of the values 1, 2, or 3)
page	integer	(Only for imagetype = gif or tiff; must be 1 if used with other formats) Extract the image with the given number from a multi-page image file. The first image has the number 1. Default: 1
rendering-intent	keyword	The rendering intent for the image. See Table 3.7 for a list of possible keywords and their meaning. Default: the value of the global renderingintent parameter.
template	boolean	If true, generate a PDF Image XObject embedded in a Form XObject (template in PDFlib) instead of a plain Image XObject. This is required for compatibility with certain OPI servers when using one of the OPI-1.3 or OPI-2.0 options. Default: false. Scope: document
width	integer	(Only for imagetype = raw and ccitt; required in this case) Image width in pixels.

---

**void PDF\_close\_image(PDF \*p, int image)**

---

Close an image.

**image** A valid image handle retrieved with PDF\_load\_image().

*Details* This function only affects PDFlib's associated internal image structure. If the image has been opened from file, the actual image file is not affected by this call since it has already been closed at the end of the corresponding PDF\_load\_image() call. An image handle cannot be used any more after it has been closed with this function, since it breaks PDFlib's internal association with the image.

*Scope* *document, page, font*; must always be paired with a matching call to *PDF\_load\_image()* unless the *inline* option has been used.

**void PDF\_fit\_image(PDF \*p, int im, double x, double y, const char \*optlist)**

Place an image or template at on the page, subject to various options.

**image** A valid image or template handle retrieved with one of the *PDF\_load\_image\*()* or *PDF\_begin\_template()* functions.

**x, y** The coordinates of the reference point in the user coordinate system where the image or template will be located, subject to various options.

**optlist** An option list specifying placement details according to Table 8.38.

*Details* The image or template (collectively referred to as an object below) will be placed relative to the reference point (x, y). By default, the lower left corner of the object will be placed at the reference point. However, the *orientate*, *boxsize*, *position*, and *fitmethod* options can modify this behavior. By default, an image will be scaled according to its resolution value(s). This behavior can be modified with the *dpi*, *scale*, and *fitmethod* options.

*Scope* *page, pattern, template, glyph* (only if the Type 3 font's *colored* option is true, or if the image is a mask); this function can be called an arbitrary number of times on arbitrary pages, as long as the image handle has not been closed with *PDF\_close\_image()*.

Table 8.38 Options for *PDF\_fit\_image()* and *PDF\_fit\_pdi\_page()*

key	type	explanation
<i>adjustpage</i>	<i>boolean</i>	<i>Adjust the dimensions of the current page to the object such that the upper right corner of the page coincides with the upper right corner of the object plus (x, y). With the value 0 for the position option the following useful cases shall be noted:</i> <i>x &gt;= 0 and y &gt;= 0</i> <i>The object is surrounded by a white margin. This margin has thickness y in horizontal direction and thickness x in vertical direction.</i> <i>x &lt; 0 and y &lt; 0</i> <i>Horizontal and vertical strips will be cropped from the image.</i> <i>This option is only effective in scope page, and must not be used when the topdown parameter has been set to true. Default: false</i>
<i>blind</i>	<i>boolean</i>	<i>If true, all positioning and scaling calculations will be done, but the object will not be placed on the output page. This is useful for processing the blocks on a page without actually using the page's contents. Default: false</i>
<i>boxsize</i>	<i>list of floats</i>	<i>Two values specifying the width and height of a box, relative to which the object will be placed and possibly scaled. The lower left corner of the box coincides with the reference point (x, y). Placing the image and fitting it into the box is controlled by the position and fitmethod options. If width = 0, only the height is considered; If height = 0, only the width is considered. In these cases the object will be placed relative to the vertical line from (x, y) to (x, y+height), or the horizontal line from (x, y) to (x+width, y), respectively. Default: {0 0}</i>

Table 8.38 Options for `PDF_fit_image()` and `PDF_fit_pdi_page()`

key	type	explanation
<code>dpi</code>	list of floats	<p>One or two values specifying the desired image resolution in pixels per inch in horizontal and vertical direction. If a single value is supplied it will be used for both dimensions. With the value 0 the image's internal resolution will be used if available, or 72 dpi otherwise. As an alternative to the value 0, the keyword <code>internal</code> can be supplied. The scaling resulting from this option is relative to the current user coordinate system; if it has been scaled the resulting physical resolution will be different from the supplied values.</p> <p>This option will be ignored for templates and PDF pages, or if the <code>fitmethod</code> option has been supplied with one of the keywords <code>auto</code>, <code>meet</code>, <code>slice</code>, or <code>entire</code>. Default: <code>internal</code></p>
<code>fitmethod</code>	keyword	<p>Specifies the method used to fit the object into the specified box. This option will be ignored if no box has been specified. Default: <code>nofit</code></p> <p><code>nofit</code> Position the object only, without any scaling or clipping.</p> <p><code>clip</code> Position the object, and clip it at the edges of the box.</p> <p><code>meet</code> Position the object according to the <code>position</code> option, and scale it so that it entirely fits into the box while preserving its aspect ratio. Generally at least two edges of the object will meet the corresponding edges of the box. The <code>dpi</code> and <code>scale</code> options are ignored.</p> <p><code>auto</code> Same as <code>meet</code>.</p> <p><code>slice</code> Position the object according to the <code>position</code> option, and scale it such that it entirely covers the box, while preserving the aspect ratio and making sure that at least one dimension of the object is fully contained in the box. Generally parts of the object's other dimension will extend beyond the box, and will therefore be clipped. The <code>dpi</code> and <code>scale</code> options are ignored.</p> <p><code>entire</code> Position the object according to the <code>position</code> option, and scale it such that it entirely covers the box. Generally this method will distort the object. The <code>dpi</code> and <code>scale</code> options are ignored.</p>
<code>orientate</code>	keyword	<p>Specifies the desired orientation of the object when it is placed. Default: <code>north</code></p> <p><code>north</code> upright</p> <p><code>east</code> pointing to the right</p> <p><code>south</code> upside down</p> <p><code>west</code> pointing to the left</p>
<code>position</code>	list of floats	<p>One or two values specifying the position of the reference point (x, y) within the object with {0 0} being the lower left corner of the object, and {100 100} the upper right corner. If the <code>boxsize</code> option has been specified, the <code>position</code> option also specifies the positioning of the box. The values are expressed as percentages of the object's width and height. If both percentages are equal it is sufficient to specify a single float value. Default: 0. Some examples:</p> <p>0 or {0 0} lower left corner</p> <p>{50 100} middle of the top edge</p> <p>50 or {50 50} center of the object</p>
<code>rotate</code>	float	<p>Rotate the coordinate system, using the reference point as center and the specified value as rotation angle in degrees. This results in the box and the object being rotated. The rotation will be reset when the object has been placed. Default: 0.</p>
<code>scale</code>	list of floats	<p>Scale the object in horizontal and vertical direction by the specified scaling factors (not percentages). If both factors are equal it is sufficient to specify a single float value. This option will be ignored if the <code>fitmethod</code> option has been supplied with one of the keywords <code>auto</code>, <code>meet</code>, <code>slice</code>, or <code>entire</code>. Default: 1</p>



# 8.6.2 Templates

*Note* The template functions described in this section are unrelated to variable data processing with PDFlib blocks. Use `PDF_fill_textblock()`, `PDF_fill_imageblock()`, and `PDF_fill_pdfblock()` to fill blocks prepared with the PDFlib block plugin (see Section 8.8, »Block Filling Functions (PPS)«, page 258).

---

**int PDF\_begin\_template(PDF \*p, double width, double height)**

---

Start a template definition.

**width, height** The dimensions of the template’s bounding box in points.

*Returns* A template handle which can be used in subsequent image-related calls, especially `PDF_fit_image()`. There is no error return.

*Details* This function will reset all text, graphics, and color state parameters to their defaults. Hypertext functions and functions for opening images must not be used during a template definition, but all text, graphics, and color functions can be used.

*Scope* *document, page*; this function starts *template* scope, and must always be paired with a matching `PDF_end_template()` call.

---

**void PDF\_end\_template(PDF \*p)**

---

Finish a template definition.

*Scope* *template*; this function terminates *template* scope, and must always be paired with a matching `PDF_begin_template()` call.

# 8.6.3 Thumbnails

---

**void PDF\_add\_thumbnail(PDF \*p, int image)**

---

Add an existing image as thumbnail for the current page.

**image** A valid image handle retrieved with `PDF_load_image()`.

*Details* This function adds the supplied image as thumbnail image for the current page. A thumbnail image must adhere to the following restrictions:

- ▶ The image must be no larger than 106 x 106 pixels.
- ▶ The image must use the grayscale, RGB, or indexed RGB color space.
- ▶ Multi-strip TIFF images can not be used as thumbnails because thumbnails must be constructed from a single PDF image object (see Section 5.1.2, »Supported Image File Formats«, page 126).

This function doesn’t generate thumbnail images for pages, but only offers a hook for adding existing images as thumbnails. The actual thumbnail images must be generated by the client. The client must ensure that color, height/width ratio, and actual contents of a thumbnail match the corresponding page contents.

Since Acrobat 5 and above generates thumbnails on the fly (though not Acrobat 5 or Adobe Reader 6 in the Browser), and thumbnails increase the overall file size of the gen-

erated PDF, it is recommended not to add thumbnails, but rely on client-side thumbnail generation instead.

*Scope* *page*; must only be called once per page. Not all pages need to have thumbnails attached to them.

# 8.7 PDF Import (PDI) Functions

*Note* All functions described in this section require the additional PDF import library (PDI) which requires PDFlib+PDI or PDFlib Personalization Server (PPS), but is not part of PDFlib Lite and PDFlib. Please visit our Web site for more information on obtaining PDI.

## 8.7.1 Document and Page

int PDF\_open\_pdi(PDF \*p, const char \*filename, const char \*optlist, int len)

Open a disk-based or virtual PDF document and prepare it for later use.

**filename** (Name string) The name of the PDF file.

**optlist** An option list specifying PDF open options according to Table 8.39.

**len** (Only for the C binding; must be 0 in other languages) Length of *filename* (in bytes) for UTF-16 strings. If *len* = 0 a null-terminated string must be provided.

**Returns** A document handle which can be used for processing individual pages of the document or for querying document properties. A return value of -1 (in PHP: 0) indicates that the PDF document couldn't be opened. An arbitrary number of PDF documents can be opened simultaneously. The return value can be used until the end of the enclosing document scope.

**Details** By default, the document will be rejected if one or more of the following conditions is found to be true (see Section 5.2.3, »Acceptable PDF Documents«, page 135, for details):

- ▶ The document is damaged.
- ▶ The document uses a higher PDF version than the current PDF document
- ▶ The document is encrypted and the corresponding password has not been supplied in the *password* option.
- ▶ The document does not conform to the current PDF/X output conformance level.
- ▶ The document is Tagged PDF, and the *tagged* option in *PDF\_begin\_document()* is *true*.

Except for the first reason, the *infomode* option can be used to open the document nevertheless. This may be useful to query information about the PDF, such as encryption or PDF/X status, document info fields, etc.

In order to get more detailed information about the nature of a PDF import-related problem (wrong PDF file name, unsupported format, bad PDF data, etc.), use *PDF\_get\_errmsg()* to receive a more detailed error message.

**Scope** *object, document, page*; in *object* scope a PDI document handle can only be used to query information from a PDF document.

**PDF/X** The imported document must be compatible to the current PDF/X output conformance level unless the *infomode* option is *true*.

**Params** See Table 8.42 and Table 8.43.

Table 8.39 Options for PDF\_open\_pdi()

key	type	explanation
infomode	boolean	If true, the document will be opened such that general information can be queried, but the pages can not be imported into the current output document. In particular, the following kinds of documents can be opened when infomode=true (default: false):  PDFs which are not compatible to the current PDF/X mode.  PDFs with a higher PDF version than the current document.  Encrypted PDFs where the password is not known.  Tagged PDF when the tagged option in PDF_begin_document() is true.
password	string	(Maximum string length: 32 characters) The master password required to open a protected PDF document for import. If infomode=true the user password (which may even be empty) is sufficient to query document information. If no password has been supplied at all for an encrypted document the document handle can only be used to query its encryption status.
pdiwarning	boolean	Specifies whether or not this function will throw an exception in case of an error. Default is the value of the pdiwarning parameter (see Table 8.43).

```
int PDF_open_pdi_callback(PDF *p, void *opaque, size_t filesize,
    size_t (*readproc)(void *opaque, void *buffer, size_t size),
    int (*seekproc)(void *opaque, long offset),
    const char *optlist)
```

Open an existing PDF document from a custom data source and prepare it for later use.

**opaque** A pointer to some user data that might be associated with the input PDF document. This pointer will be passed as the first parameter of the callback functions, and can be used in any way. PDI will not use the opaque pointer in any other way.

**filesize** The size of the complete PDF document in bytes.

**readproc** A callback function which copies *size* bytes to the memory pointed to by *buffer*. If the end of the document is reached it may copy less data than requested. The function must return the number of bytes copied.

**seekproc** A callback function which sets the current read position in the document. *offset* denotes the position from the beginning of the document (0 meaning the first byte). If successful, this function must return 0, otherwise -1.

**optlist** An option list specifying PDF open options according to Table 8.39.

**Returns** A document handle which can be used for processing individual pages of the document or for querying document properties. A return value of -1 indicates that the PDF document couldn't be opened. An arbitrary number of PDF documents can be opened simultaneously. The return value can be used until the end of the enclosing document scope.

**Details** This is a specialized interface for applications which retrieve arbitrary chunks of PDF data from some data source instead of providing the PDF document in a disk file or in memory.

**Scope** *object, document, page*; in *object* scope a PDI document handle can only be used to query information from a PDF document.

*Params* See Table 8.42 and Table 8.43.

*Bindings* Only available in the C binding.

---

**void PDF\_close\_pdi(PDF \*p, int doc)**

---

Close all open PDI page handles, and close the input PDF document.

**doc** A valid PDF document handle retrieved with *PDF\_open\_pdi\*( )*.

*Details* This function closes a PDF import document, and releases all resources related to the document. All document pages which may be open are implicitly closed. The document handle must not be used after this call. A PDF document should not be closed if more pages are to be imported. Although you can open and close a PDF import document an arbitrary number of times, doing so may result in unnecessary large PDF output files.

*Scope* *object, document, page*

*Params* See Table 8.42 and Table 8.43.

---

**int PDF\_open\_pdi\_page(PDF \*p, int doc, int pagenumber, const char\* optlist)**

---

Prepare a page for later use with *PDF\_fit\_pdi\_page( )*.

**doc** A valid PDF document handle retrieved with *PDF\_open\_pdi\*( )*.

**pagenumber** The number of the page to be opened. The first page has page number 1.

**optlist** An option list specifying page options according to Table 8.40.

*Returns* A page handle which can be used for placing pages with *PDF\_fit\_pdi\_page( )*. A return value of -1 (in PHP: 0) indicates that the page couldn't be opened. The return value can be used until the end of the enclosing document scope. If the *infomode* option is *true*, or was set to true when the document has been opened with *PDF\_open\_pdi( )* the handle can only be used to retrieve information about the page with *PDF\_get\_pdi\_value( )* and *PDF\_get\_pdi\_parameter( )*, but the handle can not be used with *PDF\_fit\_pdi\_page( )*.

*Details* This function will copy all data comprising the imported page to the output document, but will not have any visible effect on the output. In order to actually place the imported page somewhere in the generated output document, *PDF\_fit\_pdi\_page( )* must be used. In order to get more detailed information about a problem related to PDF import (unsupported format, bad PDF data, etc.), set the *pdiwarning* parameter or option to *true*. If the page has been opened with the *infomode* option set to *true* no data will be copied to the output file.

This function will fail if the PDF version number of the imported document is higher than the PDF version number of the generated PDF output document.

An arbitrary number of pages can be opened simultaneously. If the same page is opened multiply, different handles will be returned, and each handle must be closed exactly once.

*Scope* *document, page*

*Params* See Table 8.42 and Table 8.43.

Table 8.40 Options for `PDF_open_pdi_page()`

key	type	explanation
<code>infomode</code>	<code>boolean</code>	If true, the page will be opened such that general information can be queried, but the pages can not be imported into the current output document. Default: the value of the <code>infomode</code> option supplied to the corresponding call to <code>PDF_open_pdi()</code> (which defaults to false). For documents opened with <code>infomode=true</code> this option will be ignored.
<code>pdiusebox</code>	<code>keyword</code>	Specifies which box dimensions will be used for determining an imported page's size. See Section 5.2.2, »Using PDI Functions with PDFlib«, page 133 for details (default: <code>crop</code> ): <code>media</code> Use the MediaBox (which is always present) <code>crop</code> Use the CropBox if present, else the MediaBox <code>bleed</code> Use the BleedBox if present, else the CropBox <code>trim</code> Use the TrimBox if present, else the CropBox <code>art</code> Use the ArtBox if present, else the CropBox
<code>pdiwarning</code>	<code>boolean</code>	Specifies whether or not this function will throw an exception in case of an error. Default is the value of the <code>pdiwarning</code> parameter (see Table 8.43).

---

**`void PDF_close_pdi_page(PDF *p, int page)`**

---

Close the page handle and free all page-related resources.

**page** A valid PDF page handle (not a page number!) retrieved with `PDF_open_pdi_page()`.

*Details* This function closes the page associated with the page handle identified by *page*, and releases all related resources. *page* must not be used after this call.

*Scope* *document, page*

*Params* See Table 8.42 and Table 8.43.

---

**`void PDF_fit_pdi_page(PDF *p, int page, double x, double y, const char *optlist)`**

---

Place an imported PDF page on the page subject to various options.

**page** A valid PDF page handle (not a page number!) retrieved with `PDF_open_pdi_page()`. The `infomode` option must have been *false* when opening the document and the page. The page handle must not have been closed.

**x, y** The coordinates of the reference point in the user coordinate system where the page will be located, subject to various options.

**optlist** An option list specifying placement details according to Table 8.38.

*Details* This function is similar to `PDF_fit_image()`, but operates on imported PDF pages instead. Most scaling and placement options discussed in Table 8.38 are supported for PDF pages, too.

*Scope* *page, pattern, template, glyph*

*Params* See Table 8.42 and Table 8.43.

*PDF/X* The document from which the page is imported must conform to a PDF/X level which is compatible to the PDF/X level of the generated output (see Table 7.7), and must use the same output intent as the generated document.

### 8.7.2 Other PDI Processing

**int PDF\_process\_pdi(PDF \*p, int doc, int page, const char\* optlist)**

Process certain elements of an imported PDF document.

**doc** A valid PDF document handle retrieved with *PDF\_open\_pdi\*( )*.

**page** If *optlist* requires a page handle (see Table 8.41), *page* must be a valid PDF page handle (not a page number!) retrieved with *PDF\_open\_pdi\_page( )*. The page handle must not have been closed. If *optlist* does not require any page handle, *page* must be -1.

**optlist** An option list specifying processing options according to Table 8.41.

*Returns* The value 1 if the function succeeded, or an error code of -1 (in PHP: o) if the function call failed.

*Scope* document

*Params* See Table 8.43.

*PDF/X* The output intent for the generated document must be set either using this function with the *copyoutputintent* option, or by calling *PDF\_load\_profile( )*.

Table 8.41 Options for *PDF\_process\_pdi( )*

key	type	explanation
action <sup>1</sup>	keyword	(Required, although currently only a single action is defined) Specifies the kind of PDF processing: copyoutputintent Copy the PDF/X output intent of the imported document to the output document. The second and subsequent attempts to copy an output intent will be ignored.
pdiwarning <sup>1</sup>	boolean	Specifies whether or not this function will throw an exception in case of an error. Default is the value of the <i>pdiwarning</i> parameter (see Table 8.43).

1. Does not require a page handle

### 8.7.3 PDI Parameter Handling

**double PDF\_get\_pdi\_value(PDF \*p, const char \*key, int doc, int page, int reserved)**

Get some PDI document parameter with numerical type.

**key** Specifies the name of the parameter to be retrieved, see Table 8.42 and Table 8.43.

**doc** A valid PDF document handle retrieved with *PDF\_open\_pdi( )*.

**page** A valid PDF page handle (not a page number!) retrieved with *PDF\_open\_pdi\_page( )*. For keys which are not page-related *page* must be -1 (in PHP: o).

**reserved** Currently unused, must be o.

**Returns** The numerical value retrieved from the document.

**Scope** any

Table 8.42 Page-related parameters and values for PDF import

function	key	explanation
get_pdi_value	width height	Get the width or height, respectively, of an imported page in default units. Cropping and rotation will be taken into account.
get_pdi_value	/Rotate	page rotation in degrees (0, 90, 180, or 270)
get_pdi_value	/CropBox, /BleedBox, /ArtBox, /TrimBox	Query one of the box parameters of the page. The parameter name must be followed by a slash '/' character and one of llx, lly, urx, ury, for example: /CropBox/llx (see Section 3.2.2, «Page Sizes and Coordinate Limits», page 59 for details). Note that these will not have the /Rotate key applied, unlike the width and height values which already reflect any rotation which may be applied to the page.
get_pdi_parameter	isempty	Returns the string true if the page is empty, false if the page is not empty, and unknown if the page contains an unsupported filter.

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

Get some PDI document parameter with string type.

**key** Specifies the name of the parameter to be retrieved, see Table 8.42 and Table 8.43.

**doc** A valid PDF document handle retrieved with *PDF\_open\_pdi()*.

**page** A valid PDF page handle (not a page number!) retrieved with *PDF\_open\_pdi\_page()*. For keys which are not page-related *page* must be -1 (in PHP: 0).

**reserved** Currently unused, must be 0.

**len** A C-style pointer to an integer which will receive the length of the returned string in bytes. If the pointer is NULL it will be ignored. This parameter is only required for the C binding, and not allowed in other language bindings.

**Returns** The string parameter retrieved from the document as a hypertext string. If no information is available an empty string will be returned.

The contents of the string will be valid until the next call of this function, or the end of the surrounding *object* scope (whatever happens first).

**Details** This function gets some string parameter related to an imported PDF document, in some cases further specified by *page* and *index*. Table 8.43 lists relevant parameter combinations.

**Bindings** C and C++: The *len* parameter must be supplied.

Other bindings: The *len* parameter must be omitted; instead, a string object of appropriate length will be returned.

**Scope** any



Table 8.43 Document-related parameters and values for PDF import

function	key	explanation
get_parameter	pdi <sup>1</sup>	Returns the string true if the PDI library is attached (which is not true for PDFlib Lite), and false otherwise. Scope: any, null <sup>2</sup> .
get_pdi_value	/Root/Pages/Count <sup>1</sup>	total number of pages in the imported document
get_pdi_parameter	filename <sup>1</sup>	name of the imported PDF file; if the file has been opened with PDF_open_pdi_callback() a dummy name will be returned.
get_pdi_parameter	/Info/<key> <sup>1</sup>	Retrieves the string value of a key in the document info dictionary (e.g. /Info/Title) as a hypertext string. Custom keys can also be queried. If the key cannot be found in the document an empty string will be returned. However, if pdiwarning is set to true, an exception will be thrown for a key that couldn't be found.
get_pdi_parameter	tagged	Returns true if the document is Tagged PDF (and therefore cannot be imported in Tagged PDF output mode).
get_pdi_parameter	pdfx <sup>1</sup>	Retrieves the PDF/X conformance level of the imported document. The result is one of »PDF/X-1:2001«, »PDF/X-1a:2001«, »PDF/X-3:2002«, »none«, or a string designating a later PDF/X conformance level (see Section 7.4, »PDF/X«, page 171).
get_pdi_value	version <sup>1</sup>	PDF version number multiplied by 10, e.g. 15 for PDF 1.5
set_parameter	pdiwarning <sup>1</sup>	This parameter can be used to obtain more detailed information about why a PDF or page couldn't be opened. Default: false true      Raise a nonfatal exception when the PDI function fails. The information string supplied with the exception may be useful in debugging import-related problems. false     Do not raise an exception when the PDI function fails. Instead, the function returns -1 (in PHP: 0) on error.
set_parameter	pdiusebox <sup>1</sup>	Deprecated, use the pdiusebox option in PDF_open_pdi_page().
get_pdi_parameter	vdp/Blocks/<block>/<property> or	Query standard and custom block properties (see Section 6.5, »Querying Block Names and Properties«, page 160). Only available in the PDFlib Personalization Server (PPS).
get_pdi_value	vdp/Blocks/<block>/Custom/<property>	
get_pdi_value	vdp/blockcount	Query the total number of blocks on the page.

1. The page parameter must be -1 (in PHP: 0).  
2. May be called with a PDF \* argument of NULL or 0.

## 8.8 Block Filling Functions (PPS)

The PDFlib Personalization Server (PPS) offers dedicated functions for processing variable data blocks of type *Text*, *Image*, and *PDF*. These blocks must be contained in the imported PDF page, but will not be retained in the generated output. The imported page must have been placed on the output page before using any of the block filling functions. When calculating the block position on the page, the block functions will take into account the scaling options provided to the most recent call to *PDF\_fit\_pdi\_page()* with the respective PDF page handle.

If only block processing is desired without actually placing the contents of the page on the output (i.e., the imported page is only used as a container of blocks) the *blind* option of *PDF\_fit\_pdi\_page()* can be used. This is useful if you want to place blocks below the contents of the original page. To achieve this, use *PDF\_fit\_pdi\_page()* with the *blind* option, fill the blocks as desired, and call *PDF\_fit\_pdi\_page()* again, this time without the *blind* option.

*Note* The block processing functions discussed in this section require the PDFlib Personalization Server (PPS). The PDFlib Block plugin for Adobe Acrobat is required for creating blocks in PDF templates. See Chapter 6 for more information about the PDFlib Block plugin.

---

```
int PDF_fill_textblock(PDF *p,  
    int page, const char *blockname, const char *text, int len, const char *optlist)
```

---

Fill a text block with variable data according to its properties.

**page** A valid PDF page handle for a page containing blocks.

**blockname** (Name string) The name of the block.

**text** (Content string) The text to be filled into the block, or an empty string if the default text is to be used.

**len** (C binding only) Length of *text* (in bytes) for UCS-2 strings. If *len* = 0 a null-terminated string must be provided.

**optlist** An option list specifying filling details according to Table 8.44.

**Returns** -1 (in PHP: 0) if the named block doesn't exist on the page, the block cannot be filled (e.g., due to font problems), or the block requires a newer PDFlib version for processing; 1 if the block could be processed successfully. Use the *pdiwarning* option to get more information about the nature of the problem.

**Details** The supplied text will be formatted into the block, subject to the block's properties. If *text* is empty the function will use the block's default text if available, and silently return otherwise. This may be useful to take advantage of other block properties, such as fill or stroke color.

If the PDF document is found to be corrupt, this function will either throw an exception or return -1 subject to the *pdiwarning* parameter or option.

**Scope** *page, template*

*Note* This function is only available in the PDFlib Personalization Server (PPS).

---

```
int PDF_fill_imageblock(PDF *p,  
    int page, const char *blockname, int image, const char *optlist)
```

---

Fill an image block with variable data according to its properties.

**page** A valid PDF page handle for a page containing blocks.

**blockname** (Name string) The name of the block.

**image** A valid image handle for the image to be filled into the block, or -1 if the default image is to be used.

**optlist** An option list specifying filling details according to Table 8.44.

**Returns** -1 (in PHP: o) if the named block doesn't exist on the page, the block cannot be filled, or the block requires a newer PDFlib version for processing; 1 if the block could be processed successfully. Use the *pdiwarning* option to get more information about the nature of the problem.

**Details** The image referred to by the supplied image handle will be placed in the block, subject to the block's properties. If *image* is -1 (in PHP: o) the function will use the block's default image if available, and silently return otherwise.

If the PDF document is found to be corrupt, this function will either throw an exception or return -1 subject to the *pdiwarning* parameter or option.

**Scope** *page, template*

**Note** *This function is only available in the PDFlib Personalization Server (PPS).*

---

```
int PDF_fill_pdfblock(PDF *p,  
    int page, const char *blockname, int contents, const char *optlist)
```

---

Fill a PDF block with variable data according to its properties.

**page** A valid PDF page handle for a page containing blocks.

**blockname** (Name string) The name of the block.

**contents** A valid PDF page handle for the PDF page to be filled into the block, or -1 if the default PDF page is to be used.

**optlist** An option list specifying filling details according to Table 8.44.

**Returns** -1 (in PHP: o) if the named block doesn't exist on the page, the block cannot be filled, or the block requires a newer PDFlib version for processing; 1 if the block could be processed successfully. Use the *pdiwarning* option to get more information about the nature of the problem.

**Details** The PDF page referred to by the supplied page handle *contents* will be placed in the block, subject to the block's properties. If *contents* is -1 (in PHP: o) the function will use the block's default PDF page if available, and silently return otherwise.

If the PDF document is found to be corrupt, this function will either throw an exception or return -1 subject to the *pdiwarning* parameter or option.

**Scope** *page, template*

*Note This function is only available in the PDFlib Personalization Server (PPS).*

*Table 8.44 Options for the PDF\_fill\_\*block() functions*

<b>key</b>	<b>type</b>	<b>explanation</b>
<i>boxsize</i>	<i>list of floats</i>	Change the block's width and height to the specified values (expressed as coordinates in the current user coordinate system). Default: as specified in the block's Rect property.
<i>charref</i>	<i>boolean</i>	(Only for PDF_fill_textblock()) See Table 8.18
<i>encoding</i>	<i>string</i>	Encoding for the font as required by PDF_load_font(). This option is required for PDF_fill_textblock() unless one of the following is true: The string in the text parameter is empty and the defaulttext property is used. The font option has been supplied.
<i>glyphwarning</i>	<i>boolean</i>	(Only for PDF_fill_textblock()) See Table 8.17
<i>font</i>	<i>font handle</i>	(Only for PDF_fill_textblock()) A font handle returned by PDF_load_font(). No default; either font or fontname must be supplied.
<i>fontwarning</i>	<i>boolean</i>	(Only for PDF_fill_textblock()) Specifies whether or not this function will throw an exception in case of font-related problems. Default is the value of the pdiwarning option.
<i>ignore-orientation</i>	<i>boolean</i>	(Only for PDF_fill_imageblock()) If true, the orientation tag in TIFF images will be ignored. Default: false
<i>imagewarning</i>	<i>boolean</i>	(Only for PDF_fill_imageblock()) Specifies whether or not this function will throw an exception in case of image-related problems. Default is the value of the pdiwarning option.
<i>pdiwarning</i>	<i>boolean</i>	Specifies whether or not this function will throw an exception in case of an error in the PDF page containing the block or the page to be used as block contents. Default is the value of the pdiwarning parameter (see Table 8.43).
<i>refpoint</i>	<i>list of floats</i>	Move the lower left corner of the block to the specified point in user coordinates. Default: as specified in the block's Rect property.
<i>shrinklimit</i>	<i>float or percentage</i>	(Only for PDF_fill_textblock()) See Table 8.17
<i>textformat</i>	<i>string</i>	(Only for PDF_fill_textblock() unless the defaulttext property is used) The format used to interpret the supplied text (see Section 4.5.2, »Content Strings, Hypertext Strings, and Name Strings«, page 90 and Table 8.17). Default: auto
<i>all options of PDF_load_font()</i>		(Only for PDF_fill_textblock(), but only if the font option is not supplied) See Table 8.14
<i>almost any property name</i>		Block property names and values (see Section 6.4, »Standard Properties for Automated Processing«, page 153) which will be used to override those in the block definition. See Section 6.2.2, »Block Properties«, page 143, for details. The following block properties can not be overridden:  Name, Description, Locked, Subtype, Type defaulttext, defaultimage, defaultpdf, defaultpdfpage  As an alternative to supplying the fontname property the font option can be used to supply a font handle (fontname will be ignored in this case).  Color properties support the following color space keywords: none, gray, rgb, cmyk, spot, spotname.

# 8.9 Hypertext Functions

Table 8.45 lists relevant parameters and values for this section.

Table 8.45 Parameters for hypertext functions (see Section 8.2.3, »Parameter Handling«, page 198)

function	key	explanation
set_parameter get_parameter	hypertextencoding <sup>1</sup>	Specifies the encoding in which hypertext functions will expect the client-supplied strings (see Section 4.5.4, »String Handling in non-Unicode-capable Languages«, page 91). An empty string is equivalent to unicode. Default: auto. Scope: any
set_parameter get_parameter	hypertextformat <sup>1</sup>	Set the format in which the hypertext functions will expect the client-supplied strings. Possible values are bytes, utf8, utf16, utf16le, utf16be, and auto. Default: auto. Scope: any
set_parameter	usercoordinates	If false, coordinates for hypertext rectangles will be expected in the default coordinate system (see Section 3.2.1, »Coordinate Systems«, page 57); otherwise the current user coordinate system will be used. Default: false. Scope: any

1. This parameter must not be used in the Unicode-capable languages Java and Tcl.

## 8.9.1 Actions

int PDF\_create\_action(PDF \*p, const char \*type, const char \*optlist)

Create an action which can be applied to various objects and events.

**type** The type of the action, specified by one of the following keywords:

- ▶ *GoTo*: go to a destination in the current document.
- ▶ *GoToR*: go to a destination in another (remote) document.
- ▶ *Launch*: launch an application or document.
- ▶ *URI*: resolve a uniform resource identifier, i.e. jump to an Internet address.
- ▶ *Hide*: hide or show an annotation or form field.
- ▶ *Named*: execute an Acrobat menu item identified by its name.
- ▶ *SubmitForm*: send data to a uniform resource locator, i.e., an Internet address.
- ▶ *ResetForm*: set some or all fields in the document to their default values.
- ▶ *ImportData*: import form field values from a file.
- ▶ *JavaScript*: execute a script with JavaScript code.
- ▶ *SetOCGState*: (PDF 1.5) hide or show layers.

**optlist** An option list specifying properties of the action according to Table 8.46.

**Returns** An action handle which can be used to attach actions to objects within the document. The action handle can be used until the end of the enclosing *document* scope.

**Details** This function creates a single action. Various objects (e.g., pages, form field events, bookmarks) can be provided with one or more action, but each action must be generated with a separate call to *PDF\_create\_action()*. Using an action multiply for different objects is allowed.

**Scope** *page, document*. The returned handle can be used until the next call to *PDF\_end\_document()*

**PDF/X** Actions are prohibited in all PDF/X modes.

Table 8.46 Options for action properties with PDF\_create\_action()

option	type	explanation
action-warning	boolean	If true, non-fatal exceptions will be thrown for action options without any effect. If false, they will be silently ignored. Default: true
canonical-date	boolean	(SubmitForm) If true, any submitted field values representing dates are converted to a standard format. The interpretation of a field as a date is not specified explicitly in the field itself, but only in the JavaScript code that processes it. Default: false
defaultdir	string	(Launch) Set the default directory for the launched application. This is only supported by Acrobat on Windows. Default: none
destination	option list	(GoTo, GoToR; required unless destname is supplied) An option list according to Table 8.47 defining the destination to jump to.
destname	hypertext string	(GoTo, GoToR; required unless destination is supplied) The name of a destination which has been defined with PDF_add_nameddest() (for GoTo), or the name of a destination in the remote document (for GoToR).
exclude	boolean	<p>(SubmitForm) If true, the namelist option specifies which fields to exclude; all fields in the document are submitted except those listed in the namelist array and those whose exportable option is false. If false, the namelist option specifies which fields to include in the submission. All members of specified field groups will be submitted as well. Default: false</p> <p>(ResetForm) If true, the namelist option specifies which fields to exclude; all fields in the document are reset except those listed in the namelist array. If false, the namelist option specifies which fields to include in resetting. All members of specified field groups will be reset as well. Default: false</p>
export-method	keyword list	<p>(SubmitForm) The format in which the field names and values are submitted, plus corresponding options (default: fdf):</p> <ul style="list-style-type: none"><li>html in HTML format</li><li>fdf in FDF format</li><li>xfdf in XFDF format</li><li>pdf in PDF format using the MIME content type application/pdf</li></ul> <p>getrequest (only for html and pdf) Submit using HTTP GET; otherwise HTTP POST</p> <p>updates (only for fdf) Include the contents of all incremental updates in the underlying PDF document</p> <p>exclurl (only for fdf) The submitted FDF will exclude the url string.</p> <p>annotfields (only for fdf) Include all annotations and fields.</p> <p>onlyuser (only for fdf and annotfields) The submit will include only those annotations whose name matches the name of the current user, as determined by the remote server to which the field is being submitted.</p> <p>coordinate (only for html) The coordinates of the mouse click that caused the submitform action will be transmitted as part of the form data. The coordinate values are relative to the upper-left corner of the field's rectangle.</p> <p>Example for combined options: exportmethod {fdf updates onlyuser}</p>
filename	string	<p>(GoToR, Launch; required) The name of an external (PDF or other) file or application which will be opened when the action is triggered.</p> <p>(ImportData; required): The name of the external file containing forms data.</p>
hide	boolean	(Hide) Indicates whether to hide (true) or show (false) annotations. Default: true
hypertext-encoding	keyword	Specifies the encoding for the supplied text (see Section 4.5.4, »String Handling in non-Unicode-capable Languages«, page 91). An empty string is equivalent to unicode. Default: the value of the global hypertextencoding parameter
ismap	boolean	(URI) If true, the coordinates of the mouse position will be added to the target URI when the url is resolved. Default: false

Table 8.46 Options for action properties with PDF\_create\_action()

option	type	explanation
layerstate	option list	(SetOCGState; required) A list of pairs where each pair consists of a keyword and a layer handle. The following keywords are supported: on           Activate the layer off          Deactivate the layer toggle      Reverse the state of the layer. If this is used the preserve-radio option should be set to false.
menuname	string	(Named; required) The name of the menu item to be performed. Well-known names are nextpage, prevpage, firstpage, lastpage, but others will be accepted. To find the names of other menu items you can execute the following code in Acrobat's JavaScript console or debugger: <pre>function MenuList(m, level) {     console.println(m.cName);     if (m.oChildren != null)         for (var i = 0; i &lt; m.oChildren.length; i++)             MenuList(m.oChildren[i], level + 1); } var m = app.listMenuItems(); for (var i=0; i &lt; m.length; i++)     MenuList(m[i], 0);</pre>
namelist	list of strings	(Hide; required) The names (including group names) of the annotations or fields to be hidden or shown.  (SubmitForm) The names (including group names) of form fields to include in the submission or which to exclude, depending on the setting of the exclude option. Default: all fields are submitted except those whose exportable option is false.  (ResetForm) The names (including group names) of form fields to include in the resetting or which to exclude, depending on the setting of the exclude option. Default: all fields are reset.
newwindow	boolean	(GoToR, Launch) A flag specifying whether to open the destination document in a new window. If this flag is false, the destination document will replace the current document in the same window. Launch: This entry is ignored if the file is not a PDF document. Default: Acrobat behaves according to the current user preference.
operation	keyword	(Launch) A keyword specifying the operation to be applied to the document specified in the filename option. This is only supported by Acrobat on Windows. If the filename option designates an application instead of a document, this option will be ignored and the application is launched (default: open): open       open a document print      print a document
parameters	string	(Launch) A parameter string to be passed to the application specified with the filename option. This is only supported by Acrobat on Windows. Multiple parameters can be separated with a space character, but individual parameters must not contain any space characters. This option should be omitted if filename designates a document. Default: none
preserve-radio	boolean	(SetOCGState) If true, preserve the radio-button state relationship between layers. Default: true
script	hypertext string	(JavaScript; required) A string containing the JavaScript code to be executed.
scriptname	hypertext string	(JavaScript) If present, the JavaScript supplied in the script option will be inserted as a document-level JavaScript with the supplied name. If the same scriptname is supplied more than once in a document only the last script will be used, the others will be silently ignored.

Table 8.46 Options for action properties with `PDF_create_action()`

option	type	explanation
submit-emptyfields	boolean	(SubmitForm; PDF 1.4) If true, all fields characterized by the <code>namelist</code> and <code>exclude</code> options are submitted, regardless of whether they have a value. For fields without a value, only the field name is transmitted. If false, fields without a value are not submitted. Default: false
url	string	(URI; required) A Uniform Resource Locator encoded in 7-bit ASCII specifying the link target. It can point to an arbitrary (Web or local) resource. The <code>textx/texty</code> , <code>currentx/currenty</code> , and <code>imagewidth/imageheight</code> parameters may be useful for retrieving positioning information for calculating the dimension of link rectangles.  (SubmitForm; required) A URL specification giving the uniform resource locator (address) of the script at the Web server that will process the submission.

## 8.9.2 Named Destinations

---

```
void PDF_add_nameddest(PDF *p, const char *name, int len, const char *optlist)
```

---

Create a named destination on an arbitrary page in the current document.

**name** (Hypertext string) The name of the destination, which can be used as a target for links, bookmarks, or other triggers. Destination names must be unique within a document. If the same name is supplied more than once for a document only the last definition will be used, the others will be silently ignored.

**len** (C binding only) Length of *name* (in bytes) for UCS-2 strings. If *len* = 0 a null-terminated string must be provided.

**optlist** An option list specifying the destination according to Table 8.47. An empty list is identical to `{type fitwindow page 0}`.

*Details* The destination details must be specified in *optlist*, and the destination may be located on any page in the current document. The provided *name* can be used as a target for all functions and parameters which accept destination options according to Table 8.47.

*Scope* document, page

Table 8.47 Destination options for `PDF_add_nameddest()`, as well as for the destination option in `PDF_create_action()`, `PDF_create_annotation()`, `PDF_create_bookmark()`, and `PDF_begin/end_document()`.

option	type	explanation
group	string	(Required if the <code>page</code> option has been specified and the document uses page groups; not allowed otherwise.) Name of the page group that the destination page belongs to.
hypertext-encoding	keyword	Specifies the encoding for the <code>name</code> parameter (see Section 4.5.4, »String Handling in non-Unicode-capable Languages«, page 91). An empty string is equivalent to <code>unicode</code> . Default: the value of the <code>global hypertextencoding</code> parameter
hypertext-format	keyword	Set the format for the <code>name</code> parameter. Possible values are <code>bytes</code> , <code>utf8</code> , <code>utf16</code> , <code>utf16le</code> , <code>utf16be</code> , and <code>auto</code> . Default: the value of the <code>hypertextformat</code> parameter



Table 8.47 Destination options for PDF\_add\_nameddest(), as well as for the destination option in PDF\_create\_action(), PDF\_create\_annotation(), PDF\_create\_bookmark(), and PDF\_begin/end\_document().

option	type	explanation
type	keyword	Specifies the location of the window on the target page (default: fitwindow): fixed Use a fixed destination view specified by the left, top, and zoom options. If any of these is missing its current value will be retained. fitwindow Fit the complete page to the window. fitwidth Fit the page width to the window, with the y coordinate top at the top edge of the window. fitheight Fit the page height to the window, with the x coordinate left at the left edge of the window. fitrect Fit the rectangle specified by left, bottom, right, and top to the window. fitvisible Fit the visible contents of the page (the ArtBox) to the window. fitvisiblewidth Fit the visible contents of the page to the window with the y coordinate top at the top edge of the window fitvisibleheight Fit the visible contents of the page to the window with the x coordinate left at the left edge of the window. nameddest (Not for PDF_add_nameddest()) A named destination specified with the name option.
name	hypertext string	(Not for PDF_add_nameddest(); required if type = nameddest, and ignored otherwise). String designating a named destination which must be defined in the target file. If this option is provided no other option except type must be used. Destination names must be unique within a document.
page	integer	The page number of the destination page (first page is 1). The page must exist in the destination PDF. Page 0 means the current page if in scope page, and page 1 if in scope document. Note that due to a bug Acrobat 6.0 will ignore the page number, and will always jump to page 1. This bug has been fixed in Acrobat 6.0.1, and is not present in older versions. Default: 0
zoom	float or percentage	(Only for type = fixed) The zoom factor (1 means 100%) to be used to display the page contents. If this option is missing or 0 the zoom factor which was in effect when the link was activated will be retained.
left	float	(Only for type = fixed, fitheight, fitrect, or fitvisibleheight) The x coordinate of the page which will positioned at the left edge of the window. Default: 0
right	float	(Only for type = fitrect) The x coordinate of the page which will positioned at the right edge of the window. Default: 1000
bottom	float	(Only for type = fitrect) The y coordinate of the page which will positioned at the bottom edge of the window. Default: 0
top	float	(Only for type = fixed, fitwidth, fitrect, or fitvisiblewidth) The y coordinate of the page which will positioned at the top edge of the window. Default: 1000

8.9.3 Annotations

```
void PDF_create_annotation(PDF *p,  
                           double llx, double lly, double urx, double ury, const char *type, const char *optlist)
```

Create a rectangular annotation on the current page.

**llx, lly, urx, ury** x and y coordinates of the lower left and upper right corners of the annotation rectangle in default coordinates (if the *usercoordinates* parameter or option is

*false*) or user coordinates (if it is *true*). Acrobat will align the upper left corner of the annotation at the upper left corner of the specified rectangle.

Note that annotation coordinates are different from the parameters of the `PDF_rect()` function. While `PDF_create_annotation()` expects parameters for two corners directly, `PDF_rect()` expects the coordinates of one corner, plus width and height values.

**type** The type of the annotation, specified by one of the following keywords:

- ▶ *Circle*: circle annotation
- ▶ *FileAttachment*: file attachment annotation. Acrobat Reader 5 is unable to deal with file attachments and will display a question mark instead. File attachments only work in the full Acrobat software.
- ▶ *FreeText*: free text annotation
- ▶ *Highlight*: highlight annotation
- ▶ *Ink*: ink annotation
- ▶ *Line*: line annotation
- ▶ *Link*: link annotation
- ▶ *Polygon*: (PDF 1.5) Polygon annotation (vertices connected by straight lines)
- ▶ *PolyLine*: (PDF 1.5) Polyline annotation; similar to polygons, except that the first and last vertices are not connected.
- ▶ *Popup*: Pop-up annotation
- ▶ *Square*: square annotation
- ▶ *Squiggly*: (PDF 1.4) squiggly-underline annotation
- ▶ *Stamp*: rubber stamp annotation
- ▶ *StrikeOut*: strikeout annotation
- ▶ *Text*: text annotation. In Acrobat this type is called *note* annotation.
- ▶ *Underline*: underline annotation

**optlist** An option list specifying annotation properties according to Table 8.48.

*Scope* page

**PDF/X** In all PDF/X modes annotations are only allowed if they are positioned completely outside of the BleedBox (or TrimBox/ArtBox if no BleedBox is present).

Table 8.48 Options for annotations with `PDF_create_annotation()`

option	type	explanation
action	action list	List of annotation actions for the following event (default: empty list): activate Actions to be performed when the annotation is activated. All types of actions are permitted.
alignment	keyword	(Only for type=FreeText) Alignment of text in the annotation: left, center, right. This option does not work in Acrobat 6, which always uses left. Default: left
annotcolor	color	The color of the background of the annotation's icon when closed, the title bar of the annotation's pop-up window, and the border of a link annotation. Supported color spaces: none, gray, rgb. Default: none
annot-warning	boolean	If true, non-fatal exceptions will be thrown for annotation options without any effect. If false, they will be silently ignored. Default: true
borderstyle	keyword	Style of the annotation border or the line of the annotation types Polygon, Poly-Line, Line, Square, Circle, Ink: solid, beveled, dashed, inset, underline. Note that the beveled, inset, and underline styles do not work reliably in Acrobat. Default: solid
cloudy	float	(Only for type=Polygon) Specifies the intensity of the »cloud« effect used to render the polygon. Possible values are 0 (no effect), 1, and 2. If this option is used the borderstyle option will be ignored. Default: 0

Table 8.48 Options for annotations with PDF\_create\_annotation()

option	type	explanation
contents	hypertext string	(Required for type=Text, FreeText, Line, Square, Circle, Highlight, Underline, PolyLine, Polygon, Squiggly, Strikeout, Stamp, Ink, FileAttachment; optional for type=Link, PopUp; if type=FreeText it must be of type string) Text to be displayed for the annotation or (if the annotation does not display text) an alternate description of its contents in human-readable form. The maximum length of contents is 65535 bytes. Carriage return or line feed characters can be used to force a new paragraph.
custom	list of option lists	(Only for advanced users) This option can be used to insert an arbitrary number of private entries in the annotation dictionary, which may be useful for specialized applications such as inserting processing instructions for digital printing machines. Using this option requires knowledge of the PDF file format and the target application. Corrupt PDF output may be generated if unsuitable values are supplied. Each list must contain three options: <div><div>key</div><div>(string) The name of the dictionary key (excluding the / character). Any non-standard PDF key can be specified, as well as the following standard keys: Contents, Name (option iconname), NM (option name), and Open. The corresponding options will be ignored in this case.</div></div> <div><div>type</div><div>(keyword) The type of the corresponding value, which must be one of boolean, name, or string</div></div> <div><div>value</div><div>(Hypertext string if type=string, otherwise string) The value as it will appear in the PDF output; PDFlib will automatically apply any decoration required for strings and names.</div></div>
dasharray	list of floats	(Only for borderstyle=dashed). The lengths of dashes and gaps for a dashed border in default units (see PDF_setdash()). Default: 3 3
destination	option list	(Only for type=Link) Defines the destination to jump to. Destination or destname actions will be dominant over this option.
destname	hypertext string	(Only for type=Link) The name of a destination which has been defined with PDF_add_nameddest(). Destination or destname actions will be dominant over this option.
display	keyword	Visibility on screen and paper: visible, hidden, noview, noprint. Default: visible
endingstyles	keyword list	(Only for type=Line, PolyLine) A list with two keywords specifying the line ending styles: none, square, circle, diamond, openarrow, closedarrow. Default: none none
filename	string	(Only for type=FileAttachment; required) The file associated with the annotation. It is recommended to use only ASCII characters in the filename.
fillcolor	color	(Only for type=FreeText) Fill color of the text. Supported color spaces: gray, rgb, cmyk. Default: gray o (=black)
font	font handle	(Only for type=FreeText; required) Specifies the font to be used for the annotation. Only PDF core fonts and the following encodings are allowed: any 8-bit encoding, UCS-2 CMaps, builtin.
fontsize	float	(Only for type=FreeText; required) The font size in default or user coordinates depending on the usercoordinates option or parameter).
highlight	keyword	(Only for type=Link) Highlight mode of the annotation when the user clicks on it: none, invert, outline, push. Default: invert
hypertext-encoding	keyword	Specifies the encoding for the supplied text (see Section 4.5.4, »String Handling in non-Unicode-capable Languages«, page 91). An empty string is equivalent to unicode. Default: the value of the global hypertextencoding parameter

Table 8.48 Options for annotations with PDF\_create\_annotation()












option	type	explanation
iconname	string	<p>(Only for type=Text, Stamp, FileAttachment) The name of an icon to be used in displaying the annotation:</p> <p>For type=Text (default: note): comment , key , note , help ,</p> <p>newparagraph , paragraph , insert </p> <p>For type=Stamp (default: draft): approved, experimental, notapproved, asis, expired, notforpublicrelease, confidential, final sold, departmental, forcomment, topsecret, draft, forpublicrelease.</p> <p>For type=FileAttachment (default: pushpin):</p> <p>graph , pushpin , paperclip , tag </p>
interiorcolor	color	(Only for type=Line, PolyLine, Square, Circle) The color for the annotation's line endings, rectangle, or ellipse, respectively. Supported color spaces: none, gray, rgb. Default: none
line	list of 4 floats	(Only for type=Line; required) A list of four numbers x1, y1, x2, y2 specifying the start and end coordinates of the line in default coordinates (if the usercoordinates parameter is false) or user coordinates (if it is true).
linewidth	integer	Width of the annotation border or the line of the annotation types Line, PolyLine, Polygon, Square, Circle, Ink in default units (=points). If linewidth = 0 the border will be invisible. Default: 1
locked	boolean	If true, the annotation properties cannot be edited in Acrobat. Default: false
mimetype	string	(Only for type=FileAttachment) The MIME type of the file. Acrobat will use it for launching the appropriate application when the annotation is activated.
name	string	A name uniquely identifying the annotation. The name is necessary for some actions, and must be unique on the page. Default: none
open	boolean	(Only for type=Text, Popup) If true, the annotation will initially be displayed open. Default: false
parentname	string	(Only for type=PopUp) The name of the parent annotation for the annotation.
polylinelist	list of four or more floats	<p>(Only for type=Polygon, PolyLine, Ink, Highlight, Underline, Squiggly, Strikeout; required). A polyline is a list of float values specifying coordinate pairs. The coordinates will be interpreted in default coordinates (if the usercoordinates option is false) or user coordinates (if it is true).</p> <p>type=Polygon, PolyLine, Ink</p> <p>The list contains 2 x n float values specifying the coordinates of n points (minimum: 2). The points will be connected by straight lines.</p> <p>others</p> <p>The list contains 8 x n float values specifying n quadrilaterals (minimum: 1). Each quadrilateral encompasses a word or group of contiguous words in the text underlying the annotation. The coordinates for each quadrilateral are given as x1 y1 x2 y2 x3 y3 x4 y4 specifying the quadrilateral's vertices in counterclockwise order. The text is oriented with respect to the edge connecting (x1, y1) and (x2, y2).</p>
popup	string	Name of a PopUp annotation for entering or editing the text associated with this annotation. Default: none
readonly	boolean	If true, do not allow the annotation to interact with the user. The annotation may be displayed or printed, but should not respond to mouse clicks or change its appearance in response to mouse motions. Default: false
rotate	boolean	If true, rotate the annotation to match the rotation of the page. Otherwise the annotation's rotation will remain fixed. This option will be ignored for the icons of text annotations. Default: true

Table 8.48 Options for annotations with PDF\_create\_annotation()

option	type	explanation
title	hypertext string	The text label to be displayed in the title bar of the annotation's pop-up window when open and active. The maximum length of title is 255 single-byte characters or 126 Unicode characters. However, a practical limit of 32 characters for title is advised. Default: none
user-coordinates	boolean	If false, annotation coordinates and font size will be expected in the default coordinate system (see Section 3.2.1, «Coordinate Systems», page 57); otherwise the current user coordinate system will be used. Default: the value of the global usercoordinates parameter
zoom	boolean	If true, scale the annotation to match the magnification of the page. Otherwise the annotation's size will remain fixed. This option will be ignored for the icons of text annotations. Default: true

8.9.4 Form Fields

```
void PDF_create_field(PDF *p, double llx, double lly, double urx, double ury,
    const char *name, int len, const char *type, const char *optlist)
```

Create a form field on the current page subject to various options.








**llx, lly, urx, ury** x and y coordinates of the lower left and upper right corners of the field rectangle in default coordinates (if the *usercoordinates* parameter or option is *false*) or user coordinates (if it is *true*).

Note that form field coordinates are different from the parameters of the *PDF\_rect()* function. While *PDF\_create\_field()* expects parameters for two corners directly, *PDF\_rect()* expects the coordinates of one corner, plus width and height values.

**name** (Hypertext string) The form field name, possibly prefixed with the name(s) of one or more groups which have been created with *PDF\_create\_fieldgroup()*. Group names must be separated from each other and from the field name by a period «.» character. Field names must be unique on a page, and must not end in a period «.» character.

**len** (C binding only) Length of *text* (in bytes) for UCS-2 strings. If *len* = 0 a null-terminated string must be provided.

**type** The field type, which must be one of the following:

- ▶  *pushbutton*
- ▶  *checkbox*
- ▶  *radiobutton*
- ▶  *listbox*
- ▶  *combobox*
- ▶  *textfield*
- ▶  *signature*

If *type=radiobutton* the *name* must be prefixed with a group name since radio buttons must always belong to a group. For all other field types group membership is optional.

**optlist** An option list specifying the field's properties according to Table 8.49. String options will be interpreted as hypertext strings or text strings as noted in the table.

*Details* The tab order of the fields on the page (the order in which they receive the focus when the tab key is pressed) is determined by the order of calls to *PDF\_create\_field()*. It can not be modified afterwards.

*Scope* page

*PDF/X* In all PDF/X modes form fields are only allowed if they are positioned completely outside of the BleedBox (or TrimBox/ArtBox if no BleedBox is present).

Table 8.49 Options for field properties with *PDF\_create\_field()* and *PDF\_create\_fieldgroup()*

option	type	explanation
action	action list	List of field actions for one or more of the following events. The activate event is allowed for all field types, the other events are not allowed for type=pushbutton, checkbox, and radiobutton (default: empty list):
		activate Actions to be performed when the field is activated.
		keystroke JavaScript actions to be performed when the user types into a text field or combo box, or modifies the selection in a scrollable list box.
		format JavaScript actions to be performed before the field is formatted to display its current value. This allows the field's value to be modified before formatting.
		validate JavaScript actions to be performed when the field's value is changed. This allows the new value to be checked for validity.
		calculate JavaScript actions to be performed in order to recalculate the value of this field when the value of another field changes.
		enter Actions to be performed when the mouse enters the field's area.
		exit Actions to be performed when the mouse exits the field's area.
		down Actions to be performed when the mouse button is pressed inside the field's area.
		up Actions to be performed when the mouse button is released inside the field's area (this is typically used to activate a field).
		focus Actions to be performed when the field receives the input focus.
		blur Actions to be performed when the field loses the input focus.
alignment	keyword	Alignment of text in the field: left, center, right. Default: left
background-color bordercolor	color	Color of the field background or border. Supported color spaces: none, gray, rgb, cmyk. Default: none
borderstyle	keyword	Style of the field border, which is one of solid, beveled, dashed, inset, underline. Default: solid
button-layout	keyword	(Only for type=pushbutton) The position of the button caption relative to the button icon, provided both have been specified: below, above, right, left, overlaid. Default: right
buttonstyle	keyword	(Only for type=radiobutton and checkbox) Specifies the symbol to be used for the field: check, cross, diamond, circle, star, square. Default: check
calcorder	integer	(Only used if the field has a JavaScript action for the calculate event) Specifies the calculation order of the field relative to other fields. Fields with smaller numbers will be calculated before fields with higher numbers. Default: 10 plus the maximum calcorder used on the current page (and 10 initially)

Table 8.49 Options for field properties with PDF\_create\_field() and PDF\_create\_fieldgroup()

option	type	explanation
caption	content string	(Only for type=pushbutton; one of the caption or icon options must be supplied for push buttons) The caption text which will be visible when the button doesn't have input focus. Default: none
captiondown	content string	(Only for type=pushbutton) The caption text which will be visible when the button is activated. Default: none
caption-rollover	content string	(Only for type=pushbutton) The caption text which will be visible when the button has input focus. Default: none
charspacing	float	(Not for type=radiobutton and checkbox) The character spacing for text in the field in units of the current user coordinate system. Default: 0
comb	boolean	(Only for type=textfield; PDF 1.5) If true and the multiline, fileselect, and password options are false, and the maxchar option has been supplied with an integer value, the field will be divided into a number of equidistant subfields (according to the maxchar value) for individual characters. Default: false
commit-onselect	boolean	(Only for type=listbox and combobox; PDF 1.5) If true, an item selected in the list will be committed immediately upon selection. If false, the item will only be committed upon exiting the field. Default: false
currentvalue	(various)	(Not for type=pushbutton and signature) The field's initial value. Type and default depend on the field type: checkbox, radiobutton (string) Arbitrary string other than Off means that the button is activated; Acrobat 6 shows erratic behavior if itemname is specified and/or uniselect is true. The string Off means that the button is deactivated. This option should be set for the first button. Default: Off textfield, combobox (content string) Contents of the field. Default: empty listbox (list of integers) Indices of the selected items within itemtextlist. Default: none
dasharray	list of floats	(Only for borderstyle=dashed). The lengths of dashes and gaps for a dashed border in default units (see PDF_setdash()). Default: 3 3
defaultvalue	(various)	(For type=textfield or combobox this option has type content string) The field's value after a reset action. Types and defaults are the same as for the currentvalue option. Exception: for listboxes only a single integer value is allowed.
display	keyword	Visibility on screen and paper: visible, hidden, noview, noprint. Default: visible
editable	boolean	(Only for type=combobox) If true, the currently selected text in the box can be edited. Default: false
exportable	boolean	The field will be exported when a SubmitForm action happens. Default: true
fieldwarning	boolean	If true, non-fatal exceptions will be thrown for field options without any effect. If false, they will be silently ignored. Default: true
fileselect	boolean	(Only for type=textfield) If true, the text in the field will be treated as a file name. Default: false
fillcolor	color	Fill color for text. Supported color spaces: gray, rgb, cmyk. Default: gray 0 (=black)
fitmethod	keyword	(Only for type=pushbutton) The method of placing a template provided with the icon, icondown, and iconrollover options within the button (default: meet): auto same as meet if the template fits into the button, otherwise clip nofit same as clip clip template will not be scaled, but clipped at the field border meet template will be scaled proportionally so that it fits into the button slice same as meet entire template will be scaled so that it fully fits into the button

Table 8.49 Options for field properties with `PDF_create_field()` and `PDF_create_fieldgroup()`

option	type	explanation
font	font handle	(Required except for <code>type=radiobutton</code> and <code>checkbox</code> which always use ZapfDingbats). Specifies the font to be used for the field. The following options must have been set in the corresponding call to <code>PDF_load_font()</code> : <code>embedding</code> (with the exception of core fonts which need not be embedded), <code>nosubsetting</code> , <code>noautocidfont</code> . The following encodings are allowed: 8-bit encodings, Unicode CMaps, builtin
fontsize	float or keyword	The font size in user coordinates. If the keyword <code>auto</code> is supplied instead of a float value Acrobat will determine the font size automatically. Default: <code>auto</code>
highlight	keyword	Highlight mode of the field when the user clicks on it: <code>none</code> , <code>invert</code> , <code>outline</code> , <code>push</code> . Default: <code>invert</code>
hypertext-encoding	keyword	Specifies the encoding for the name parameter (see Section 4.5.4, »String Handling in non-Unicode-capable Languages«, page 91). An empty string is equivalent to <code>unicode</code> . Default: the value of the global <code>hypertextencoding</code> parameter
hypertext-format	keyword	Set the format for the name parameter. Possible values are <code>bytes</code> , <code>utf8</code> , <code>utf16</code> , <code>utf16le</code> , <code>utf16be</code> , and <code>auto</code> . Default: the value of the <code>hypertextformat</code> parameter
icon	template handle	(Only for <code>type=pushbutton</code> ; one of the <code>caption</code> or <code>icon</code> options must be supplied for push buttons) The handle for a template which will be visible when the button doesn't have input focus. Default: <code>none</code>
icondown	template handle	(Only for <code>type=pushbutton</code> ) The handle for a template which will be visible when the button is activated. Default: <code>none</code>
iconrollover	template handle	(Only for <code>type=pushbutton</code> ) The handle for a template which will be visible when the button has input focus. Default: <code>none</code>
itemname	hypertext string	(Only for <code>type=radiobutton</code> and <code>checkbox</code> ; must be used if the export value is not a Latin 1 string) Export value of the field. Item names for multiple radio buttons in a group may be identical. Acrobat 6: Checkboxes within a group which have the same item name will be switched on or off simultaneously, even if they are located on separate pages. Default: field name
item-namelist	hypertext string	(Only for <code>type=listbox</code> and <code>combobox</code> ) Export values of the list items. Multiple items may have the same export value. Default: <code>none</code>
itemtextlist	list of content strings	(only for <code>type=listbox</code> and <code>combobox</code> , and required in these cases) Text contents for all items in the list. If both <code>itemnamelist</code> and <code>itemtextlist</code> are specified both must contain the same number of strings.
linewidth	integer	Line width of the field border in default units (=points). Default: 1
locked	boolean	If true, the field properties cannot be edited in Acrobat. Default: <code>false</code>
maxchar	integer or keyword	(Only for <code>type=textfield</code> ) The upper limit for the number of text characters in the field, or the keyword <code>unlimited</code> if there is no limit. Default: <code>unlimited</code>
multiline	boolean	(Only for <code>type=textfield</code> ) If true, text will be wrapped to multiple lines if required. Default: <code>false</code>
multiselect	boolean	(Only for <code>type=listbox</code> ) If true, multiple items in the list can be selected. Default: <code>false</code>
orientate	keyword	Orientation of the contents within the field rectangle: <code>north</code> , <code>west</code> , <code>south</code> , <code>east</code> . Default: <code>north</code>
password	boolean	(Only for <code>type=textfield</code> ) If true, the text will be simulated with bullets or asterisks upon input. Default: <code>false</code>
position	list of floats	(Only for <code>type=pushbutton</code> ) Relative position of a template provided with the <code>icon...</code> options within the button, specified as a percentage. Default: 50 50
readonly	boolean	If true, the field does not allow any input. Default: <code>false</code>
required	boolean	If true, the field must contain a value when the form is submitted. Default: <code>false</code>



Table 8.49 Options for field properties with PDF\_create\_field() and PDF\_create\_fieldgroup()

option	type	explanation
scrollable	boolean	(Only for type=textfield) If true, text will be moved to the invisible area outside the field if the text doesn't fit into the field. If false, no more input will be accepted when the text fills the full field. Default: true
sorted	boolean	(Only for type=listbox and combobox) If true, the contents of the list will be sorted. Default: false
spellcheck	boolean	(Only for type=textfield and combobox) If true, the spell checker will be active in the field. Default: true
strokecolor	color	Stroke color for text. Supported color spaces: gray, rgb, cmyk. Default: gray 0 (=black).
submitname	hypertext string	(Recommended only for type=pushbutton) URL-encoded string of the Internet address to which the form will be submitted. Default: None
taborder	integer	Specifies the tab order of the field relative to other fields. Fields with smaller numbers will be reached before fields with higher numbers. Default: 10 plus the maximum taborder used on the current page (and 10 initially); the result of this default is that the creation order will specify the tab order.
toggle	boolean	(Only for type=radiobutton) If true, a radio button within a group can be activated and deactivated by clicking. If false, it can only be activated by clicking, and deactivating by clicking another button. Default: false
tooltip	hypertext string	The text visible in the field's tooltip. For radio buttons Acrobat will always use the tooltip of the first button in the group, others will be ignored. Default: none
topindex	integer	(Only for type=listbox) Index of the first visible entry. The first item has index 0. Default: 0
unisonselect	boolean	(Only for type=radiobutton; PDF 1.5) If true, radio buttons with the same field name or item name will be selected simultaneously. Default: false
user-coordinates	boolean	If false, field coordinates will be expected in the default coordinate system (see Section 3.2.1, »Coordinate Systems«, page 57); otherwise the current user coordinate system will be used. Default: the value of the global usercoordinates parameter

**void PDF\_create\_fieldgroup(PDF \*p, const char \*name, int len, const char \*optlist)**

Create a form field group subject to various options.

**name** (Hypertext string) The name of the form field group, which may in turn be prefixed with the name of another group. Field groups can be nested to an arbitrary level. Group names must be separated with a period ».« character. Group names must be unique within the document, and must not end in a period ».« character.

**len** (C binding only) Length of *text* (in bytes) for UCS-2 strings. If *len* = 0 a null-terminated string must be provided.

**optlist** An option list specifying field properties according to Table 8.49 and Table 8.50.

*Details* If the name of a field group is provided as prefix for a field name created with *PDF\_create\_field()*, the new field will be part of this group. All field property options provided in the *optlist* for a group will be inherited by all fields belonging to this group.

*Scope* page, document

Table 8.50 Options for field properties with `PDF_create_fieldgroup()`

option	type	explanation
<i>fieldtype</i>	keyword	Type of the fields contained in the group: <i>mixed</i> , <i>pushbutton</i> , <i>checkbox</i> , <i>radio-button</i> , <i>listbox</i> , <i>combobox</i> , <i>textfield</i> , or <i>signature</i> . Unless <i>fieldtype=mixed</i> the group may only contain fields of the specified type. If a particular <i>fieldtype</i> has been specified for the group, the current value is displayed in all contained fields simultaneously, even if the fields are located on separate pages. If <i>fieldtype=radio-button</i> the option <i>unisonselect</i> must be supplied. The options <i>itemtextlist</i> , <i>itemnamelist</i> , <i>currentvalue</i> and <i>defaultvalue</i> must be specified in the field group options, and not in the individual fields' options. Default: <i>mixed</i>

## 8.9.5 Bookmarks

---

**int** `PDF_create_bookmark(PDF *p, const char *text, int len, const char *optlist)`

---

Create a bookmark subject to various options.

**text** (Hypertext string) Contains the text of the bookmark. The maximum length of *text* is 255 single-byte characters (8-bit encodings), or 126 Unicode characters. However, a practical limit of 32 characters for *text* is recommended.

**len** (Only for the C binding.) Length of *text* (in bytes) for UCS-2 strings. If *len = 0* a null-terminated string must be provided.

**optlist** An option list specifying the bookmark's properties according to Table 8.51.

**Returns** A handle for the generated bookmark, which may be used with the *parent* option in subsequent calls.

**Details** This function adds a PDF bookmark with the supplied *text*. Unless the *destination* option has been specified the bookmark will point to the current page (or the last page if used in *document* scope, or the first page if used before the first page).

Creating bookmarks sets the *openmode* option of `PDF_begin_document()` and `PDF_end_document()` to *bookmarks* unless another mode has explicitly been set.

**Scope** *document, page*

Table 8.51 Options for `PDF_create_bookmark()`

option	type	explanation
<i>action</i>	action list	List of bookmark actions for the following event (default: <i>GoTo</i> action with the target specified in the destination option): <i>activate</i> Actions to be performed when the bookmark is activated. All types of actions are permitted.
<i>destination</i>	option list	An option list specifying the bookmark destination according to Table 8.47. Actions will be dominant over this option. Default: { <i>type fitwindow page 0</i> } if <i>destination</i> , <i>destname</i> , and <i>action</i> are absent.
<i>destname</i>	hypertext string	The name of a destination which has been defined with <code>PDF_add_nameddest()</code> . Destination or <i>destname</i> actions will be dominant over this option.
<i>fontstyle</i>	keyword	Specifies the font style of the bookmark text: <i>normal</i> , <i>bold</i> , <i>italic</i> , <i>bolditalic</i> . Default: <i>normal</i>
<i>hypertext-encoding</i>	keyword	Specifies the encoding for the supplied text (see Section 4.5.4, »String Handling in non-Unicode-capable Languages«, page 91). An empty string is equivalent to <i>unicode</i> . Default: the value of the global <i>hypertextencoding</i> parameter

Table 8.51 Options for PDF\_create\_bookmark()

option	type	explanation
hypertext-format	keyword	Set the format for the supplied text. Possible values are bytes, utf8, utf16, utf16le, utf16be, and auto. Default: the value of the global hypertextformat parameter
index	integer	The index where to insert the bookmark within the parent. Values between 0 and the number of bookmarks of the same level will be used to insert the bookmark at that specific location within the parent. The value -1 can be used to insert the bookmark as the last one on the current level. Default: -1. However, for inserted or resumed pages bookmarks will be placed as if all pages had been generated in their physical order (the bookmarks will reflect the page order).
open	boolean	If false, subordinate bookmarks will not be visible. If true, all children will be folded out. Default: false
parent	bookmark handle	The new bookmark will be specified as a subordinate of the bookmark specified in the handle. If parent=0 a new top-level bookmark will be created. Default: 0
textcolor	color	Specifies the color of the bookmark text. Supported color spaces: none, gray, rgb. Default: rgb {0 0 0} (=black)

### 8.9.6 Document Information Fields

```
void PDF_set_info(PDF *p, const char *key, const char *value)
void PDF_set_info2(PDF *p, const char *key, const char *value, int len)
```

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

**key** (Name string) The name of the document info field, which may be any of the standard names, or an arbitrary custom name (see Table 8.52). There is no limit for the number of custom fields. Regarding the use and semantics of custom document information fields, PDFlib users are encouraged to take a look at the Dublin Core Metadata element set.<sup>1</sup>

**value** (Hypertext string) The string to which the *key* parameter will be set. Acrobat imposes a maximum length of *value* of 255 bytes. Note that due to a bug in Adobe Reader 6 for Windows the & character does not display properly in some info strings.

**len** (Only for PDF\_set\_info2(), and only for the C binding.) Length of *value* (in bytes) for UCS-2 strings. If *len* = 0 a null-terminated string must be provided.

*Details* The supplied info value will only be used for the current document, but not for all documents generated within the same *object* scope.

*Scope* *object, document, page*. If used in *object* scope the supplied values will only be used for the next document.

Table 8.52 Values for the document information field *key*

key	explanation
Subject	Subject of the document
Title	Title of the document
Creator	Software used to create the document (as opposed to the Producer of the PDF output, which is always PDFlib)
Author	Author of the document

1. See [dublincore.org](http://dublincore.org)

Table 8.52 Values for the document information field key

key	explanation
Keywords	Keywords describing the contents of the document
Trapped	Indicates whether trapping has been applied to the document. Allowed values are True, False, and Unknown.
any name other than CreationDate, Producer, and ModDate	User-defined field. PDFlib supports an arbitrary number of fields. A custom field name should only be supplied once. With multiple occurrences of the same field name the last one will be used.

## 8.9.7 Deprecated Hypertext Parameters and Functions

Table 8.53 lists deprecated parameters and values for hypertext features.

Table 8.53 Deprecated document and hypertext parameters

function	key	explanation
set_parameter	openaction	Use the action option with type=open in PDF_begin/end_document()
set_parameter	openmode	Use the openmode option in PDF_begin/end_document()
set_parameter	hidetoolbar hidemenubar hidewindowui fitwindow centerwindow displaydoctitle nonfullscreen- pagemode direction viewarea, viewclip printarea, printclip	Use the viewerpreferences option in PDF_begin/end_document()
set_parameter	bookmarkdest	Use the action, destination, fontstyle, and textcolor options in PDF_create_bookmark() instead.
set_parameter	transition	Use the transition option in PDF_begin/end_page_ext()
set_value	duration	Use the duration option in PDF_begin/end_page_ext()
set_parameter	base	Use the uri option in PDF_begin/end_document()
set_parameter	launchlink:parameters launchlink:operation launchlink:defaultdir	Use the parameters, operation, and defaultdir options in PDF_create_action().

---

```
int PDF_add_bookmark(PDF *p, const char *text, int parent, int open)
int PDF_add_bookmark2(PDF *p, const char *text, int len, int parent, int open)
```

---

Deprecated, use `PDF_create_bookmark()`.

---

```
void PDF_add_note(PDF *p, double llx, double lly, double urx, double ury,
    const char *contents, const char *title, const char *icon, int open)
void PDF_add_note2(PDF *p, double llx, double lly, double urx, double ury,
    const char *contents, int contents_len, const char *title, int title_len,
    const char *icon, int open)
```

---

Deprecated, use `PDF_create_annotation()` with `type=Text`.

---

```
void PDF_attach_file(PDF *p, double llx, double lly, double urx, double ury, const char
    *filename, const char *description, const char *author, const char *mimetype,
    const char *icon)
```

```
void PDF_attach_file2(PDF *p, double llx, double lly, double urx, double ury, const char
    *filename, int len, const char *description, int desc_len, const char *author,
    int author_len, const char *mimetype, const char *icon)
```

---

Deprecated, use *PDF\_create\_annotation()* with *type=FileAttachment*. The *description* parameter corresponds to the *contents* option, the *author* parameter to the *title* option, and the *icon* parameter to the *iconname* option.

---

```
void PDF_add_pdflink(PDF *p, double llx, double lly, double urx, double ury,
    const char *filename, int page, const char *optlist)
```

---

Deprecated, use *PDF\_create\_action()* with *type=GoToR* and *PDF\_create\_annotation()* with *type=Link*.

---

```
void PDF_add_locallink(PDF *p,
    double llx, double lly, double urx, double ury, int page, const char *optlist)
```

---

Deprecated, use *PDF\_create\_action()* with *type=GoTo* and *PDF\_create\_annotation()* with *type=Link*.

---

```
void PDF_add_launchlink(PDF *p, double llx, double lly, double urx, double ury,
    const char *filename)
```

---

Deprecated, use *PDF\_create\_action()* with *type=Launch* and *PDF\_create\_annotation()* with *type=Link*.

---

```
void PDF_add_weblink(PDF *p, double llx, double lly, double urx, double ury, const char *url)
```

---

Deprecated, use *PDF\_create\_action()* with *type=URI* and *PDF\_create\_annotation()* with *type=Link*.

---

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

---

Deprecated, use the *borderstyle* and *linewidth* options in *PDF\_create\_annotation()*.

---

```
void PDF_set_border_color(PDF *p, double red, double green, double blue)
```

---

Deprecated, use the *annotcolor* option in *PDF\_create\_annotation()*.

---

```
void PDF_set_border_dash(PDF *p, double b, double w)
```

---

Deprecated, use the *dasharray* option in *PDF\_create\_annotation()*.

# 8.10 Structure Functions for Tagged PDF

The *tagged* option in *PDF\_begin\_document()* must have been set to *true* in order to generate Tagged PDF; the *lang* option must be provided as well.

```
int PDF_begin_item(PDF *p, const char *tag, const char *optlist)
```

Open a structure element or other content item with attributes supplied as options.

**tag** The item’s element type according to Table 8.54. It must be one of the standard structure types allowed for the current PDF compatibility level, or a pseudo tag.

Table 8.54 Standard item tags

category	tags
grouping	Document, Part, Art, Sect, Div, BlockQuote, Caption, TOC, TOCI, Index, NonStruct, Private
paragraph-like	P, H, H1-H6 (BLSEs)
list	L, LI, Lbl, LBody (BLSEs)
table	Table (BLSE), TR, TH, TD, THead <sup>1</sup> , TBody <sup>1</sup> , TFoot <sup>1</sup>
inline-level	Span, Quote, Note, Reference, BibEntry, Code, (ILSEs)
illustration	Figure, Formula, Form
Japanese	Ruby <sup>1</sup> (grouping), RB <sup>1</sup> , RT <sup>1</sup> , RP <sup>1</sup> , Warichu <sup>1</sup> (grouping), WT <sup>1</sup> , WP <sup>1</sup>
pseudo tags	The following tags create items which are not structure elements:
	Artifact specifies an artifact, to be distinguished from real page content.
	ASpan (accessibility span; will be written to PDF as Span, but must be distinguished from the inline-level item Span) can be used to attach accessibility properties to content which does not belong to a structure element, or which resembles only a fraction of a structure element.
	ReversedChars specifies text in a right-to-left language with reversed character sequence.
	Clip specifies a marked clipping sequence. This is a sequence containing only clipping paths or text in rendering mode 7, but no visible graphics or PDF_save() / PDF_restore().

1. Requires PDF 1.5

**optlist** An option list specifying details of the item according to Table 8.55. All inheritable settings will be inherited to child elements, and therefore need not be repeated. All properties of an item must be set here since they cannot be modified later.

**Returns** An item handle which can be used in subsequent item-related calls.

**Details** This function generates the document’s structure tree, which is essential for Tagged PDF. The position of the new element in the structure tree can be controlled with the *parent* and *index* options. Structure elements can be nested to an arbitrary level. Regular items are not bound to the page where they have been opened, but can be continued on an arbitrary number of pages.

**Scope** *page* for inline items, and for regular items also *document*; must always be paired with a matching *PDF\_end\_item()* call. This function is only allowed in Tagged PDF mode.

Table 8.55 Options for the properties of structure and pseudo tags with `PDF_begin_item()`

option	type	explanation
Alt	hypertext string	(Not for pseudo tags except in PDF 1.5 with ASpan) Alternate description for the content item. It should be provided for figures, images, etc., which cannot be recognized as text. Alternate text for images is required for accessibility. If this option is used in PDF 1.4 mode the inline option must be set to false.
ActualText	hypertext string	(Not for pseudo tags except in PDF 1.5 with ASpan; required for text in fonts which are not Unicode-compatible) Equivalent replacement text for the content item. It should be provided for text content which is represented in some non-standard way, such as ligatures, swash characters in illustrations, drop caps, etc. If this option is used in PDF 1.4 mode the inline option must be set to false.
artifacttype	keyword	(Only for tag=Artifact) Identifies the artifact type of the content item: <i>Pagination</i> , <i>Layout</i> , or <i>Page</i>
Attached	keyword list	(Only for tag=Artifact and artifacttype=Pagination) A list containing one to four of the keywords <i>Top</i> , <i>Bottom</i> , <i>Left</i> , and <i>Right</i>
BBox	rectangle	(Only for tag=Artifact as well as all table and illustration tags; optional, but recommended for reflow) The artifact's bounding box in the default coordinate system (if <i>usercoordinates</i> is false) or the user coordinate system (if <i>usercoordinates</i> is true). If this option has not been supplied PDFlib will automatically create a BBox entry for imported images and PDF pages.
ColSpan	integer	The number of table columns spanned by a cell.
E	hypertext string	(Not for pseudo tags except ASpan; requires PDF 1.5 for structure tags) Abbreviation expansion for the content item. It should be provided for explaining abbreviations and acronyms. Acrobat's Read Aloud feature will consider the expansion text as a separate word even in the absence of explicit word breaks.
hypertext-encoding	keyword	Specifies the encoding for the supplied text (see Section 4.5.4, »String Handling in non-Unicode-capable Languages«, page 91). An empty string is equivalent to unicode. Default: empty string for Unicode-capable language bindings, otherwise auto.
index	integer	(Not for pseudo tags) The index at which to insert the element within the parent. Values between 0 and the current number of children will be used to insert the item at that specific location within the parent. The value -1 can be used to insert the element as the last item. Default: -1
inline	boolean	(Only for tag=ASpan and all inline-level tags) If true, the content item will be written inline, and no structure element will be created. Default: true
Lang	string	(Not for pseudo tags except ASpan) Language identifier for the content item in the format described in Table 8.4 for the lang option. This can be used to override the document's dominating language for individual content items.
parent	item handle	(Not for pseudo tags) The item handle of the element's parent, as returned by another call to <code>PDF_begin_item()</code> . The value 0 refers to the structure tree root. -1 refers to the parent of the least recently opened element on the current page. In other words, <code>parent=-1</code> opens a sibling of the current element. Default: -1
RowSpan	integer	The number of table rows spanned by a cell.
Title	hypertext string	(Not for inline and pseudo tags) Name of the structure element

---

**void PDF\_end\_item(PDF \*p, int id)**

---

Close a structure element or other content item.

**id** The item's handle, which must have been retrieved with `PDF_begin_item()`.

*Details* All inline items must be closed before the end of the page. All regular items must be closed before the end of the document. However, it is strongly recommended to close all regular items as soon as they are completed to reduce memory consumption. An item can only be closed if all of its children have been closed before. After closing an item its parent will become the active item.

*Scope* *page* for inline items, and for regular items also *document*; must always be paired with a matching *PDF\_begin\_item()* call. This function is only allowed in Tagged PDF mode.

---

**void PDF\_activate\_item(PDF \*p, int id)**

---

Activate a previously created structure element or other content item.

**id** The item's handle, which must have been retrieved with *PDF\_begin\_item()*, and must not yet have been closed. Pseudo and inline-level items can not be activated.

*Details* Putting aside a structure element and activating it later gives additional flexibility for efficiently creating Tagged PDF pages even when there are multiple parallel structure branches on a page, e.g. with multi-column layouts or text inserts which interrupt the main text. See Section 7.5.3, »Activating Items for complex Layouts«, page 179, for more details.

*Scope* *document*, *page*; This function is only allowed in Tagged PDF mode.



# A Literature

[1] Adobe Systems Incorporated: PDF Reference, Fourth Edition: Version 1.5. Available as PDF from [partners.adobe.com/asn/tech/pdf/specifications.jsp](http://partners.adobe.com/asn/tech/pdf/specifications.jsp)

[2] The following book by the principal author of PDFlib is only available in German. It discusses a variety of PostScript, PDF and font-related topics:

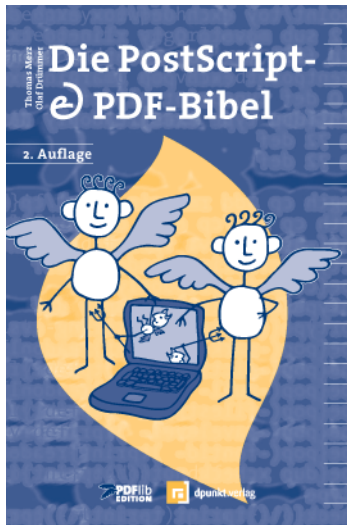
Thomas Merz, Olaf Drümmer: Die PostScript- & PDF-Bibel.

Zweite Auflage. ISBN 3-935320-01-9, PDFlib Edition 2002

PDFlib GmbH, 80331 München, Tal 40, fax +49 • 89 • 29 16 46 86

Freely available as PDF from [www.pdflib.com](http://www.pdflib.com)

Order printed edition by e-mail via [books@pdflib.com](mailto:books@pdflib.com)





# B PDFlib Quick Reference

## General Functions

Function prototype	page
<code>void PDF_boot(void)</code>	187
<code>void PDF_shutdown(void)</code>	187
<code>PDF *PDF_new(void)</code>	188
<code>PDF *PDF_new2(void (*errorhandler)(PDF *p, int errortype, const char *msg), void* (*allocproc)(PDF *p, size_t size, const char *caller), void* (*reallocproc)(PDF *p, void *mem, size_t size, const char *caller), void (*freeproc)(PDF *p, void *mem), void *opaque)</code>	188
<code>void PDF_delete(PDF *p)</code>	189
<code>int PDF_begin_document(PDF *p, const char *filename, int len, const char *optlist)</code>	190
<code>int PDF_begin_document_callback(PDF *p, const char *filename, int len, size_t (*writeproc)(PDF *p, void *data, size_t size), const char *optlist)</code>	190
<code>const char *PDF_get_buffer(PDF *p, long *size)</code>	195
<code>void PDF_begin_page_ext(PDF *p, double width, double height, const char *optlist)</code>	195
<code>void PDF_end_page_ext(PDF *p, const char *optlist)</code>	196
<code>void PDF_suspend_page(PDF *p, const char *optlist)</code>	197
<code>void PDF_resume_page(PDF *p, const char *optlist)</code>	198
<code>double PDF_get_value(PDF *p, const char *key, double modifier)</code>	198
<code>void PDF_set_value(PDF *p, const char *key, double value)</code>	199
<code>const char *PDF_get_parameter(PDF *p, const char *key, double modifier)</code>	199
<code>void PDF_set_parameter(PDF *p, const char *key, const char *value)</code>	199
<code>void PDF_create_pvf(PDF *p, const char *filename, int len, const void *data, size_t size, const char *optlist)</code>	199
<code>int PDF_delete_pvf(PDF *p, const char *filename, int len)</code>	200
<code>int PDF_get_errnum(PDF *p)</code>	201
<code>const char *PDF_get_errmsg(PDF *p)</code>	201
<code>const char *PDF_get_apiname(PDF *p)</code>	201
<code>void *PDF_get_opaque(PDF *p)</code>	202
<code>const char *PDF_utf16_to_utf8(PDF *p, const char *utf16string, int len, int *size)</code>	202
<code>const char *PDF_utf8_to_utf16(PDF *p, const char *utf8string, const char *ordering, int *size)</code>	202

## Font Functions

Function prototype	page
<code>int PDF_load_font(PDF *p, const char *fontname, int len, const char *encoding, const char *optlist)</code>	205
<code>void PDF_setfont(PDF *p, int font, double fontsize)</code>	207
<code>void PDF_begin_font(PDF *p, char *fontname, int reserved, double a, double b, double c, double d, double e, double f, const char *optlist)</code>	207
<code>void PDF_end_font(PDF *p)</code>	208
<code>void PDF_begin_glyph(PDF *p, char *glyphname, double wx, double llx, double lly, double urx, double ury)</code>	208
<code>void PDF_end_glyph(PDF *p)</code>	208
<code>void PDF_encoding_set_char(PDF *p, const char *encoding, int slot, const char *glyphname, int uv)</code>	209

## Text Output Functions

<b>Function prototype</b>	<b>page</b>
<code>void PDF_set_text_pos(PDF *p, double x, double y)</code>	211
<code>void PDF_show(PDF *p, const char *text)</code>	211
<code>void PDF_xshow(PDF *p, const char *text, int len, const double *xadvancelist)</code>	211
<code>void PDF_show_xy(PDF *p, const char *text, double x, double y)</code>	212
<code>void PDF_continue_text(PDF *p, const char *text)</code>	212
<code>void PDF_fit_textline(PDF *p, const char *text, int len, double x, double y, const char *optlist)</code>	213
<code>double PDF_stringwidth(PDF *p, const char *text, int font, double fontsize)</code>	216
<code>int PDF_create_textflow(PDF *p, const char *text, int len, const char *optlist)</code>	216
<code>const char *PDF_fit_textflow(PDF *p, int textflow, double llx, double lly, double urx, double ury, const char *optlist)</code>	218
<code>double PDF_info_textflow(PDF *p, int textflow, const char *keyword)</code>	218
<code>void PDF_delete_textflow(PDF *p, int textflow)</code>	219

## Graphics Functions

<b>Function prototype</b>	<b>page</b>
<code>void PDF_setdash(PDF *p, double b, double w)</code>	224
<code>void PDF_setdashpattern(PDF *p, const char *optlist)</code>	224
<code>void PDF_setflat(PDF *p, double flatness)</code>	224
<code>void PDF_setlinejoin(PDF *p, int linejoin)</code>	225
<code>void PDF_setlinecap(PDF *p, int linecap)</code>	225
<code>void PDF_setmiterlimit(PDF *p, double miter)</code>	226
<code>void PDF_setlinewidth(PDF *p, double width)</code>	226
<code>void PDF_initgraphics(PDF *p)</code>	226
<code>void PDF_save(PDF *p)</code>	226
<code>void PDF_restore(PDF *p)</code>	227
<code>void PDF_translate(PDF *p, double tx, double ty)</code>	227
<code>void PDF_scale(PDF *p, double sx, double sy)</code>	227
<code>void PDF_rotate(PDF *p, double phi)</code>	228
<code>void PDF_skew(PDF *p, double alpha, double beta)</code>	228
<code>void PDF_concat(PDF *p, double a, double b, double c, double d, double e, double f)</code>	228
<code>void PDF_setmatrix(PDF *p, double a, double b, double c, double d, double e, double f)</code>	229
<code>int PDF_create_gstate(PDF *p, const char *optlist)</code>	229
<code>void PDF_set_gstate(PDF *p, int gstate)</code>	230
<code>void PDF_moveto(PDF *p, double x, double y)</code>	230
<code>void PDF_lineto(PDF *p, double x, double y)</code>	231
<code>void PDF_curveto(PDF *p, double x1, double y1, double x2, double y2, double x3, double y3)</code>	231
<code>void PDF_circle(PDF *p, double x, double y, double r)</code>	231
<code>void PDF_arc(PDF *p, double x, double y, double r, double alpha, double beta)</code>	231
<code>void PDF_arcn(PDF *p, double x, double y, double r, double alpha, double beta)</code>	232
<code>void PDF_rect(PDF *p, double x, double y, double width, double height)</code>	232
<code>void PDF_closepath(PDF *p)</code>	232
<code>void PDF_stroke(PDF *p)</code>	233

<b>Function prototype</b>	<b>page</b>
<code>void PDF_closepath_stroke(PDF *p)</code>	233
<code>void PDF_fill(PDF *p)</code>	233
<code>void PDF_fill_stroke(PDF *p)</code>	233
<code>void PDF_closepath_fill_stroke(PDF *p)</code>	234
<code>void PDF_clip(PDF *p)</code>	234
<code>void PDF_endpath(PDF *p)</code>	234

## Color Functions

<b>Function prototype</b>	<b>page</b>
<code>void PDF_setcolor(PDF *p, const char *fstype, const char *colorspace, double c1, double c2, double c3, double c4)</code> 237	
<code>int PDF_makespotcolor(PDF *p, const char *spotname, int reserved)</code>	238
<code>int PDF_load_iccprofile(PDF *p, const char *profilename, int len, const char *optlist)</code>	239
<code>int PDF_begin_pattern(PDF *p, double width, double height, double xstep, double ystep, int painttype)</code>	240
<code>void PDF_end_pattern(PDF *p)</code>	241
<code>int PDF_shading_pattern(PDF *p, int shading, const char *optlist)</code>	241
<code>void PDF_shfill(PDF *p, int shading)</code>	241
<code>int PDF_shading(PDF *p, const char *shtype, double xo, double yo, double x1, double y1, double c1, double c2, double c3, double c4, const char *optlist)</code>	242

## Image Functions

<b>Function prototype</b>	<b>page</b>
<code>int PDF_load_image(PDF *p, const char *imagetype, const char *filename, int len, const char *optlist)</code>	243
<code>void PDF_close_image(PDF *p, int image)</code>	246
<code>void PDF_fit_image(PDF *p, int im, double x, double y, const char *optlist)</code>	247
<code>int PDF_begin_template(PDF *p, double width, double height)</code>	249
<code>void PDF_end_template(PDF *p)</code>	249
<code>void PDF_add_thumbnail(PDF *p, int image)</code>	249

## PDF Import (PDI) Functions

<b>Function prototype</b>	<b>page</b>
<code>int PDF_open_pdi(PDF *p, const char *filename, const char *optlist, int len)</code>	251
<code>int PDF_open_pdi_callback(PDF *p, void *opaque, size_t filesize, size_t (*readproc)(void *opaque, void *buffer, size_t size), int (*seekproc)(void *opaque, long offset), const char *optlist)</code>	252
<code>void PDF_close_pdi(PDF *p, int doc)</code>	253
<code>int PDF_open_pdi_page(PDF *p, int doc, int pagenumber, const char *optlist)</code>	253
<code>void PDF_close_pdi_page(PDF *p, int page)</code>	254
<code>void PDF_fit_pdi_page(PDF *p, int page, double x, double y, const char *optlist)</code>	254
<code>int PDF_process_pdi(PDF *p, int doc, int page, const char *optlist)</code>	255
<code>double PDF_get_pdi_value(PDF *p, const char *key, int doc, int page, int reserved)</code>	255
<code>const char *PDF_get_pdi_parameter(PDF *p, const char *key, int doc, int page, int reserved, int *len)</code>	256

# Block Filling (PPS) Functions

Function prototype	page
<i>int PDF_fill_textblock(PDF *p, int page, const char *blockname, const char *text, int len, const char *optlist)</i>	258
<i>int PDF_fill_imageblock(PDF *p, int page, const char *blockname, int image, const char *optlist)</i>	259
<i>int PDF_fill_pdfblock(PDF *p, int page, const char *blockname, int contents, const char *optlist)</i>	259

# Hypertext Functions

Function prototype	page
<i>int PDF_create_action(PDF *p, const char *type, const char *optlist)</i>	261
<i>void PDF_add_nameddest(PDF *p, const char *name, int len, const char *optlist)</i>	264
<i>void PDF_create_annotation(PDF *p, double llx, double lly, double urx, double ury, const char *type, const char *optlist)</i>	265
<i>void PDF_create_field(PDF *p, double llx, double lly, double urx, double ury, const char *name, int len, const char *type, const char *optlist)</i>	269
<i>void PDF_create_fieldgroup(PDF *p, const char *name, int len, const char *optlist)</i>	273
<i>int PDF_create_bookmark(PDF *p, const char *text, int len, const char *optlist)</i>	274
<i>void PDF_set_info(PDF *p, const char *key, const char *value)</i>	275

# Tagged PDF and Structure Functions

Function prototype	page
<i>int PDF_begin_item(PDF *p, const char *tag, const char *optlist)</i>	278
<i>void PDF_end_item(PDF *p, int id)</i>	279
<i>void PDF_activate_item(PDF *p, int id)</i>	280

## Parameters and Values

<b>category</b>	<b>function</b>	<b>keys</b>
<b>setup</b>	<i>set_parameter</i>	<i>resourcefile, SearchPath, license, licensefile, warning, asciifile, trace, tracefile, tracemsg</i>
	<i>set_value</i>	<i>compress</i>
<b>versioning</b>	<i>get_value</i>	<i>major, minor, revision</i>
	<i>get_parameter</i>	<i>version</i>
<b>page</b>	<i>get_value</i>	<i>pagewidth, pageheight</i>
<b>font</b>	<i>set_parameter</i>	<i>FontAFM, FontPFM, FontOutline, Encoding, fontwarning, kerning, autosubsetting, autocidfont, textformat, unicodemap</i>
	<i>get_parameter</i>	<i>fontname, fontencoding, fontstyle, textformat</i>
	<i>set_value</i>	<i>subsetlimit, subsetminsize</i>
<b>text</b>	<i>get_value</i>	<i>ascender, capheight, descender, font, fontsize, fontmaxcode, monospace</i>
	<i>set_value</i>	<i>leading, textrise, horizscaling, textrendering, charspacing, wordspace, italicangle</i>
	<i>get_value</i>	<i>leading, textrise, horizscaling, textrendering, charspacing, wordspace, textx, texty, italicangle</i>
	<i>set_parameter</i>	<i>autospace, underline, overline, strikeout, kerning, glyphwarning</i>
	<i>get_parameter</i>	<i>underline, overline, strikeout, fontstyle</i>
<b>graphics</b>	<i>set_parameter</i>	<i>fillrule, topdown</i>
	<i>get_parameter</i>	<i>scope</i>
	<i>get_value</i>	<i>currentx, currenty</i>
<b>color</b>	<i>set_parameter</i>	<i>iccwarning, honoricprofile, ICCProfile, StandardOutputIntent, renderingintent, preserveoldpantone names, spotcolorlookup</i>
	<i>set_value</i>	<i>defaultgray, defaultrgb, defaultcmymk, setcolor:iccprofilegray, setcolor:iccprofilergb, setcolor:iccprofilecmyk</i>
	<i>get_value</i>	<i>image:iccprofile, icccomponents</i>
<b>image</b>	<i>get_value</i>	<i>imagewidth, imageheight, resx, resy</i>
	<i>set_parameter</i>	<i>imagewarning</i>
<b>PDI</b>	<i>get_parameter</i>	<i>pdi</i>
	<i>set_parameter</i>	<i>pdiwarning</i>
	<i>get_pdi_value</i>	<i>/Root/Pages/Count, /Rotate, version, width, height CropBox, BleedBox, ArtBox, TrimBox: these must be followed by a slash '/' character and one of llx, lly, urx, ury, for example: CropBox/llx</i>
	<i>get_pdi_parameter</i>	<i>filename, /Info/&lt;key&gt;, vdp/Blocks/&lt;blockname&gt;/&lt;propertyname&gt;, vdp/Blocks/&lt;blockname&gt;/Custom/&lt;propertyname&gt;</i>
<b>hypertext</b>	<i>set_parameter</i>	<i>hypertextformat, hypertextencoding, usercoordinates</i>
	<i>get_parameter</i>	<i>hypertextformat</i>

# C Revision History

<i>Date</i>	<i>Changes</i>
<i>June 18, 2004</i>	► Major changes for PDFlib 6
<i>January 21, 2004</i>	► Minor additions and corrections for PDFlib 5.0.3
<i>September 15, 2003</i>	► Minor additions and corrections for PDFlib 5.0.2; added block specification
<i>May 26, 2003</i>	► Minor updates and corrections for PDFlib 5.0.1
<i>March 26, 2003</i>	► Major changes and rewrite for PDFlib 5.0.0
<i>June 14, 2002</i>	► Minor changes for PDFlib 4.0.3 and extensions for the .NET binding
<i>January 26, 2002</i>	► Minor changes for PDFlib 4.0.2 and extensions for the IBM eServer edition
<i>May 17, 2001</i>	► Minor changes for PDFlib 4.0.1
<i>April 1, 2001</i>	► Documents PDI and other features of PDFlib 4.0.0
<i>February 5, 2001</i>	► Documents the template and CMYK features in PDFlib 3.5.0
<i>December 22, 2000</i>	► ColdFusion documentation and additions for PDFlib 3.03; separate COM edition of the manual
<i>August 8, 2000</i>	► Delphi documentation and minor additions for PDFlib 3.02
<i>July 1, 2000</i>	► Additions and clarifications for PDFlib 3.01
<i>Feb. 20, 2000</i>	► Changes for PDFlib 3.0
<i>Aug. 2, 1999</i>	► Minor changes and additions for PDFlib 2.01
<i>June 29, 1999</i>	► Separate sections for the individual language bindings ► Extensions for PDFlib 2.0
<i>Feb. 1, 1999</i>	► Minor changes for PDFlib 1.0 (not publicly released)
<i>Aug. 10, 1998</i>	► Extensions for PDFlib 0.7 (only for a single customer)
<i>July 8, 1998</i>	► First attempt at describing PDFlib scripting support in PDFlib 0.6
<i>Feb. 25, 1998</i>	► Slightly expanded the manual to cover PDFlib 0.5
<i>Sept. 22, 1997</i>	► First public release of PDFlib 0.4 and this manual



# Index

## O-9

16-bit encodings 89

8-bit encodings 83

## A

Acrobat plugin for creating blocks 141

action lists 49

Adobe Font Metrics (AFM) 74

AFM (Adobe Font Metrics) 74

alignment (position option) 214

All spot color name 238

alpha channel 128

alphaissshape gstate option 229

annotations 90

antialias option 242

API (Application Programming Interface)  
reference 185

ArtBox 60, 256, 257

artificial font styles 99

AS/400 55

ascender 97

ascender parameter 204

asciifile parameter 56, 187

Asian FontPack 101

attachments 90

Author field 275

auto text format 94

autocidfont parameter 80, 81, 204

autospace parameter 209

autosubsetting parameter 81, 204

availability of PDFlib 19

## B

baseline compression 126

Bézier curve 231

Big Five 105

bindings 19

BleedBox 60, 256, 257

blendmode gstate option 229

blocks 141

plugin 141

properties 143

BMP 128

builtin encoding 86

byte order mark (BOM) 93

byte text format 94

bytes: see *hypertextformat*

byteserving 170

## C

C binding 24

memory management 27

C++ binding 28

memory management 29

capheight 97

capheight parameter 204

categories of resources 52

CCITT 128

CCSID 83, 85

CFF (Compact Font Format) 71

character metrics 97

character names 75

character references 94

character sets 83

characters per inch 97

charref parameter 209

charspacing parameter 209

Chinese 101, 104, 105

CIE L\*a\*b\* color space 67

CJK (Chinese, Japanese, Korean)

custom fonts 105

standard fonts 101

Windows code pages 105

clip 60

CMaps 101, 104

CMYK color 63

Cobol binding 20

code page

Microsoft Windows 1250-1258 84

Unicode-based 89

color 63

color functions 237

color values in option lists 49

COM (Component Object Model) binding 24

commercial license 10

compress parameter 187

coordinate range 60

coordinate system 57

metric 57

top-down 58

copyoutputintent option 175

core fonts 79

CPI (characters per inch) 97

Creator field 275

CropBox 60, 256, 257

current point 61

currentx and currenty parameter 97, 230

custom encoding 85

## D

- dash pattern for lines* 224
- default coordinate system* 57
- defaultgray/rgb/cmyk parameters* 69
- demo stamp* 9
- descender* 97
- descender parameter* 204
- descriptor* 80
- document and page functions* 189
- document information fields* 90, 275
- downsampling* 126
- dpi calculations* 126
- Dublin Core* 275

## E

- EBCDIC* 55
- ebcdic encoding* 84
- ebcdicutf8: see hypertextformat*
- EJB (Enterprise Java Beans)* 31
- embedded systems* 19
- embedding fonts* 79
- encoding* 83
  - CJK* 103
  - custom* 85
  - fetching from the system* 83
  - for hypertext* 93
- Encoding parameter* 204
- encryption* 168
- environment variable PDFLIBRESOURCE* 54
- error handling* 46
  - API* 189
- eServer zSeries and iSeries* 55
- EUDC (end-user defined characters)* 106
- EUDC fonts* 75
- Euro character* 87
- evaluation stamp* 9
- exceptions* 46
- explicit graphics state* 229
- explicit transparency* 129
- extendo and extend1 options* 242

## F

- fast Web view* 191
- features of PDFlib* 15
- file attachments* 90
- filename parameter for PDI* 257
- fill* 60
- fillrule parameter* 233
- flatness gstate option* 229
- font metrics* 97
- font parameter* 204
- font style names for Windows* 76
- font styles* 99
- font subsetting* 81
- FontAFM parameter* 204
- fontencoding parameter* 204

- fontmaxcode parameter* 87, 204
- fontname parameter* 204
- FontOutline parameter* 204
- FontPFM parameter* 204
- fonts*
  - AFM files* 74
  - Asian fontpack* 101
  - descriptor* 80
  - embedding* 79
  - glyph names* 75
  - legal aspects of embedding* 80
  - monospaced* 97
  - OpenType* 71
  - PDF core set* 79
  - PFA files* 74
  - PFB files* 74
  - PFM files* 74
  - PostScript* 71, 74
  - resource configuration* 51
  - TrueType* 71
  - Type 1* 74
  - Type 3 (user-defined) fonts* 77
  - Type 3* 77
  - Unicode support* 89
  - user-defined (Type 3)* 77
- fontsize parameter* 204
- FontSpecific encoding* 86
- fontstyle parameter* 204
- fontwarning parameter* 47, 204
- form fields: converting to blocks* 150
- form XObjects* 61
- function scopes* 45

## G

- gaiji characters* 72
- GBK* 105
- GIF* 127
- glyph id addressing* 87
- glyphwarning parameter* 210
- gradients* 64
- graphics functions* 224
- graphics state*
  - explicit* 229
- graphics state functions* 224
- grid.pdf* 57
- gstate* 241

## H

- HKS colors* 66
- honoriccprofile parameter* 243
- horizontal writing mode* 102, 103
- horizscaling parameter* 210
- host encoding* 83
- host fonts* 78
- HostFont parameter* 204
- hypertextencoding parameter* 93, 261
- hypertextformat parameter* 92, 261

## I

- IBM eServer 55
- ICC-based color 63
- icccomponents parameter 240
- ICCProfile parameter 240
- iccwarning parameter 240
- ignoremask 130
- image data, re-using 125
- image file formats 126
- image functions 243
- image mask 128, 130
- image scaling 126
- image:iccprofile parameter 69, 243
- imagewarning parameter 126, 243
- imagewidth and imageheight parameters 243
- implicit transparency 129
- import functions for PDF 251
- inch 57
- in-core PDF generation 54
- indexed color 63
- info fields 275
- Info keys in imported PDF documents 257
- inline images 125
- invisible text 210
- iSeries 55
- ISO 10646 89
- ISO 15930 171
- ISO 8859-2 to -15 84
- italicangle parameter 210

## J

- Japanese 101, 104, 105
- Java application servers 31
- Java binding 30
  - EJB 31
  - javadoc 30
  - package 30
  - servlet 31
- JFIF 126
- Johab 105
- JPEG 126

## K

- kerneling 98
- kerneling parameter 98, 210
- Keywords field 276
- Korean 101, 104, 105

## L

- landscape mode 197
- language bindings: see bindings
- layers and PDI 135
- leading 97
- leading parameter 210
- license parameter 187

- licensefile parameter 187
- licensing PDFlib and PDI 9
- line spacing 97
- linearized PDF 170, 191
- linecap gstate option 229
- linejoin gstate option 229
- lines
  - tashed and patterned 224
- linewidth gstate option 229
- list values in option lists 49
- LWFN (LaserWriter Font) 74
- LZW compression 127

## M

- Mac OS
  - UPR configuration 52
- macroman encoding 83, 84
- macroman\_euro encoding 88
- major parameter 187
- makepsres utility 51
- mask 129
- masked 129
- masking images 128
- masterpassword 169
- MediaBox 60
- memory management
  - API 189
  - in C 27
  - in C++ 29
- memory, generating PDF documents in 54
- metadata 193
- metric coordinates 57
- metrics 97
- millimeters 57
- minor parameter 187
- mirroring 228
- miterlimit gstate option 229
- monospace parameter 204
- monospaced fonts 97
- multi-page image files 131

## N

- N option 242
- nagger 9
- .NET binding 33
- None spot color name 238
- note annotations 90

## O

- opacityfill gstate option 229
- opacitystroke gstate option 229
- OpenType fonts 71
- optimized PDF 170
- option lists 48
- outline text 210
- output accuracy 60

- output condition for PDF/X* 172
- output intent for PDF/X* 172
- overline parameter* 100, 210
- overprintfill gstate option* 229
- overprintmode gstate option* 229
- overprintstroke gstate option* 229

## P

- page* 131
- page descriptions* 57
- page formats* 59
- page size formats* 59
  - limitations in Acrobat* 59
- page-at-a-time download* 170
- PANTONE colors* 65
- parameter handling functions* 198
- passwords* 168
- path* 60
  - painting and clipping* 233
- patterns* 63
- PDF import functions* 251
- PDF import library (PDI)* 133, 251
- PDF/X* 171
  - importing PDF documents* 174
  - output intent* 255
- PDF\_activate\_item()* 280
- PDF\_add\_nameddest()* 264
- PDF\_add\_thumbnail()* 249
- PDF\_arc()* 231
- PDF\_arcn()* 232
- PDF\_begin\_document()* 190
- PDF\_begin\_font()* 207
- PDF\_begin\_glyph()* 208
- PDF\_begin\_item()* 278
- PDF\_begin\_layer()* 236
- PDF\_begin\_page\_ext()* 195, 196
- PDF\_begin\_pattern* 240
- PDF\_begin\_template()* 249
- PDF\_boot()* 187
- PDF\_circle()* 231
- PDF\_clip()* 234
- PDF\_close\_image()* 246
- PDF\_close\_pdi()* 253
- PDF\_close\_pdi\_page()* 254
- PDF\_closepath()* 232
- PDF\_closepath\_fill\_stroke()* 234
- PDF\_closepath\_stroke()* 233
- PDF\_concat()* 228
- PDF\_continue\_text()* 212
- PDF\_continue\_text2()* 212
- PDF\_create\_action()* 261
- PDF\_create\_annotation()* 265
- PDF\_create\_bookmark()* 274
- PDF\_create\_field()* 269
- PDF\_create\_fieldgroup()* 273
- PDF\_create\_gstate()* 229
- PDF\_create\_pvf()* 199
- PDF\_create\_textflow()* 216

- PDF\_curveto()* 231
- PDF\_define\_layer()* 234
- PDF\_delete()* 189
- PDF\_delete\_dl()* 189
- PDF\_delete\_pvf()* 200
- PDF\_delete\_textflow()* 219
- PDF\_encoding\_set\_char()* 209
- PDF\_end\_document()* 192
- PDF\_end\_font()* 208
- PDF\_end\_glyph()* 208
- PDF\_end\_item()* 279
- PDF\_end\_layer()* 236
- PDF\_end\_pattern()* 241
- PDF\_end\_template()* 249
- PDF\_endpath()* 234
- PDF\_fill()* 233
- PDF\_fill\_imageblock()* 259
- PDF\_fill\_pdfblock()* 259
- PDF\_fill\_stroke()* 233
- PDF\_fill\_textblock()* 258
- PDF\_fit\_image()* 247
- PDF\_fit\_pdi\_page()* 254
- PDF\_fit\_textflow()* 218
- PDF\_fit\_textline()* 213
- PDF\_get\_apiname()* 201
- PDF\_get\_buffer()* 55, 195
- PDF\_get\_errmsg()* 201
- PDF\_get\_errnum()* 201
- PDF\_get\_opaque()* 202
- PDF\_get\_parameter()* 199
- PDF\_get\_pdi\_parameter()* 256
- PDF\_get\_pdi\_value()* 255
- PDF\_get\_value()* 198
- PDF\_info\_textflow()* 218
- PDF\_initgraphics()* 226
- PDF\_lineto()* 231
- PDF\_load\_font()* 205
- PDF\_load\_iccprofile()* 239
- PDF\_load\_image()* 243
- PDF\_makespotcolor()* 238
- PDF\_moveto()* 230
- PDF\_new()* 188
- PDF\_new\_dl()* 188
- PDF\_newz()* 188
- PDF\_open\_pdi()* 251
- PDF\_open\_pdi\_callback()* 252
- PDF\_open\_pdi\_page()* 253
- PDF\_process\_pdi()* 255
- PDF\_rect()* 232
- PDF\_restore()* 227
- PDF\_resume\_page()* 198
- PDF\_rotate()* 228
- PDF\_save()* 226
- PDF\_scale()* 227
- PDF\_set\_gstate()* 230
- PDF\_set\_info()* 275
- PDF\_set\_info2()* 275
- PDF\_set\_layer\_dependency()* 235

- PDF\_set\_parameter()* 54, 199
- PDF\_set\_text\_pos()* 211
- PDF\_set\_value()* 199
- PDF\_setcolor()* 237
- PDF\_setdash()* 224
- PDF\_setdashpattern()* 224
- PDF\_setflat()* 224
- PDF\_setfont()* 207
- PDF\_setlinecap()* 225
- PDF\_setlinejoin()* 225
- PDF\_setlinewidth()* 226
- PDF\_setmatrix()* 229
- PDF\_setmiterlimit()* 226
- PDF\_shading()* 242
- PDF\_shading\_pattern()* 241
- PDF\_shfill()* 241
- PDF\_show()* 211
- PDF\_show\_xy()* 212
- PDF\_show\_xy2()* 212
- PDF\_show2()* 211
- PDF\_shutdown()* 187
- PDF\_skew()* 228
- PDF\_stringwidth()* 216
- PDF\_stringwidth2()* 216
- PDF\_stroke()* 233
- PDF\_suspend\_page()* 197
- PDF\_translate()* 227
- PDF\_utf16\_to\_utf8()* 202
- PDF\_utf8\_to\_utf16()* 202
- PDF\_xshow()* 211
- PDFlib*
  - features 15
  - program structure 45
- PDFlib Personalization Server* 141, 258
- pdflib.upr* 54
- PDFLIBRESOURCE* environment variable 54
- pdfx* parameter for PDI 175, 257
- PDI* 133, 251
- pdi* parameter 257
- pdiusebox* 134
- pdiusebox* parameter 257
- pdiwarning* parameter 135, 257
- Perl* binding 33
- permissions 168, 169
- PFA* (Printer Font ASCII) 74
- PFB* (Printer Font Binary) 74
- PFM* (Printer Font Metrics) 74
- PHP* binding 34
- platforms 19
- plugin for creating blocks 141
- PNG* 126, 129
- Portable Document Format Reference Manual* 281
- PostScript* fonts 71, 74
- PPS* (PDFlib Personalization Server) 141, 258
- preserveoldpantonenames* parameter 237
- print\_glyphs.ps* 75
- Printer Font ASCII* (PFA) 74
- Printer Font Binary* (PFB) 74

- Printer Font Metrics* (PFM) 74
- program structure 45
- Python* binding 38

## R

- r0* and *r1* options 242
- raster images
  - functions 243
- raw image data 128
- REALbasic* binding 39
- rectangles in option lists 49
- reflection 228
- rendering intents 67
- renderingintent* *gstate* option 230
- renderingintent* option 67
- renderingintent* parameter 243
- resource category 52
- resourcefile* parameter 54, 187
- resx* and *resy* parameter 243
- RGB* color 63
- Rotate entry in imported PDF pages 256
- rotating objects 58
- RPG* binding 39

## S

- S/390* 55
- scaling images 126
- scope parameter 187
- scopes 45
- SearchPath* parameter 52, 187, 204
- security 168
- separation color space 63
- servlet 31
- setcolor*
  - iccprofilegray/rgb/cmyk* parameters 69
- setcolor:iccprofilegray/rgb/cmyk* parameters 240
- setup functions 187
- shadings 64
- Shift-JIS* 105
- skewing 228
- smooth blends 64
- smoothness* *gstate* option 230
- soft mask 128
- SPIFF* 127
- spot color (separation color space) 63, 64
- spotcolorlookup* parameter 237
- sRGB* color space 68
- standard output 190
- standard output conditions for PDF/X 173
- standard page sizes 59
- StandardOutputIntent* parameter 240
- stdout* channel 190
- strikeout* parameter 100, 210
- stroke 60
- strokeadjust* *gstate* option 230
- structure of *PDFlib* programs 45
- style names for Windows 76

Subject field 275  
subpath 60  
subscript 97, 210  
subsetlimit parameter 82, 204  
subsetminsize parameter 81, 204  
superscript 97, 210  
Symbol font 86  
system encoding support 83

## T

Tcl binding 43  
templates 61  
temporary disk space requirements 170  
text functions 204  
text metrics 97  
text position 97  
text variations 97  
textformat parameter 92, 210  
textknockout gstate option 230  
textrendering parameter 100, 210  
textrise parameter 210  
textx and texty parameter 97, 103, 210  
thumbnails 249  
TIFF 127  
    multi-page 131  
Title field 275  
top-down coordinates 58  
topdown parameter 58, 190  
ToUnicode CMap 73, 89  
trace, tracefile, tracemsg parameters 187  
transparency 128  
    problems with 130  
Trapped field 276  
TrimBox 60, 256, 257  
TrueType fonts 71  
TTC (TrueType Collection) 75, 105  
TTF (TrueType font) 71  
Type 1 fonts 74  
Type 3 (user-defined) fonts 77

## U

U+XXXX encoding 89  
UHC 105

underline parameter 100, 210  
Unicode 89  
unicodemap parameter 90, 204  
units 57  
UPR (Unix PostScript Resource) 51  
    file format 52  
    file searching 54  
user space 57  
usercoordinates parameter 57, 261  
user-defined (Type 3) fonts 77  
userpassword 169  
utf16: see hypertextformat  
utf16be: see hypertextformat  
utf16le: see hypertextformat  
utf8: see hypertextformat

## V

value: see parameter  
Variable Data Processing with blocks 141, 258  
vdp/Block parameters for PDI 257  
vdp/blockcount parameter for PDI 257  
version parameter 187  
version parameter for PDI 257  
vertical writing mode 102, 103

## W

warning parameter 47, 201  
web-optimized PDF 170, 191  
width and height parameters for PDI 256  
winansi encoding 84  
wordspacing parameter 211  
writing modes 102, 103

## X

XMP metadata 193  
XObjects 61

## Z

ZapfDingbats font 86  
zSeries 55