

PDFlib, PDFlib+PDI, PPS

A library for generating PDF on the fly
Version 8.0.5

Tutorial

Edition for Cobol, C, C++, Java, Objective-C,
Perl, PHP, Python, RPG, Ruby, and Tcl



Copyright © 1997–2012 PDFlib GmbH and Thomas Merz. All rights reserved.
PDFlib users are granted permission to reproduce printed or digital copies of this manual for internal use.

PDFlib GmbH
Franziska-Bilek-Weg 9, 80339 München, Germany
www.pdfli**b**.com
phone +49 • 89 • 452 33 84-0
fax +49 • 89 • 452 33 84-99

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

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

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

PDFlib and the PDFlib logo are 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, PostScript, and XMP are trademarks of Adobe Systems Inc. AIX, IBM, OS/390, WebSphere, iSeries, and zSeries are trademarks of International Business Machines Corporation. ActiveX, Microsoft, OpenType, and Windows 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
GIF image decoder, Copyright © 1990-1994 David Koblas
PNG image reference library (libpng), Copyright © 1998-2004 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)
Independent JPEG Group's JPEG software, Copyright © 1991-1998, Thomas G. Lane
Cryptographic software, Copyright © 1998-2002 The OpenSSL Project (www.openssl.org)
Expat XML parser, Copyright © 1998, 1999, 2000 Thai Open Source Software Center Ltd
ICU International Components for Unicode, Copyright © 1995-2009 International Business Machines Corporation and others
Reference sRGB ICC color profile data, Copyright (c) 1998 Hewlett-Packard Company

PDFlib contains the RSA Security, Inc. MD5 message digest algorithm.



Contents

o Applying the PDFlib License Key 9

1 Introduction 13

- 1.1 Roadmap to Documentation and Samples 13
- 1.2 PDFlib Programming 15
- 1.3 What's new in PDFlib/PDFlib+PDI/PPS 8? 17
 - 1.3.1 PDF Features for Acrobat 9 17
 - 1.3.2 Font Handling and Text Output 18
 - 1.3.3 PDFlib Block Plugin and PDFlib Personalization Server (PPS) 19
 - 1.3.4 Other important Features 20
- 1.4 Features in PDFlib 21
- 1.5 Additional Features in PDFlib+PDI 24
- 1.6 Additional Features in PPS 25
- 1.7 Availability of Features in different Products 26

2 PDFlib Language Bindings 27

- 2.1 Data Types for Language Bindings 27
- 2.2 Cobol Binding 28
- 2.3 COM Binding 29
- 2.4 C Binding 30
- 2.5 C++ Binding 33
- 2.6 Java Binding 36
- 2.7 .NET Binding 39
- 2.8 Objective-C Binding 40
- 2.9 Perl Binding 42
- 2.10 PHP Binding 44
- 2.11 Python Binding 46
- 2.12 REALbasic Binding 47
- 2.13 RPG Binding 48
- 2.14 Ruby Binding 50
- 2.15 Tcl Binding 52

3 Creating PDF Documents 53

- 3.1 General PDFlib Programming Aspects 53
 - 3.1.1 Exception Handling 53
 - 3.1.2 The PDFlib Virtual File System (PVF) 55
 - 3.1.3 Resource Configuration and File Search 56
 - 3.1.4 Generating PDF Documents in Memory 61

- 3.1.5 Large PDF Documents **62**
- 3.1.6 Using PDFlib on EBCDIC-based Platforms **62**
- 3.2 Page Descriptions 64**
 - 3.2.1 Coordinate Systems **64**
 - 3.2.2 Page Size **66**
 - 3.2.3 Direct Paths and Path Objects **67**
 - 3.2.4 Templates **69**
 - 3.2.5 Referenced Pages from an external PDF Document **70**
- 3.3 Encrypted PDF 72**
 - 3.3.1 PDF Security Features **72**
 - 3.3.2 Protecting Documents with PDFlib **73**
- 3.4 Web-Optimized (Linearized) PDF 75**
- 3.5 Working with Color 76**
 - 3.5.1 Patterns and Smooth Shadings **76**
 - 3.5.2 Pantone, HKS, and custom Spot Colors **77**
 - 3.5.3 Color Management and ICC Profiles **80**
- 3.6 Interactive Elements 83**
 - 3.6.1 Links, Bookmarks, and Annotations **83**
 - 3.6.2 Form Fields and JavaScript **85**
- 3.7 Georeferenced PDF 89**
 - 3.7.1 Using Georeferenced PDF in Acrobat **89**
 - 3.7.2 Geographic and projected Coordinate Systems **89**
 - 3.7.3 Coordinate System Examples **90**
 - 3.7.4 Georeferenced PDF restrictions in Acrobat **91**

4 Unicode and Legacy Encodings 93

- 4.1 Important Unicode Concepts 93**
- 4.2 Single-Byte (8-Bit) Encodings 95**
- 4.3 Chinese, Japanese, and Korean Encodings 99**
- 4.4 String Handling in PDFlib 102**
 - 4.4.1 Content Strings, Hypertext Strings, and Name Strings **102**
 - 4.4.2 Strings in Unicode-aware Language Bindings **103**
 - 4.4.3 Strings in non-Unicode-aware Language Bindings **103**
- 4.5 Addressing Characters 107**
 - 4.5.1 Escape Sequences **107**
 - 4.5.2 Character References **108**

5 Font Handling 111

- 5.1 Font Formats 111**
 - 5.1.1 TrueType Fonts **111**
 - 5.1.2 OpenType Fonts **111**
 - 5.1.3 PostScript Type 1 Fonts **112**
 - 5.1.4 SING Fonts (Glyphlets) **112**
 - 5.1.5 Type 3 Fonts **113**

- 5.2 Unicode Characters and Glyphs 115**
 - 5.2.1 Glyph IDs 115
 - 5.2.2 Unicode Mappings for Glyphs 115
 - 5.2.3 Unicode Control Characters 116
- 5.3 The Text Processing Pipeline 118**
 - 5.3.1 Normalizing Input Strings to Unicode 118
 - 5.3.2 Converting Unicode Values to Glyph IDs 119
 - 5.3.3 Transforming Glyph IDs 120
- 5.4 Loading Fonts 121**
 - 5.4.1 Selecting an Encoding for Text Fonts 121
 - 5.4.2 Selecting an Encoding for symbolic Fonts 123
 - 5.4.3 Example: Selecting a Glyph from the Wingdings Symbol Font 124
 - 5.4.4 Searching for Fonts 126
 - 5.4.5 Host Fonts on Windows and Mac OS X 131
 - 5.4.6 Fallback Fonts 133
- 5.5 Font Embedding and Subsetting 137**
 - 5.5.1 Font Embedding 137
 - 5.5.2 Font Subsetting 138
- 5.6 Querying Font Information 140**
 - 5.6.1 Font-independent Encoding, Unicode, and Glyph Name Queries 140
 - 5.6.2 Font-specific Encoding, Unicode, and Glyph Name Queries 141
 - 5.6.3 Querying Codepage Coverage and Fallback Fonts 142

6 Text Output 145

- 6.1 Text Output Methods 145**
- 6.2 Font Metrics and Text Variations 147**
 - 6.2.1 Font and Glyph Metrics 147
 - 6.2.2 Kerning 148
 - 6.2.3 Text Variations 149
- 6.3 OpenType Layout Features 152**
 - 6.3.1 Supported OpenType Layout Features 152
 - 6.3.2 OpenType Layout Features with Textlines and Textflows 154
- 6.4 Complex Script Output 158**
 - 6.4.1 Complex Scripts 158
 - 6.4.2 Script and Language 160
 - 6.4.3 Complex Script Shaping 162
 - 6.4.4 Bidirectional Formatting 162
 - 6.4.5 Arabic Text Formatting 164
- 6.5 Chinese, Japanese, and Korean Text Output 166**
 - 6.5.1 Standard CJK Fonts 166
 - 6.5.2 Custom CJK Fonts 168
 - 6.5.3 EUDC and SING Fonts for Gaiji Characters 169
 - 6.5.4 OpenType Layout Features for advanced CJK Text Output 170

7 Importing Images and PDF Pages 173

7.1 Importing Raster Images 173

- 7.1.1 Basic Image Handling 173
- 7.1.2 Supported Image File Formats 175
- 7.1.3 Clipping Paths 178
- 7.1.4 Image Masks and Transparency 178
- 7.1.5 Colorizing Images 180

7.2 Importing PDF Pages with PDI 182

- 7.2.1 PDI Features and Applications 182
- 7.2.2 Using PDI Functions with PDFlib 182
- 7.2.3 Acceptable PDF Documents 184

7.3 Placing Images and imported PDF Pages 186

- 7.3.1 Simple Object Placement 186
- 7.3.2 Placing an Object in a Box 186
- 7.3.3 Orientating an Object 188
- 7.3.4 Rotating an Object 190
- 7.3.5 Adjusting the Page Size 191
- 7.3.6 Querying Information about placed Images and PDF Pages 192

8 Text and Table Formatting 193

8.1 Placing and Fitting Textlines 193

- 8.1.1 Simple Textline Placement 193
- 8.1.2 Positioning Text in a Box 194
- 8.1.3 Fitting Text into a Box 195
- 8.1.4 Aligning Text at a Character 197
- 8.1.5 Placing a Stamp 198
- 8.1.6 Using Leaders 198
- 8.1.7 Text on a Path 199

8.2 Multi-Line Textflows 201

- 8.2.1 Placing Textflows in the Fitbox 202
- 8.2.2 Paragraph Formatting Options 204
- 8.2.3 Inline Option Lists and Macros 204
- 8.2.4 Tab Stops 207
- 8.2.5 Numbered Lists and Paragraph Spacing 208
- 8.2.6 Control Characters and Character Mapping 209
- 8.2.7 Hyphenation 212
- 8.2.8 Controlling the standard Linebreak Algorithm 212
- 8.2.9 Advanced script-specific Line Breaking 216
- 8.2.10 Wrapping Text around Paths and Images 217

8.3 Table Formatting 221

- 8.3.1 Placing a Simple Table 222
- 8.3.2 Contents of a Table Cell 225
- 8.3.3 Table and Column Widths 227
- 8.3.4 Mixed Table Contents 228
- 8.3.5 Table Instances 231
- 8.3.6 Table Formatting Algorithm 234

- 8.4 Matchboxes 237**
 - 8.4.1 Decorating a Textline 237
 - 8.4.2 Using Matchboxes in a Textflow 238
 - 8.4.3 Matchboxes and Images 239

9 The pCOS Interface 241

10 PDF Versions and Standards 243

- 10.1 Acrobat and PDF Versions 243**
- 10.2 ISO 32 000 246**
- 10.3 PDF/X for Print Production 247**
 - 10.3.1 The PDF/X Family of Standards 247
 - 10.3.2 Generating PDF/X-conforming Output 248
 - 10.3.3 Output Intent and Standard Output Conditions 251
 - 10.3.4 Importing PDF/X Documents with PDI 252
- 10.4 PDF/A for Archiving 254**
 - 10.4.1 The PDF/A Standards 254
 - 10.4.2 Generating PDF/A-conforming Output 255
 - 10.4.3 Importing PDF/A Documents with PDI 258
 - 10.4.4 Color Strategies for creating PDF/A 259
 - 10.4.5 XMP Document Metadata for PDF/A 260
- 10.5 Tagged PDF 262**
 - 10.5.1 Generating Tagged PDF with PDFlib 262
 - 10.5.2 Creating Tagged PDF with direct Text Output and Textflows 264
 - 10.5.3 Activating Items for complex Layouts 265
 - 10.5.4 Using Tagged PDF in Acrobat 268

11 PPS and the PDFlib Block Plugin 271

- 11.1 Installing the PDFlib Block Plugin 271**
- 11.2 Overview of the Block Concept 273**
 - 11.2.1 Separation of Document Design and Program Code 273
 - 11.2.2 Block Properties 273
 - 11.2.3 Why not use PDF Form Fields? 274
- 11.3 Editing Blocks with the Block Plugin 276**
 - 11.3.1 Creating Blocks 276
 - 11.3.2 Editing Block Properties 280
 - 11.3.3 Copying Blocks between Pages and Documents 281
 - 11.3.4 Converting PDF Form Fields to PDFlib Blocks 282
 - 11.3.5 Customizing the Block Plugin User Interface with XML 285
- 11.4 Previewing Blocks in Acrobat 286**
- 11.5 Filling Blocks with PPS 290**
- 11.6 Block Properties 294**
 - 11.6.1 Administrative Properties 294
 - 11.6.2 Rectangle Properties 295
 - 11.6.3 Appearance Properties 296

11.6.4	Text Preparation Properties	298
11.6.5	Text Formatting Properties	299
11.6.6	Object Fitting Properties	302
11.6.7	Properties for default Contents	305
11.6.8	Custom Properties	305
11.7	Querying Block Names and Properties with pCOS	306
11.8	PDFlib Block Specification	308
11.8.1	PDF Object Structure for PDFlib Blocks	308
11.8.2	Block Dictionary Keys	310
11.8.3	Generating PDFlib Blocks with pdfmarks	311

A Revision History 313

Index 315

o Applying the PDFlib License Key

Restrictions of the evaluation version. 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 display a *www.pdfli**l**.com* demo stamp across all generated pages, and the integrated pCOS interface is limited to small documents (up to 10 pages and 1 MB file size). Unlicensed binaries must not be used for production purposes, but only for evaluating the product. Using any PDFlib GmbH product for production purposes requires a valid license.

Companies which are interested in PDFlib licensing and wish to get rid of the evaluation restrictions during the evaluation phase or for prototype demos can submit their company and project details with a brief explanation to *sales@pdfli**l**.com*, and apply for a temporary license key (we reserve the right to refuse evaluation key requests, e.g. for anonymous requests).

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.

Once you purchased a license key you must apply it in order to get rid of the demo stamp. Several methods are supported for setting the license key; they are detailed below.

Cookbook A full code sample can be found in the Cookbook topic *general/license_key*.

Windows installer. If you are working with the Windows installer you can enter the license key when you install the product. The installer will add the license key to the registry (see below).

Applying a license key with an API call at runtime. Add a line to your script or program which sets the license key at runtime. The *license* parameter must be set immediately after instantiating the PDFlib object (i.e., after *PDF_new()* or equivalent call). The exact syntax depends on your programming language:

- In C and Python:

```
PDF_set_option(p, "license=...your license key...")
```

- In C++, Java, and Ruby:

```
p.set_option("license=...your license key...")
```

- In Objective-C:

```
[pdflil set_option: @"license=...your license key..."];
```

- In Perl and PHP:

```
p->set_option("license=...your license key...")
```

- In RPG:

```
c                                callp      PDF_set_option(p:%ucs2('license=...your license key...'))
```

- In Tcl:

```
PDF_set_option $p, "license=...your license key..."
```

Working with a license file. As an alternative to supplying the license key with a runtime call, you can enter the license key in a text file according to the following format (you can use the license file template *licensekeys.txt* which is contained in all PDFlib distributions). Lines beginning with a '#' character contain comments and will be ignored; the second line contains version information for the license file itself:

```
# Licensing information for PDFlib GmbH products
PDFlib license file 1.0
PDFlib      8.0.5    ...your license key...
```

The license file may contain license keys for multiple PDFlib GmbH products on separate lines. It may also contain license keys for multiple platforms so that the same license file can be shared among platforms. License files can be configured in the following ways:

- A file called *licensekeys.txt* will be searched in all default locations (see »Default file search paths«, page 11).
- You can specify the *licensefile* parameter with the *set_option()* API function:

```
p.set_option("licensefile={/path/to/licensekeys.txt}");
```

- You can set an environment (shell) variable which points to a license file. On Windows use the system control panel and choose *System, Advanced, Environment Variables*; on Unix apply a command similar to the following:

```
export PDFLIBLICENSEFILE=/path/to/licensekeys.txt
```

- On IBM i5/iSeries the license file can be specified as follows (this command can be specified in the startup program *QSTRUP* and will work for all PDFlib GmbH products):

```
ADDENVVAR ENVVAR(PDFLIBLICENSEFILE) VALUE(<... path ...>) LEVEL(*SYS)
```

License keys in the registry. On Windows you can also enter the name of the license file in the following registry key:

```
HKLM\SOFTWARE\PDFlib\PDFLIBLICENSEFILE
```

As another alternative you can enter the license key directly in one of the following registry keys:

```
HKLM\SOFTWARE\PDFlib\PDFlib8\license
HKLM\SOFTWARE\PDFlib\PDFlib8\8.0.5\license
```

The MSI installer will write the license key provided at install time in the last of these entries.

Note Be careful when manually accessing the registry on 64-bit Windows systems: as usual, 64-bit PDFlib binaries will work with the 64-bit view of the Windows registry, while 32-bit PDFlib binaries running on a 64-bit system will work with the 32-bit view of the registry. If you must add registry keys for a 32-bit product manually, make sure to use the 32-bit version of the regedit tool. It can be invoked as follows from the Start, Run... dialog:

```
%systemroot%\syswow64\regedit
```

Default file search paths. On Unix, Linux, Mac OS X and i5/iSeries systems some directories will be searched for files by default even without specifying any path and directory names. Before searching and reading the UPR file (which may contain additional search paths), the following directories will be searched:

```
<rootpath>/PDFlib/PDFlib/8.0/resource/cmap
<rootpath>/PDFlib/PDFlib/8.0/resource/codelist
<rootpath>/PDFlib/PDFlib/8.0/resource/glyphlst
<rootpath>/PDFlib/PDFlib/8.0/resource/fonts
<rootpath>/PDFlib/PDFlib/8.0/resource/icc
<rootpath>/PDFlib/PDFlib/8.0
<rootpath>/PDFlib/PDFlib
<rootpath>/PDFlib
```

On Unix, Linux, and Mac OS X *<rootpath>* will first be replaced with */usr/local* and then with the HOME directory. On i5/iSeries *<rootpath>* is empty.

Default file names for license and resource files. By default, the following file names will be searched for in the default search path directories:

licensekeys.txt	(license file)
pdflib.upr	(resource file)

This feature can be used to work with a license file without setting any environment variable or runtime option.

Multi-system license files on i5/iSeries and zSeries. License keys for i5/iSeries and zSeries are system-specific and therefore cannot be shared among multiple systems. In order to facilitate resource sharing and work with a single license file which can be shared by multiple systems, the following license file format can be used to hold multiple system-specific keys in a single file:

```
PDFlib license file 2.0
# Licensing information for PDFlib GmbH products
PDFlib      8.0.5    ...your license key...    ...serial number of machine 1...
PDFlib      8.0.5    ...your license key...    ...serial number of machine 2...
```

Note the changed version number in the first line and the presence of multiple license keys, followed by the corresponding eight-digit hexadecimal serial number (on i5/iSeries) or four-digit hexadecimal CPU ID (on zSeries).

Working with license files on i5/iSeries. On i5/iSeries systems the license file must be encoded in ASCII (see *ascii* parameter). The following command sets the *PDFLIBLICENSEFILE* environment variable to point to a suitable license file:

```
ADDENVVAR ENVVAR(PDFLIBLICENSEFILE) VALUE('/PDFLIB/8.0.5/licensefile.txt')
LEVEL(*SYS)
```

Updates and Upgrades. If you purchased an update (change from an older version of a product to a newer version of the same product) or upgrade (change from PDFlib to PDFlib+PDI or PPS, or from PDFlib+PDI to PPS) you must apply the new license key that you received for your update or upgrade. The old license key for the previous product

must no longer be used. Note that license keys will work for all maintenance releases of a particular product version; as far as licensing is concerned, all versions 8.o.x are treated the same.

Evaluating features which are not yet licensed. You can fully evaluate all features 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, you can replace it with the dummy license string »o« (digit zero) to enable functionality of a higher product class for evaluation. 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
Franziska-Bilek-Weg 9, 80339 München, Germany
www.pdflib.com
phone +49 • 89 • 452 33 84-0
fax +49 • 89 • 452 33 84-99
Licensing contact: sales@pdflib.com
Support for PDFlib licensees: support@pdflib.com

1 Introduction

1.1 Roadmap to Documentation and Samples

We provide the material listed below to assist you in using PDFlib products successfully.

Note On Windows Vista and Windows 7 the mini samples and starter samples will be installed in the »Program Files« directory by default. Due to the Windows protection scheme the PDF output files created by these samples will only be visible under »compatibility files«. Recommended workaround: copy the folder with the samples to a user directory.

Mini samples for all language bindings. The *mini samples* (hello, image, pdfclock, etc.) are available in all packages and for all language bindings. They provide minimalistic sample code for text output, images, and vector graphics. The mini samples are mainly useful for testing your PDFlib installation, and for getting a very quick overview of writing PDFlib applications.

Starter samples for all language bindings. The *starter samples* are contained in all packages and are available for a variety of language bindings. They provide a useful generic starting point for important topics, and cover simple text and image output, Text-flow and table formatting, PDF/A and PDF/X creation and many other topics. The starter samples demonstrate basic techniques for achieving a particular goal with PDFlib products. It is strongly recommended to take a look at the starter samples.

PDFlib Tutorial. The *PDFlib Tutorial* (this manual), which is contained in all packages as a single PDF document, explains important programming concepts in more detail, including small pieces of sample code. If you start extending your code beyond the starter samples you should read up on relevant topics in the PDFlib Tutorial.

Note Most examples in this PDFlib Tutorial are provided in the Java language (except for the language-specific samples in Chapter 2, »PDFlib Language Bindings«, page 27, and a few C-specific samples which are marked as such). Although syntax details vary with each language, the basic concepts of PDFlib programming are the same for all language bindings.

PDFlib API Reference. The *PDFlib API Reference*, which is contained in all packages as a single PDF document, contains a concise description of all functions, parameters, and options which together comprise the PDFlib application programming interface (API). The PDFlib API Reference is the definitive source for looking up parameter details, supported options, input conditions, and other programming rules which must be observed. Note that some other reference documents are incomplete, e.g. the Javadoc API listing for PDFlib and the PDFlib function listing on *php.net*. Make sure to always use the full PDFlib API Reference when working with PDFlib.

pCOS Path Reference. The pCOS interface can be used to query a variety of properties from PDF documents. pCOS is included in PDFlib+PDI and PPS. The pCOS Path Reference contains a description of the path syntax used to address individual objects within a PDF document in order to retrieve the corresponding values.

PDFlib Cookbook. The *PDFlib Cookbook* is a collection of PDFlib coding fragments for solving specific problems. Most Cookbook examples are written in the Java language, but can easily be adjusted to other programming languages since the PDFlib API is almost identical for all supported language bindings. The PDFlib Cookbook is maintained as a growing list of sample programs. It is available at the following URL:

www.pdflib.com/pdflib-cookbook/

pCOS Cookbook. The *pCOS Cookbook* is a collection of code fragments for the pCOS interface which is contained in PDFlib+PDI and PPS. The pCOS interface can be used to query a variety of properties from PDF documents. It is available at the following URL:

www.pdflib.com/pcos-cookbook/

TET Cookbook. PDFlib TET (Text Extraction Toolkit) is a separate product for extracting text and images from PDF documents. It can be combined with PDFlib+PDI to process PDF documents based on their contents. The *TET Cookbook* is a collection of code fragments for TET. It contains a group of samples which demonstrate the combination of TET and PDFlib+PDI, e.g. add Web links or bookmarks based on the text on the page, highlight search terms, split documents based on text, create a table of contents, etc. The TET Cookbook is available at the following URL:

www.pdflib.com/tet-cookbook/

1.2 PDFlib Programming

What is PDFlib? PDFlib is a development component which allows you to generate files in Adobe's Portable Document Format (PDF). PDFlib acts as a backend to your own programs. While the application programmer is responsible for retrieving the data to be processed, PDFlib takes over the task of generating the PDF output which graphically represents the data. PDFlib frees you from the internal details of PDF, and offers various methods which help you formatting the output. The distribution packages contain different products in a single binary:

- ▶ PDFlib contains all functions required to create PDF output containing text, vector graphics and images plus hypertext elements. PDFlib offers powerful formatting features for placing single- or multi-line text, images, and creating tables.
- ▶ PDFlib+PDI includes all PDFlib functions, plus the PDF Import Library (PDI) for including pages from existing PDF documents in the generated output, and the pCOS interface for querying arbitrary PDF objects from an imported document (e.g. list all fonts on page, query metadata, and many more).
- ▶ 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 i5/iSeries and zSeries. PDFlib 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 zSeries)
- ▶ Java, including J2EE Servlets and JSP
- ▶ .NET for use with C#, VB.NET, ASP.NET, and other environments
- ▶ Objective C (Mac OS X, iOS)
- ▶ PHP
- ▶ Perl
- ▶ Python
- ▶ REALbasic
- ▶ RPG (IBM i5/iSeries)
- ▶ Ruby, including Ruby on Rails
- ▶ Tcl

What can I use PDFlib for? PDFlib's primary target is dynamic PDF creation within your own software or on a Web server. Similar to HTML pages dynamically generated on a 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.
- ▶ 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.

1.3 What's new in PDFlib/PDFlib+PDI/PPS 8?

The following list discusses the most important new or improved features in PDFlib/PDFlib+PDI/PPS 8 and Block Plugin 4. There are many more new features; see the PDFlib API Reference for details.

1.3.1 PDF Features for Acrobat 9

PDFlib supports various PDF features according to Acrobat 9 (technically: PDF 1.7 Adobe extension level 3).

External graphical content. Pages in PDF documents can contain references to pages in another PDF file, so-called Reference XObjects. The original file contains only a placeholder, e.g. a low-resolution version of an image or simply a note which mentions that the actual page contents are missing. Using this technique repeated content (e.g. for transactional printing) does not have to be transferred again and again. Reference XObjects are a crucial component of PDF/X-5g and PDF/X-5pg.

Layer variants. Layer variants (also called layer configurations) can be considered groups of layers. The grouping makes layers safe for production because the user can no longer inadvertently activate or deactivate the wrong set of layers (e.g. enable a particular language layer but forget to activate the image layer which is common to all languages). For this reason layer variants are the basis for using layers in the PDF/X-4 and PDF/X-5 standards.

PDF Portfolios. PDF Portfolios group PDF and other documents in a single entity which can conveniently be used with Acrobat 9. If no hierarchical folders are used for organizing the file attachments, the resulting PDF collections can be used with Acrobat 8 as well. PDFlib also supports predefined and custom metadata fields to facilitate the organization of file attachments within a PDF Portfolio. New actions can be used to create bookmarks which directly jump to a page in an embedded document.

Georeferenced PDF. Georeferenced PDF contains geographic reference information for the whole page or individual maps on the page. Acrobat 9 and above offer various features for interacting with Georeferenced PDF. PDFlib can be used to assign geospatial reference data to images and partial or full pages.

AES-256 encryption and Unicode passwords. PDFlib supports AES-256 encryption for improved security. AES-256 encryption has been introduced with Acrobat 9 and also allows Unicode passwords.

Import Acrobat 9 documents. PDFlib+PDI and PPS can import and process Acrobat 9 documents. The pCOS interface can also analyze Acrobat 9 documents.

1.3.2 Font Handling and Text Output

Complex script shaping and bidirectional text formatting. Simple scripts, e.g. Latin, are scripts in which characters are placed one after the other from left to right. Complex scripts require additional processing for shaping the text (selecting appropriate glyph form depending on context), reordering characters, or formatting text from right to left. PDFlib supports complex script output for a variety of scripts including the Arabic, Hebrew, Devanagari, and Thai scripts.

Fallback fonts. Fallback fonts are a powerful mechanism for dealing with a variety of font and encoding-related restrictions. You can mix and match fonts, pull missing glyphs from another font, extend encodings, etc. Fallback fonts can adjust the size of individual glyphs automatically to account for design differences in the combined fonts.

OpenType layout features. OpenType layout features add intelligence to an OpenType font in the form of additional tables in the font file. These tables describe advanced typographic features such as ligatures, small capitals, swash characters, etc. They also support advanced CJK text output with halfwidth, fullwidth, and proportional glyphs, alternate forms, and many others.

Retain fonts across documents. Fonts and associated data can be kept in memory after the generated document is finished. This improves performance since the font doesn't have to be parsed again for the next document, while still doing document-specific processing such as font subsetting.

SING fonts for CJK Gaiji characters. The Japanese term Gaiji refers to custom characters (e.g. family or place names) which are in common use, but are not included in any encoding standard. Adobe's SING font architecture (*glyphlets*) solves the Gaiji problem for CJK text. PDFlib supports SING fonts as well as the related Microsoft concept of EUDC fonts (end-user defined fonts). Using the *fallback font* feature SING and EUDC fonts can be merged into an existing font.

Redesigned font engine. PDFlib's font engine has been redesigned and streamlined, resulting in a variety of Unicode and encoding-related advantages as well as general performance improvements and reduced memory requirements. Due to the redesign some restrictions could be eliminated and the functionality of existing features extended. For example, it is now possible to address more than 256 glyphs in Type 1 or Type 3 fonts, address swash characters by glyph name, etc.

Wrap text around image clipping paths. The Textflow formatting engine wraps text around arbitrary paths and can also use the clipping path of an imported TIFF or JPEG image. This way multi-line text can be wrapped around an image.

Text on a path. Text can be placed on arbitrary vector paths consisting of an arbitrary mixture of straight line segments, curves, and arcs. The paths can be constructed programmatically. Alternatively, the clipping paths from TIFF or JPEG images can be extracted and used as text paths.

1.3.3 PDFlib Block Plugin and PDFlib Personalization Server (PPS)

The PDFlib Block Plugin is used to prepare PDF documents for Block filling (personalization) with the PDFlib Personalization Server (PPS).

Preview PPS Block processing in Acrobat. The Plugin can generate previews of the Block filling process with PPS directly in Acrobat. The immediate preview allows designers to quickly review the results of PPS-filling their Block documents before submitting them to the server for processing. The preview PDF contains bookmarks, layers, and annotations with possible error messages as debugging and development aids. The preview feature speeds up development cycles and can also be used as an interactive test framework for trying PDFlib features.

Clone PDF/A or PDF/X status of the Block container. When generating Block previews based on PDF/A or PDF/X documents, the Block Plugin can clone all relevant aspects of the standard, e.g. standard identification, output intent, and metadata. If a Block filling operation in PDF/A or PDF/X cloning mode would violate the selected standard (e.g. because a default image uses RGB color space although the document does not contain a suitable output intent) an error message will be displayed. This way users can catch potential standard violations very early in the workflow.

Redesigned user interface and snap-to-grid. The user interface of the PDFlib Block Plugin has been restructured to facilitate access to the large number of existing and new Block properties.

The new snap-to-grid feature is useful for quickly laying out Blocks according to a design raster.

Additional Block properties. More Block properties have been added to the Block Plugin and PPS, e.g. for specifying transparency of text, image, or PDF contents placed in a Block.

Leverage PDFlib 8 features with Blocks. Relevant new features of PDFlib 8 such as text output for complex scripts and OpenType layout features can be activated directly with Block properties. For example, Blocks can be filled with Arabic or Hindi text.

1.3.4 Other important Features

Reusable path objects. Path objects can be constructed independently from any page and used one or more times for stroking, filling, or clipping. Path objects can also be used as wrapping shapes (format text into irregularly shaped areas) or to place text on the path.

PDF/X-4 and PDF/X-5. PDFlib creates output according to the PDF/X-4 and PDF/X-5 standards for the graphic arts industry. Compared to the earlier PDF/X-1 and PDF/X-3 standards these are based on a newer PDF version and allow transparency and layers. PDF/X-4p and PDF/X-5pg support externally referenced ICC profiles as output intents. PDF/X-5g and PDF/X-5pg support the use of external graphical content.

Alpha channel in TIFF and PNG images. PDFlib honors image transparency (alpha channel) when importing TIFF and PNG images. Alpha channels can be used to create smooth transitions and to blend an image with the background.

JBIG2-compressed images. JBIG2 is a highly effective image compression format for black and white images. PDFlib imports single- and multi-page JBIG2 images and maintains the advantages of their compression in the generated PDF output.

Compressed object streams and cross-reference streams. PDFlib creates compressed object streams and cross-reference streams. These methods reduce the overall file size of the generated PDF documents and help PDF documents to jump over the previous 10 GB limit which holds for PDFs with conventional cross-reference tables. While 10 GB may seem an awful lot of data, an increasing number of applications in transaction printing are approach this limit. We expect to see more and more scenarios where PDFlib users want to create PDF documents in this range.

Builtin PANTONE® Goe™ color libraries. PDFlib supports the new PANTONE® Goe™ color libraries with 2058 new colors for coated and uncoated paper as well as a new color naming scheme. The Goe™ color libraries have been introduced by Pantone, Inc. in 2008.

Improvements in existing functions. The list below mentions some of the most important improvements of existing features in PDFlib 8:

- ▶ query image details with *PDF_info_image()*
- ▶ PPS and Block Plugin: additional Block properties which make new PDFlib features accessible via PDFlib Blocks
- ▶ Unicode filenames on Unix systems
- ▶ Table formatter: place path objects, annotations, and form fields in table cells
- ▶ Textflow: additional formatting control options, advanced language-specific line-breaking
- ▶ shadow text
- ▶ retain XMP metadata in imported images
- ▶ many improvements in *PDF_info_font()*
- ▶ additional options for creating annotations
- ▶ Configurable string data type for the C++ binding, e.g. *wstring* for Unicode support

1.4 Features in PDFlib

Table 1.1 lists major PDFlib features for generating PDF. New and improved features in PDFlib 8 are marked.

Table 1.1 Feature list for PDFlib

topic	features
PDF output	Generate PDF documents on disk file or directly in memory (for Web servers)
	High-volume output and arbitrary PDF file size (even beyond 10 GB)
	Suspend/resume and insert page features to create pages out of order
PDF flavors	PDF 1.3 – PDF 1.7ext8 ¹ (Acrobat 4–X) including ISO 32000-1 (=PDF 1.7)
	Linearized (web-optimized) PDF for byteserving over the Web
	Tagged PDF for accessibility and reflow
ISO standards	Marked Content for adding application-specific data or alternate text without Tagging ¹
	ISO 15930: PDF/X for the graphic arts industry ¹
	ISO 19005: PDF/A for archiving
	ISO 32000: standardized version of PDF 1.7 ¹
Graphics	Common vector graphics primitives: lines, curves, arcs, ellipses ¹ , rectangles, etc.
	Smooth shadings (color blends), pattern fills and strokes
	Transparency (opacity) and blend modes
	External graphical content (Reference XObjects) for variable data printing ¹
	Reusable path objects and clipping paths imported from images ¹
Layers	Optional page content which can selectively be displayed
	Annotations and form fields can be placed on layers
	Layers can be locked, automatically activated depending on zoom factor, etc.
	Layer variants ¹ (production-safe groups of layers) for PDF/X-4 and PDF/X-5
Fonts	TrueType (TTF and TTC) and PostScript Type 1 fonts (PFB and PFA, plus LWFN on the Mac)
	OpenType fonts with PostScript or TrueType outlines (TTF, OTF)
	Support for dozens of OpenType layout features for Western and CJK text output, e.g. ligatures, small caps, old-style numerals, swash characters, simplified/traditional forms, vertical alternates ¹
	Directly use fonts which are installed on the Windows or Mac system («host fonts»)
	Font embedding for all font types; subsetting for TrueType, OpenType, and Type 3 fonts
	User-defined (Type 3) fonts for bitmap fonts or custom logos
	EUDC and SING ¹ fonts (glyphlets) for CJK Gaiji characters
	Fallback fonts (pull missing glyphs from an auxiliary font) ¹
	Retain fonts across documents to increase performance ¹
	Font metrics for improved character spacing
Text output	Text output in different fonts; underlined, overlined, and strikeout text
	Glyphs in a font can be addressed by numerical value, Unicode value, or glyph name ¹
	Artificial bold, italic, and shadow ¹ text
	Create text on a path ¹
	Proportional widths for CJK fonts ¹
	Configurable replacement of missing glyphs

Table 1.1 Feature list for PDFlib

topic	features
Internationalization	Unicode strings for page content, interactive elements, and file names ¹ ; UTF-8, UTF-16, and UTF-32 formats
	Support for a variety of 8-bit and legacy multi-byte CJK encodings (e.g. Shift-JIS; Big5)
	Fetch code pages from the system (Windows, IBM i5/iSeries and zSeries)
	Standard and custom CJK fonts and CMaps for Chinese, Japanese, and Korean text
	Vertical writing mode for Chinese, Japanese, and Korean text
	Character shaping for complex scripts, e.g. Arabic, Thai, Devanagari ¹
	Bidirectional text formatting for right-to-left scripts, e.g. Arabic and Hebrew ¹
Images	Embed Unicode information in PDF for proper text extraction in Acrobat
	Embed BMP, GIF, PNG, TIFF, JBIG2 ¹ , JPEG, JPEG 2000 ¹ , and CCITT raster images
	Automatic detection of image file formats
	Query image information (pixel size, resolution, ICC profile, clipping path, etc.) ¹
	Interpret clipping paths in TIFF and JPEG images
Color	Interpret alpha channel (transparency) in TIFF and PNG images ¹
	Image masks (transparent images with a color applied), colorize images with a spot color
	Grayscale, RGB (numerical, hexadecimal strings, HTML color names), CMYK, CIE L*a*b* color
	Integrated support for PANTONE® colors (incl. PANTONE® Goe™) ¹ and HKS® colors
	User-defined spot colors
Color management	ICC-based color with ICC profiles; support for ICC 4 profiles ¹
	Rendering intent for text, graphics, and raster images
	Default gray, RGB, and CMYK color spaces to remap device-dependent colors
	ICC profiles as output intent for PDF/A and PDF/X
Archiving	PDF/A-1a and PDF/A-1b (ISO 19005-1)
	XMP extension schemas for PDF/A-1
Graphic arts	PDF/X-1a, PDF/X-3, PDF/X-4 ¹ , PDF/X-4p ¹ , PDF/X-5p ¹ , PDF/X-5pg ¹ (ISO 15930)
	Embedded or externally referenced ¹ output intent ICC profile
	External graphical content (referenced pages) for PDF/X-5p and PDF/X-5pg ¹
	Create OPI 1.3 and OPI 2.0 information for imported images
	Separation information (PlateColor)
Textflow Formatting	Settings for text knockout, overprinting etc.
	Format text into one or more rectangular or arbitrarily shaped areas with hyphenation (user-supplied hyphenation points required), font and color changes, justification methods, tabs, leaders, control commands; wrap text around images
	Advanced line-breaking with language-specific processing
	Flexible image placement and formatting
Table formatting	Wrap text around images or image clipping paths ¹
	Table formatter places rows and columns, and automatically calculates their sizes according to a variety of user preferences. Tables can be split across multiple pages.
	Table cells can hold single- or multi-line text, images, PDF pages, path objects, annotations, and form fields
	Table cells can be formatted with ruling and shading options
	Flexible stamping function
	Matchbox concept for referencing the coordinates of placed images or other objects

Table 1.1 Feature list for PDFlib

topic	features
Security	Encrypt PDF output with RC4 (40/128 bit) or AES encryption algorithms (128/256 ¹ bit)
	Unicode passwords ¹
	Specify permission settings (e.g. printing or copying not allowed)
Interactive elements	Create form fields with all field options and JavaScript
	Create barcode form fields
	Create actions for bookmarks, annotations, page open/close and other events
	Create bookmarks with a variety of options and controls
	Page transition effects, such as shades and mosaic
	Create all PDF annotation types, such as PDF links, launch links (other document types), Web links
	Named destinations for links, bookmarks, and document open action
	Create page labels (symbolic names for pages)
Multimedia	Embed 3D animations in PDF
Georeferenced PDF	Create PDF with geospatial reference information ¹
Tagged PDF	Create Tagged PDF and structure information for accessibility, page reflow, and improved content repurposing; links and other annotations can be integrated in the document structure
Metadata	Document information: common fields (Title, Subject, Author, Keywords) and user-defined fields
	Create XMP metadata from document info fields or from client-supplied XMP streams
	Process XMP image metadata in TIFF, JPEG, and JPEG 2000 images ¹
Programming	Language bindings for Cobol, COM, C, C++ ¹ , Objective C ¹ , Java, .NET, Perl, PHP, Python, REALbasic, RPG, Ruby, Tcl
	Virtual file system for supplying data in memory, e.g., images from a database

1. New or considerably improved in PDFlib 8

1.5 Additional Features in PDFlib+PDI

Table 1.2 lists features in PDFlib+PDI and PPS in addition to the basic PDF generation features in Table 1.1.

Table 1.2 Additional features in PDFlib+PDI

topic	features
PDF input (PDI)	Import pages from existing PDF documents
	Import all PDF versions up to PDF 1.7 extension level 3 (Acrobat 9) ¹
	Import documents which are encrypted with any of PDF's standard encryption algorithms (master password required) ¹
	Query information about imported pages ¹
	Clone page geometry of imported pages (e.g. BleedBox, TrimBox, CropBox) ¹
	Delete redundant objects (e.g. identical fonts) across multiple imported PDF documents
	Repair malformed input PDF documents ¹
	Copy PDF/A or PDF/X output intent from imported PDF documents
pCOS interface	pCOS interface for querying details about imported PDF documents ¹

1. New or considerably improved in PDFlib+PDI 8

1.6 Additional Features in PPS

Table 1.3 lists features which are only available in the PDFlib Personalization Server (PPS) (in addition to the basic PDF generation features in Table 1.1 and the PDF import features in Table 1.2).

Table 1.3 Additional features in the PDFlib Personalization Server (PPS)

topic	features
Variable Data Processing (PPS)	PDF personalization with PDFlib Blocks for text, image, and PDF data
PDFlib Block Plugin	PDFlib Block plugin for creating PDFlib Blocks interactively in Acrobat on Windows and Mac
	Redesigned user interface ¹
	Preview PPS Block filling in Acrobat ¹
	Snap-to-grid for interactively creating or editing Blocks in Acrobat ¹
	Clone PDF/X or PDF/A properties of the Block container ¹
	Convert PDF form fields to PDFlib Blocks for automated filling
	Textflow Blocks can be linked so that one Block holds the overflow text of a previous Block
	List of PANTONE® and HKS® spot color names integrated in the Block plugin ¹

1. New or considerably improved in PDFlib Personalization Server 8

1.7 Availability of Features in different Products

Table 1.4 details the availability of features in different products with the PDFlib family.

Table 1.4 Availability of features in different products

feature	API functions and options	PDFlib	PDFlib+PDI	PPS
basic PDF generation	all except those listed below	X	X	X
linearized (Web-optimized) PDF	linearize option in PDF_end_document()	X ¹	X	X
optimize PDF (only relevant for inefficient client code and non-optimized imported PDF documents)	optimize option in PDF_end_document()	X ¹	X	X
Referenced PDF, PDF/X-5g and PDF/X-5pg	reference option in PDF_begin_template_ext() and PDF_open_pdi_page()	X ¹	X	X
Parsing PDF documents for Portfolio creation	password option in PDF_add_portfolio_file()	X ¹	X	X
PDF import (PDI)	all PDI functions	–	X	X
Query information from PDF with pCOS	all pCOS functions	–	X	X
Variable data processing and personalization with Blocks	all PPS functions for Block filling	–	–	X
PDFlib Block plugin for Acrobat	interactively create PDFlib blocks for use with PPS	–	–	X

1. Not available in PDFlib source code packages since PDI is required internally for this feature

2 PDFlib Language Bindings

Note It is strongly recommended to take a look at the starter examples which are contained in all PDFlib packages. They provide a convenient starting point for your own application development, and cover many important aspects of PDFlib programming.

2.1 Data Types for Language Bindings

This manual documents the function/method prototypes for various language bindings. The main difference between language bindings is that in object-oriented language bindings the PDFlib methods do not have the *PDF_* prefix in the name, while in other language bindings the *PDF_* prefix is part of all function names. Also, the PDF context parameter must be supplied as the first argument to all functions in non-object oriented language bindings. In contrast, the object-oriented language bindings hide the PDF context in an object created by the language wrapper.

Table 2.1 details the use of the PDF document type and the string data type in all language bindings. See the *PDFlib Tutorial* for more details on text and string handling. The data types *integer*, *long*, and *double* are not mentioned since there is an obvious mapping of these types in all bindings.

Table 2.1 Data types in the language bindings

language binding	p parameter and PDF_ prefix?	string data type	binary data type
C	yes	const char * ¹	const char *
C++	no	std::wstring by default ²	const char *
Cobol	yes ³	STRING	STRING
Java	no	String	byte[]
Objective-C	no	NSString	NSData
Perl	no	string	string
PHP	no	string	string
Python	no	string	string
RPG	yes	Unicode string (use %ucs2)	data
Ruby	no	string	string
Tcl	yes	string	byte array

1. C language NULL string values and empty strings are considered equivalent.
2. The C++ API can be customized via instantiation of the std::basic_string template. For example, the API can be switched to std::string to achieve compatibility with older applications. Alternatively, user-defined data types can also be used as the basis of the string type used in the API (see Section 2.5, »C++ Binding«, page 33).
3. Cobol programs must use abbreviated names for the PDFlib functions.

2.2 Cobol Binding

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

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

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

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

Data types. The data types used in the PDFlib API Reference must be mapped to Cobol data types as in the following samples:

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.

2.3 COM Binding

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



2.4 C Binding

PDFlib is written in C with some C++ modules. 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 next section for details).

Note Applications which use the PDFlib binding for C must be linked with a C++ compiler since the PDFlib library includes some parts which are implemented in C++. Using a C linker may result in unresolved externals unless the application is explicitly linked against the required C++ support libraries.

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.

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

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

PDF_delete(p);
```

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 com-

prehend. Make sure to use the macros exactly as shown above, with no additional code between the *TRY* and *CATCH* clauses (except *PDF_CATCH()*).

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. 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 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.

Volatile variables. Special care must be taken regarding variables that are used in both the *PDF_TRY()* and the *PDF_CATCH()* blocks. Since the compiler doesn't know about the control transfer from one block to the other, it might produce inappropriate code (e.g., register variable optimizations) in this situation. Fortunately, there is a simple rule to avoid these problems:

Note Variables used in both the PDF_TRY() and PDF_CATCH() blocks should be declared volatile.

Using the *volatile* keyword signals to the compiler that it must not apply (potentially dangerous) optimizations to the variable.

Nesting try/catch blocks and rethrowing exceptions. *PDF_TRY()* blocks may be nested to an arbitrary depth. In the case of nested error handling, the inner catch block can activate the outer catch block by re-throwing the exception:

```
PDF_TRY(p)                                /* outer try block */
{
    /* ... */

    PDF_TRY(p)                             /* inner try block */
    {
        /* ... */
    }
    PDF_CATCH(p)                           /* inner catch block */
    {
        /* error cleanup */
        PDF_RETHROW(p);
    }
    /* ... */
}
PDF_CATCH(p)                             /* outer catch block */
{
    /* more error cleanup */
    PDF_delete(p);
}
```

The *PDF_RETHROW()* invocation in the inner error handler will transfer program execution to the first statement of the outer *PDF_CATCH()* block immediately.

Prematurely exiting a try block. If a *PDF_TRY()* block is left – e.g., by means of a return statement –, thus bypassing the invocation of the corresponding *PDF_CATCH()* macro, the *PDF_EXIT_TRY()* macro must be used to inform the exception machinery. No other library function must be called between this macro and the end of the try block:

```
PDF_TRY(p)
{
    /* ... */

    if (error_condition)
    {
        PDF_EXIT_TRY(p);
        return -1;
    }
}
PDF_CATCH(p)
{
    /* error cleanup */
    PDF_RETHROW(p);
}
```

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 memory 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.

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

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*. Since *pdflib.hpp* contains a template-based implementation no corresponding *.cpp* module is required. Using the C++ object wrapper replaces the *PDF_* prefix in all PDFlib function names with a more object-oriented approach.

Using PDFlib as a DLL loaded at runtime. Similar to the C language binding the C++ binding allows you to dynamically attach PDFlib to your application at runtime (see »Using PDFlib as a DLL loaded at runtime«, page 30). Dynamic loading can be enabled as follows when compiling the application module which includes *pdflib.hpp*:

```
#define PDFCPP_DL 1
```

In addition you must compile the auxiliary module *pdflibdl.c* and link your application against the resulting object file. Since the details of dynamic loading are hidden in the PDFlib object it does not affect the C++ API: all method calls look the same regardless of whether or not dynamic loading is enabled.

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

String handling in C++. PDFlib 8 introduces a new Unicode-capable C++ binding. The new template-based approach supports the following usage patterns with respect to string handling:

- ▶ Strings of the C++ standard library type *std::wstring* are used as basic string type. They can hold Unicode characters encoded as UTF-16 or UTF-32. This is the default behavior in PDFlib 8 and the recommended approach for new applications unless custom data types (see next item) offer a significant advantage over wstrings.
- ▶ Custom (user-defined) data types for string handling can be used as long as the custom data type is an instantiation of the *basic_string* class template and can be converted to and from Unicode via user-supplied converter methods. As an example a custom string type implementation for UTF-8 strings is included in the PDFlib distribution.
- ▶ Plain C++ strings can be used for compatibility with existing C++ applications which have been developed against PDFlib 7 or earlier versions. This compatibility variant is only meant for existing applications (see below for notes on source code compatibility).

The new interface assumes that all strings passed to and received from PDFlib methods are native wstrings. Depending on the size of the *wchar_t* data type, wstrings are assumed to contain Unicode strings encoded as UTF-16 (2-byte characters) or UTF-32 (4-byte characters). Literal strings in the source code must be prefixed with *L* to designate wide strings. Unicode characters in literals can be created with the *\u* and *\U* syntax. Although this syntax is part of standard ISO C++, some compilers don't support it. In this case literal Unicode characters must be created with hex characters.

*Note On EBCDIC-based systems the formatting of option list strings for the wstring-based interface requires additional conversions to avoid a mixture of EBCDIC and UTF-16 wstrings in option lists. Convenience code for this conversion and instructions are available in the auxiliary module *utf16num_ebcdic.hpp*.*

Adjusting applications to the new C++ binding. Existing C++ applications which have been developed against PDFlib 7 or earlier versions can be adjusted to PDFlib 8 as follows:

- ▶ Since the PDFlib C++ class now lives in the *pdflib* namespace the class name must be qualified. In order to avoid the *pdflib::PDFlib* construct client applications should add the following before using PDFlib methods:

```
using namespace pdflib;
```

- ▶ Switch the application's string handling to wstrings. This includes data from external sources. However, string literals in the source code (including option lists) must also be adjusted by prepending the *L* prefix, e.g.

```
const wstring imagefile = L"nesrin.jpg";  
image = p.load_image(L"auto", imagefile, L"");
```

- ▶ Suitable wstring-capable methods (*wcerr* etc.) must be used to process PDFlib error messages and exception strings (*get_errmsg()* method in the *PDFlib* and *PDFlib-Exception* classes).
- ▶ Remove PDFlib method calls which are required only for non-Unicode-capable languages, especially the following:

```
p.set_parameter("hypertextencoding", "host");
```

- ▶ The *pdflib.cpp* module is no longer required for the PDFlib C++ binding. Although the PDFlib distribution contains a dummy implementation of this module, it should be removed from the build process for PDFlib applications.

Full source code compatibility with legacy applications. The new C++ binding has been designed with application-level source code compatibility mind, but client applications must be recompiled. The following aids are available to achieve full source code compatibility for legacy applications:

- ▶ Disable the wstring-based interface as follows before including *pdflib.hpp*:

```
#define PDFCPP_PDFLIB_WSTRING 0
```

- ▶ Disable the PDFlib namespace as follows before including *pdflib.hpp*:

```
#define PDFCPP_USE_PDFLIB_NAMESPACE 0
```

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) {  
    wcerr << L"PDFlib exception occurred in hello sample: " << endl  
        << L "[" << ex.get_errnum() << L"] " << ex.get_apiname()
```

```
    << L": " << ex.get_errmsg() << endl;  
}
```

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

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

2.6 Java Binding

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

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

Taking into account PDFlib's stability and maturity, attaching the native PDFlib library to the Java VM doesn't impose any stability or security restrictions on your Java application, while at the same time offering the performance benefits of a native implementation.

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. 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*. 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.

Using PDFlib in J2EE application servers and Servlet containers. PDFlib is perfectly suited for server-side Java applications. The PDFlib distribution contains sample code and configuration for using PDFlib in J2EE environments. The following configuration issues must be observed:

- ▶ The directory where the server looks for native libraries varies among vendors. Common candidate locations are system directories, directories specific to the underly-

ing Java VM, and local server directories. Please check the documentation supplied by the server vendor.

- Application servers and Servlet containers often use a special class loader which may be restricted or uses a dedicated classpath. For some servers it is required to define a special classpath to make sure that the PDFlib package will be found.

More detailed notes on using PDFlib with specific Servlet engines and application servers can be found in additional documentation in the J2EE directory of the PDFlib distribution.

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.

Unicode and legacy encoding conversion. For the convenience of PDFlib users we list some useful string conversion methods here. Please refer to the Java documentation for more details. The following constructor creates a Unicode string from a byte array, using the platform's default encoding:

```
String(byte[] bytes)
```

The following constructor creates a Unicode string from a byte array, using the encoding supplied in the *enc* parameter (e.g. *SJIS*, *UTF8*, *UTF-16*):

```
String(byte[] bytes, String enc)
```

The following method of the String class converts a Unicode string to a string according to the encoding specified in the *enc* parameter:

```
byte[] getBytes(String enc)
```

Javadoc documentation for PDFlib. The PDFlib package contains Javadoc documentation for PDFlib. The Javadoc contains only abbreviated descriptions of all PDFlib API methods; please refer to the PDFlib API Reference for more details.

In order to configure Javadoc for PDFlib in Eclipse proceed as follows:

- ▶ In the Package Explorer right-click on the Java project and select *Javadoc Location*.
- ▶ Click on *Browse...* and select the path where the Javadoc (which is part of the PDFlib package) is located.

After these steps you can browse the Javadoc for PDFlib, e.g. with the *Java Browsing* perspective or via the *Help* menu.

Using PDFlib with Groovy. The PDFlib Java binding can also be used with the Groovy language. The API calls are identical to the Java calls; only the object instantiation is slightly different. A simple example for using PDFlib with Groovy is contained in the PDFlib distribution.

2.7 .NET Binding

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



2.8 Objective-C Binding

Although the C and C++ language bindings can be used with Objective-C¹, a genuine language binding for Objective-C is also available. The PDFlib framework is available in the following flavors:

- ▶ *PDFlib* for use on Mac OS X
- ▶ *PDFlib_ios* for use on iOS

Both frameworks contain language bindings for C, C++, and Objective-C.

Installing the PDFlib Edition for Objective-C on Mac OS X. In order to use PDFlib in your application you must copy *PDFlib.framework* or *PDFlib_ios.framework* to the directory */Library/Frameworks*. Installing the PDFlib framework in a different location is possible, but requires use of Apple's *install_name_tool* which is not described here. The *PDFlib_objc.h* header file with PDFlib method declarations must be imported in the application source code:

```
#import "PDFlib/PDFlib_objc.h"
```

or

```
#import "PDFlib_ios/PDFlib_objc.h"
```

Data types and parameter naming conventions. PDFlib expects the following Objective-C datatypes in its method interfaces: *NSString* (instead of *string* in C++), *NSInteger* (instead of *int*), *NSData* (instead of *const char **). For PDFlib method calls you must supply parameters according to the following conventions:

- ▶ The value of the first parameter is provided directly after the method name, separated by a colon character.
- ▶ For each subsequent parameter the parameter's name and its value (again separated from each other by a colon character) must be provided. The parameter names can be found in the PDFlib API Reference or in *PDFlib_objc.h*.

For example, the following line in the PDFlib API Reference:

```
void begin_page_ext(double width, double height, String optlist)
```

corresponds to the following Objective-C method:

```
-(void) begin_page_ext: (double) width height: (double) height optlist: (NSString *) optlist;
```

This means your application must make a call similar to the following:

```
[pdflib begin_page_ext:595.0 height:842.0 optlist:@""];
```

XCode Code Sense for code completion can be used with the PDFlib framework.

Error handling in Objective-C. The Objective-C binding installs a special error handler which translates PDFlib errors to native Objective-C exceptions. In case of a runtime problem PDFlib throws a native Objective-C exception of the class *PDFlibException*. These exceptions can be handled with the usual *try/catch* mechanism:

1. See developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html


```

@try {
    ...some PDFlib instructions...
}
@catch (PDFlibException *ex) {
    NSString * errorMessage =
        [NSString stringWithFormat:@"PDFlib error %d in '%@': %@",
        [ex get_errnum], [ex get_apiname], [ex get_errmsg]];
    UIAlertView *alert = [[UIAlertView alloc] init];
    [alert setMessageText: errorMessage];
    [alert runModal];
    [alert release];
}
@catch (NSEException *ex) {
    UIAlertView *alert = [[UIAlertView alloc] init];
    [alert setMessageText: [ex reason]];
    [alert runModal];
    [alert release];
}
@finally {
    [pdflib release];
}

```

In addition to the *get_errmsg* method you can also use the *reason* field of the exception object to retrieve the error message.

2.9 Perl Binding

The PDFlib wrapper for Perl¹ consists of a C wrapper file and two Perl package modules, one for providing a Perl equivalent for each PDFlib API function and another one for the PDFlib object. 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.

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 modules *pdflib_pl.pm* and *PDFlib/PDFlib.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 *pdflib_pl.so* (on Mac OS X: *pdflib_pl.bundle*), *pdflib_pl.pm* and *PDFlib/PDFlib.pm* in the current directory or the directory printed by the following Perl command:

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

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

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

Windows. PDFlib supports the ActiveState port of Perl 5 to Windows, also known as ActivePerl.² The DLL *pdflib_pl.dll* and the modules *pdflib_pl.pm* and *PDFlib/PDFlib.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
```

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...  
};  
if ($?) {  
    die("$0: PDFlib Exception occurred:\n$?");  
}
```

¹ See www.perl.com

² See www.activestate.com

More than one way of String handling. Depending on the requirements of your application you can work with UTF-8, UTF-16, or legacy encodings. The following code snippets demonstrate all three variants. All examples create the same Japanese output, but accept the string input in different formats.

The first example works with Unicode UTF-8 and uses the *Unicode::String* module which is part of most modern Perl distributions, and available on CPAN). Since Perl works with UTF-8 internally no explicit UTF-8 conversion is required:

```
use Unicode::String qw(utf8 utf16 uhex);
...
$p->set_parameter("textformat", "utf8");
$font = $p->load_font("Arial Unicode MS", "unicode", "");
$p->setfont($font, 24.0);
$p->set_text_pos(50, 700);
$p->show(uhex("U+65E5 U+672C U+8A9E"));
```

The second example works with Unicode UTF-16 and little-endian byte order:

```
$p->set_parameter("textformat", "utf16le");
$font = $p->load_font("Arial Unicode MS", "unicode", "");
$p->setfont($font, 24.0);
$p->set_text_pos(50, 700);
$p->show("\xE5\x65\x2C\x67\x9E\x8A");
```

The third example works with Shift-JIS. Except on Windows systems it requires access to the *goms-RKSJ-H* CMap for string conversion:

```
$p->set_parameter("SearchPath", "../../../resource/cmap");
$font = $p->load_font("Arial Unicode MS", "cp932", "");
$p->setfont($font, 24.0);
$p->set_text_pos(50, 700);
$p->show("\x93\xFA\x96\x7B\x8C\xEA");
```

Unicode and legacy encoding conversion. For the convenience of PDFlib users we list some useful string conversion methods here. Please refer to the Perl documentation for more details. The following constructor creates a UTF-16 Unicode string from a byte array:

```
$logos="\x{039b}\x{03bf}\x{03b3}\x{03bf}\x{03c3}\x{0020}" ;
```

The following constructor creates a Unicode string from the Unicode character name:

```
$delta = "\N{GREEK CAPITAL LETTER DELTA}";
```

The *Encode* module supports many encodings and has interfaces for converting between those encodings:

```
use Encode 'decode';
$data = decode("iso-8859-3", $data); # convert from legacy to UTF-8
```

2.10 PHP Binding

Note Detailed information about the various flavors and options for using PDFlib with PHP¹, including the question of whether or not to use a loadable PDFlib module for PHP, can be found in the *PDFlib-in-PHP-HowTo.pdf* document which is contained in the distribution packages and also available on the PDFlib Web site.

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

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

```
extension=libpdf_php.so      ; for Unix and 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 additionally 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 Binary Version* (and the PDFlib version number) you are using the supported new PDFlib wrapper. The unsupported old wrapper will display *PDFlib GmbH Version* instead.

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

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

Modified error return for PDFlib functions in PHP. Since PHP uses the convention of returning the value 0 (FALSE) when an error occurs within a function, all PDFlib functions have been adjusted to return 0 instead of -1 in case of an error. This difference is noted in the function descriptions in the PDFlib API Reference. However, take care when reading the code fragment examples in Section 3, »Creating PDF Documents«, page 53, since they 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 the use of PDFlib's *SearchPath* facility is strongly recommended (see Section 3.1.3, »Resource Configuration and File Search«, page 56).

Exception handling in PHP. Since PHP supports structured exception handling, PDFlib exceptions will be propagated as PHP exceptions. PDFlib will throw an exception of the

¹. See www.php.net

class *PDFlibException*, which is derived from PHP's standard Exception class. 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;  
}
```

Unicode and legacy encoding conversion. The *iconv* module can be used for string conversions. Please refer to the PHP documentation for more details.

PDFlib development with Eclipse and Zend Studio. The PHP Development Tools (PDT)¹ support PHP development with Eclipse and Zend Studio. PDT can be configured to support context-sensitive help with the steps outlined below.

Add PDFlib to the Eclipse preferences so that it will be known to all PHP projects:

- ▶ Select *Window, Preferences, PHP, PHP Libraries, New...* to launch a wizard.
- ▶ In *User library name* enter *PDFlib*, click *Add External folder...* and select the folder *bind\php\Eclipse PDT*.

In an existing or new PHP project you can add a reference to the PDFlib library as follows:

- ▶ In the PHP Explorer right-click on the PHP project and select *Include Path, Configure Include Path...*
- ▶ Go to the *Libraries* tab, click *Add Library...*, and select *User Library, PDFlib*.

After these steps you can explore the list of PDFlib methods under the *PHP Include Path/PDFlib/PDFlib* node in the PHP Explorer view. When writing new PHP code Eclipse will assist with code completion and context-sensitive help for all PDFlib methods.

1. See www.eclipse.org/pdt

2.11 Python Binding

Installing the PDFlib Python Edition. The Python¹ extension mechanism works by loading shared libraries at runtime. For the PDFlib binding to work, the Python interpreter must have access to the PDFlib library for Python which will be searched in the directories listed in the PYTHONPATH environment variable. The name of the Python wrapper depends on the platform:

- ▶ Unix and Mac OS X: *pdflib_py.so*
- ▶ Windows: *pdflib_py.pyd*

In addition to the PDFlib library the following files must be available in the same directory where the library sits:

- ▶ *PDFlib/PDFlib.py*
- ▶ *PDFlib/__init__.py*

Error Handling in Python. PDFlib installs an error handler which translates PDFlib exceptions to native Python exceptions. The Python exceptions can be dealt with by the usual *try/catch* technique:

```
try:
    ...some PDFlib instructions...
except PDFlibException:
    print("PDFlib exception occurred:\n[%d] %s: %s" %
          ((p.get_errnum()), p.get_apiname(), p.get_errmsg()))

finally:
    p.delete()
```

¹. See www.python.org

2.12 REALbasic Binding

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



2.13 RPG Binding

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

Unicode string handling. Since all functions provided by PDLlib use Unicode strings with variable length as parameters, you have to use the `%UCS2` builtin function to convert a single-byte string to a Unicode string. All strings returned by PDLlib functions are Unicode strings with variable length. Use the `%CHAR` builtin function to convert these Unicode strings to single-byte strings.

Note The `%CHAR` and `%UCS2` functions use the current job's `CCSID` to convert strings from and to Unicode. The examples provided with PDLlib are based on `CCSID 37` (US EBCDIC). Some special characters in option lists (e.g. `{ [] }`) may not be translated correctly if you run the examples under other codepages.

Since all strings are passed as variable length strings you must not pass the *length* parameters in various functions which expect explicit string lengths (the length of a variable length string is stored in the first two bytes of the string).

Compiling and binding RPG Programs for PDLlib. Using PDLlib functions from RPG requires the compiled PDLlib and PDLlib_RPG service programs. To include the PDLlib definitions at compile time you have to specify the name of the */copy* member in the *D* specs of your ILE-RPG program:

```
d/copy QRPGLSRC,PDLlib
```

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

```
d/copy PDFsrcLib/QRPGLSRC,PDLlib
```

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

```
CRTBNDDIR BNDDIR(PDLlib/PDLlib) TEXT('PDLlib Binding Directory')
```

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

```
ADDBNDDIRE BNDDIR(PDLlib/PDLlib) OBJ((PDLlib/PDLlib *SRVPGM))
ADDBNDDIRE BNDDIR(PDLlib/PDLlib) OBJ((PDLlib/PDLlib_RPG *SRVPGM))
```

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

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


Error Handling in RPG. PDFlib clients written in ILE-RPG can use the *monitor/on-error/endmon* error handling mechanism that ILE-RPG provides. Another way to monitor for exceptions is to use the **PSSR* global error handling subroutine in ILE-RPG. If an exception occurs, the job log shows the error number, the function that failed and the reason for the exception. PDFlib sends an escape message to the calling program.

```
c      eval      p=PDF_new
*
c      monitor
*
c      eval      doc=PDF_begin_document(p:%ucs2('/tmp/my.pdf'):docoptlist)
:
:
*      Error Handling
c      on-error
*      Do something with this error
*      don't forget to free the PDFlib object
c      callp      PDF_delete(p)
c      endmon
```

2.14 Ruby Binding

Installing the PDFlib Ruby edition. The Ruby¹ extension mechanism works by loading a shared library at runtime. For the PDFlib binding to work, the Ruby interpreter must have access to the PDFlib extension library for Ruby. This library (on Windows and Unix: *PDFlib.so*; on Mac OS X: *PDFlib.bundle*) will usually be installed in the *site_ruby* branch of the local ruby installation directory, i.e. in a directory with a name similar to the following:

```
/usr/local/lib/ruby/site_ruby/<version>/
```

However, Ruby will search other directories for extensions as well. In order to retrieve a list of these directories you can use the following ruby call:

```
ruby -e "puts $:"
```

This list will usually include the current directory, so for testing purposes you can simply place the PDFlib extension library and the scripts in the same directory.

Error Handling in Ruby. The Ruby binding installs an error handler which translates PDFlib exceptions to native Ruby exceptions. The Ruby exceptions can be dealt with by the usual *rescue* technique:

```
begin
    ...some PDFlib instructions...
rescue PDFlibException => pe
    print "PDFlib exception occurred in hello sample:\n"
    print "[" + pe.get_errnum.to_s + "]" + pe.get_apiname + ": " + pe.get_errmsg + "\n"
end
```

Ruby on Rails. Ruby on Rails² is an open-source framework which facilitates Web development with Ruby. The PDFlib extension for Ruby can be used with Ruby on Rails. Follow these steps to run the PDFlib examples for Ruby on Rails:

- ▶ Install Ruby and Ruby on Rails.
- ▶ Set up a new controller from the command line:

```
$ rails new pdflibdemo
$ cd pdflibdemo
$ cp <PDFlib dir>/bind/ruby/<version>/PDFlib.so vendor/ # use .so/.dll/.bundle
$ rails generate controller home demo
$ rm public/index.html
```

- ▶ Edit *config/routes.rb*:

```
...
# remember to delete public/index.html
root :to => "home#demo"
```

- ▶ Edit *app/controllers/home_controller.rb* as follows and insert PDFlib code for creating PDF contents. Keep in mind that the PDF output must be generated in memory, i.e.

1. See www.ruby-lang.org/en

2. See www.rubyonrails.org

an empty file name must be supplied to *begin_document()*. As a starting point you can use the code in the *hello-rails.rb* sample:

```
class HomeController < ApplicationController
  def demo
    require "PDFlib"
    begin
      p = PDFlib.new
      ...
      ...PDFlib application code, see hello-rails.rb...
      ...
      send_data p.get_buffer(), :filename => "hello.pdf",
      :type => "application/pdf", :disposition => "inline"
      rescue PDFlibException => pe
      # error handling
    end
  end
end
```

- In order to test your installation start the WEBrick server with the command

```
$ rails s
```

and point your browser to *http://0.0.0.0:3000*. The generated PDF document will be displayed in the browser.

Local PDFlib installation. If you want to use PDFlib only with Ruby on Rails, but cannot install it globally for general use with Ruby, you can install PDFlib locally in the *vendors* directory within the Rails tree. This is particularly useful if you do not have permission to install Ruby extensions for general use, but want to work with PDFlib in Rails nevertheless.

2.15 Tcl Binding

Installing the PDFlib Tcl edition. The Tcl¹ 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/tcl8.4/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\tcl8.4\pdflib
```

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
}
```

¹ See www.tcl.tk

3 Creating PDF Documents

3.1 General PDFlib Programming Aspects

Cookbook Code samples regarding general programming issues can be found in the general category of the *PDFlib Cookbook*.

3.1.1 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 (i.e. special error return codes such as -1) for function calls which may often fail, 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 PDFlib API Reference. This error code must be checked by the application developer for all functions which are documented to return -1 on error.

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), or supplying wrong options.

When PDFlib detects such a situation, an exception will be thrown instead of passing a special error return value to the caller. It is important to understand that the generated PDF document cannot be finished when 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 will contain the following information:

- ▶ A unique error number;
- ▶ The name of the PDFlib API function which caused the exception;
- ▶ A descriptive text containing details of the problem.

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 the program flow or supplying different data. 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. In this case it may be desirable to retrieve an error message which describes the problem in more detail. In this situation the functions *PDF_get_errnum()*, *PDF_get_errmsg()*, and *PDF_get_apiname()* can be called immediately after a failed function call, i.e., a function call which returned a -1 (in PHP: 0) error value.

Error policies. When PDFlib detects an error condition, it will react according to one of several strategies which can be configured with the *errorpolicy* parameter. All functions which can return error codes also support an *errorpolicy* option. The following error policies are supported:

- ▶ *errorpolicy=legacy*: this deprecated setting ensures behavior which is compatible to earlier versions of PDFlib, where exceptions and error return values are controlled by parameters and options such as *fontwarning*, *imagewarning*, etc. This is only recommended for applications which require source code compatibility with PDFlib 6. It should not be used for new applications. The *legacy* setting is the default error policy.
- ▶ *errorpolicy=return*: when an error condition is detected, the respective function will return with a -1 (in PHP: 0) error value regardless of any warning parameters or options. The application developer must check the return value to identify problems, and must react on the problem in whatever way is appropriate for the application. This is the recommended approach since it allows a unified approach to error handling.
- ▶ *errorpolicy=exception*: an exception will be thrown when an error condition is detected. However, the output document will be unusable after an exception. This can be used for lazy programming without any error conditionals at the expense of sacrificing the output document even for problems which may be fixable by the application.

The following code fragments demonstrate different strategies with respect to exception handling. The examples try to load a font which may or may not be available.

If *errorpolicy=return* the return value must be checked for an error. If it indicates failure, the reason of the failure can be queried in order to properly deal with the situation:

```
font = p.load_font("MyFontName", "unicode", "errorpolicy=return");
if (font == -1)
{
    /* font handle is invalid; find out what happened. */
    errmsg = p.get_errmsg();
    /* Try a different font or give up */
    ...
}
/* font handle is valid; continue */
```

If *errorpolicy=exception* the document must be abandoned if an error occurs:

```
font = p.load_font("MyFontName", "unicode", "errorpolicy=exception");
/* Unless an exception was thrown the font handle is valid;
 * if an exception occurred, the PDF output cannot be continued
 */
```

Cookbook A full code sample can be found in the *Cookbook* topic `general/error_handling`.

Warnings. Some problem conditions can be detected by PDFlib internally, but do not justify interrupting the program flow by throwing an exception. While earlier versions of PDFlib supported the concept of non-fatal exceptions which can be disabled, PDFlib 7 never throws an exception for non-fatal conditions. Instead, a description of the condition will be logged (if logging is enabled). Logging can be enabled as follows:

```
p.set_parameter("logging", "filename=private.log");
```

We recommend the following approach with respect to warnings:

- ▶ Enable warning logging in the development phase, and carefully study any warning messages in the log file. They may point to potential problems in your code or data, and you should try to understand or eliminate the reason for those warnings.
- ▶ Disable warning logging in the production phase, and re-enable it only in case of problems.

3.1.2 The PDFlib Virtual File System (PVF)

Cookbook A full code sample can be found in the Cookbook topic `general/starter_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 sup-

plied), 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()*.

Creating PDF output in a virtual file. In addition to supplying user data to PDFlib, PVF can also hold the PDF document data generated by PDFlib. This can be achieved by supplying the *createpvf* option to *PDF_begin_document()*. The PVF file name can later be supplied to other PDFlib API functions. This is useful, for example, when generating PDF documents for inclusion in a PDF Portfolio. It is not possible to directly retrieve the PVF data created by PDFlib; use the active or passive in-core PDF generation interface to fetch PDF data from memory (see Section 3.1.4, »Generating PDF Documents in Memory«, page 61).

3.1.3 Resource Configuration and File Search

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 3.1. Other resource categories will be ignored. The values are treated as name strings; they can be encoded in ASCII or UTF-8 (with BOM). Unicode values may be useful for localized font names with the *HostFont* resource.

Table 3.1 Resource categories supported in PDFlib

category	format	explanation
SearchPath	value	Relative or absolute path name of directories containing data files
CMap	key=value	CMap file for CJK encoding
FontAFM	key=value	PostScript font metrics file in AFM format
FontPFM	key=value	PostScript font metrics file in PFM format
FontOutline	key=value	PostScript, TrueType or OpenType font outline file
Encoding	key=value	text file containing an 8-bit encoding or code page table
HostFont	key=value	Name of a font installed on the system
ICCProfile	key=value	name of an ICC color profile
StandardOutputIntent	key=value	name of a standard output condition for PDF/X (in addition to those built into PDFlib, see PDFlib API Reference for a complete list)

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 1023 characters.
- ▶ A backslash character '\ ' at the end of a line cancels the line end. This may be used to extend lines.
- ▶ A percent '%' character introduces a comment until the end of the line. Percent characters which are part of the line data (i.e. which do not start a comment) must be protected with a preceding backslash character.
- ▶ Backslash characters in front of a backslash which protects the line end and backslash characters which protect a percent character must be duplicated if they are part of the line data.
- ▶ An isolated period character '.' serves as a section terminator.
- ▶ All entries are case-sensitive.
- ▶ Whitespace is ignored everywhere except in resource names and file names.
- ▶ Resource names and values must not contain any equal character '='.
- ▶ If a resource is defined more than once, the last definition will overwrite earlier definitions.

UPR files consist of the following components:

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

PS-Resources-1.0
- ▶ An optional 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.
If this optional section is not present, a single period character must be present nevertheless.
- ▶ 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 search and the *SearchPath* resource category. PDFlib reads a variety of data items, such as raster images, font outline and metrics information, encoding definitions, PDF documents, and ICC color profiles from disk files. In addition to relative or absolute path names you can also use file names without any path specification. The *SearchPath* resource category can be used to specify a list of path names for directories containing the required data files. When PDFlib must open a file it will first use the file name exactly as supplied and try to open the file. If this attempt fails, PDFlib will try to open the file in the directories specified in the *SearchPath* resource category one after another until it succeeds. *SearchPath* entries can be accumulated, and will be searched in reverse order (paths set at a later point in time will be searched before earlier ones). This feature can be used to free PDFlib applications from platform-specific file system schemes. You can set search path entries as follows:

```
p.set_parameter("SearchPath", "/path/to/dir1");
p.set_parameter("SearchPath", "/path/to/dir2");
```

In order to disable the search you can use a fully specified path name in the PDFlib functions. Note the following platform-specific features of the *SearchPath* resource category:

- ▶ On Windows PDFlib will initialize the *SearchPath* with entries from the registry. The following registry entries may contain a list of path names separated by a semicolon ';' character. They will be searched in the order provided below:

```
HKLM\SOFTWARE\PDFlib\PDFlib8\8.0.5\SearchPath
HKLM\SOFTWARE\PDFlib\PDFlib8\SearchPath
HKLM\SOFTWARE\PDFlib\SearchPath
```

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

```
/PDFlib/PDFlib/8.0/resource/icc
/PDFlib/PDFlib/8.0/resource/fonts
/PDFlib/PDFlib/8.0/resource/cmap
/PDFlib/PDFlib/8.0
/PDFlib/PDFlib
/PDFlib
```

The last of these entries is especially useful for storing a license file for multiple products.

- ▶ On OpenVMS logical names can be supplied as *SearchPath*.

Default file search paths. On Unix, Linux, Mac OS X and i5/iSeries systems some directories will be searched for files by default even without specifying any path and directory names. Before searching and reading the UPR file (which may contain additional search paths), the following directories will be searched:

```
<rootpath>/PDFlib/PDFlib/8.0/resource/cmap
<rootpath>/PDFlib/PDFlib/8.0/resource/codelist
<rootpath>/PDFlib/PDFlib/8.0/resource/glyphlst
<rootpath>/PDFlib/PDFlib/8.0/resource/fonts
<rootpath>/PDFlib/PDFlib/8.0/resource/icc
<rootpath>/PDFlib/PDFlib/8.0
```

```
<rootpath>/PDFlib/PDFlib
<rootpath>/PDFlib
```

On Unix, Linux, and Mac OS X *<roothpath>* will first be replaced with */usr/local* and then with the HOME directory. On i5/iSeries *<roothpath>* is empty.

Default file names for license and resource files. By default, the following file names are searched for in the default search path directories:

licensekeys.txt	(license file; on MVS: license)
pdflib.upr	(resource file)

This feature can be used to work with a license file without setting any environment variable or runtime option.

Sample UPR file. The following listing gives an example of a UPR configuration file:

```
PS-Resources-1.0
.
SearchPath
/usr/local/lib/fonts
C:/psfonts/pfm
C:/psfonts
/users/kurt/my_images
.
FontAFM
Code-128=Code_128.afm
.
FontPFM
Corporate-Bold=corpb____.pfm
Mistral=c:/psfonts/pfm/mist____.pfm
.
FontOutline
Code-128=Code_128.pfa
ArialMT=Arial.ttf
.
HostFont
Wingdings=Wingdings
.
Encoding
myencoding=myencoding.enc
.
ICCPProfile
highspeedprinter=cmykhighspeed.icc
.
```

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

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

- ▶ On Unix systems, Mac OS X systems, and i5/iSeries some directories will be searched by default for license and resource files even without specifying any path and direc-

tory names. Before searching and reading the UPR file, the following directories will be searched (in this order):

```
<rootpath>/PDFlib/PDFlib/8.0/resource/icc  
<rootpath>/PDFlib/PDFlib/8.0/resource/fonts  
<rootpath>/PDFlib/PDFlib/8.0/resource/cmap  
<rootpath>/PDFlib/PDFlib/8.0  
<rootpath>/PDFlib/PDFlib  
<rootpath>/PDFlib
```

On Unix systems and Mac OS X *<rootpath>* will first be replaced with */usr/local* and then with the HOME directory. On i5/iSeries *<rootpath>* is empty. This feature can be used to work with a license file, UPR file, or resources without setting any environment variables or runtime parameters.

- ▶ 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 i5/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 following registry entries which will be searched in the order provided below:

```
HKLM\Software\PDFlib\PDFlib8\8.0.5\resourcefile  
HKLM\Software\PDFlib\PDFlib8\resourcefile  
HKLM\Software\PDFlib\resourcefile
```

The values of these entries will be taken as the name of the resource file to be used. If this file cannot be read an exception will be thrown.

Be careful when manually accessing the registry on 64-bit Windows systems: as usual, 64-bit PDFlib binaries will work with the 64-bit view of the Windows registry, while 32-bit PDFlib binaries running on a 64-bit system will work with the 32-bit view of the registry. If you must add registry keys for a 32-bit product manually, make sure to use the 32-bit version of the regedit tool. It can be invoked as follows from the Start, Run... dialog:

```
%systemroot%\syswow64\regedit
```

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

```
p.set_parameter("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:

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

Note Font configuration is discussed in more detail in Section 5.4.4, »Searching for Fonts«, page 126.

Querying resource values. In addition to setting resource entries you can query values using `PDF_get_parameter()`. Specify the category name as key and the index in the list as modifier. For example, the following call:

```
s = p.get_parameter("SearchPath", n);
```

will retrieve the *n*-th entry in the SearchPath list. If *n* is larger than the number of available entries for the requested category an empty string will be returned. The returned string is valid until the next call to any API function.

3.1.4 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 a single 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.

You can use the `createpvf` option to create PDF data in memory and subsequently pass it to PDFlib without writing a disk file (see »Creating PDF output in a virtual file«, page 56).

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()`:

```
p.begin_document("", "");
...create document...
p.end_document("");

buf = p.get_buffer();
... use the PDF data contained in the buffer ...
p.delete();
```

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()`.

3.1.5 Large PDF Documents

Although most users won't see any need for PDF documents in the range of Gigabytes, some enterprise applications must create or process documents containing a large number of, say, invoices or statements. While PDFlib itself does not impose any limits on the size of the generated documents, there are several restrictions mandated by the PDF Reference and some PDF standards:

- ▶ 2 GB file size limit: The PDF/A-1 standard limits the file size to 2 GB. If a document gets larger than this limit, PDFlib will throw an exception in PDF/A-1, PDF/X-4 and PDF/X-5 mode. Otherwise documents beyond 2 GB can be created.
- ▶ 10 GB file size limit: PDF documents have traditionally been limited internally by the cross-reference table to 10 decimal digits and therefore 10^{10} -1 bytes, which equates to roughly 9.3 GB. However, using compressed object streams this limit can be exceeded. If you plan to create output documents beyond 10 GB you must set the *objectstreams={other}* option in `PDF_begin_document()`. This requires PDF 1.5 or above. While compressed object streams will reduce the overall file size anyway, the compressed cross-reference streams which are part of the *objectstreams* implementation are no longer subject to the 10-decimal-digits limit, and therefore allow creation of PDF documents beyond 10 GB.
- ▶ Number of objects: while the object count in a document is not limited by PDF in general, the PDF/A-1, PDF/X-4 and PDF/X-5 standards limits the number of indirect objects in a document to 8,388,607. If a document requires objects beyond this limit, PDFlib will throw an exception in PDF/A-1, PDF/X-4 and PDF/X-5 mode. Otherwise documents with more objects can be created. While the number of generated objects can not directly be retrieved from PDFlib, the client application can save some objects by re-using image handles and loading images outside of *page* scope. The number of objects in PDF depends on the complexity of the page contents, number of interactive elements, etc. Since typical high-volume documents with simple contents require ca. 4-10 objects per page on average, documents with ca. 1-2 million pages can be created without exceeding the object limit.

3.1.6 Using PDFlib on EBCDIC-based Platforms

The operators and structure elements in the PDF file format are based on ASCII which doesn't work well with EBCDIC-based platforms such as IBM iSeries and zSeries with the z/OS, USS or MVS operating systems (but not zLinux which is based on ASCII). However, a special mainframe version of PDFlib is available 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 coordinate system 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:

```
p.scale(28.3465, 28.3465);
```

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

As a related feature, the *userunit* option in *PDF_begin/end_page_ext()* (PDF 1.6) can be specified to supply a scaling factor for the whole page. Note that user units will only affect final page display in Acrobat, but not any coordinate scaling in PDFlib.

Cookbook A full code sample can be found in the *Cookbook* topic `general/metric_topdown_coordinates`.

Coordinates for interactive elements. PDF always expects coordinates for interactive 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*:

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

Since PDF supports only link and field 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 interactive 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.

You can visualize page coordinates in Acrobat as follows:

- ▶ To display cursor coordinates use the following:
Acrobat X: *View, Show/Hide, Cursor Coordinates*
Acrobat 9: *View, Cursor Coordinates*
Acrobat 8: *View, Navigation Tabs, Info*
- ▶ The coordinates will be displayed in the unit which is currently selected in Acrobat. To change the display units in Acrobat 8/9/X proceed as follows: go to *Edit, Preferences, [General...], Units & Guides* and choose one of Points, Inches, Millimeters, Picas, Centimeters.

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. See »Using top-down coordinates«, page 66, for details on selecting a coordinate system which aligns with Acrobat's coordinate display.

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

- ▶ The *Page Scaling*: option in Acrobat's print dialog has a setting different from *None*, 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 can easily be achieved with the *rotate* option of *PDF_fit_textline()*, *PDF_fit_textflow()*, *PDF_fit_image()*, and *PDF_fit_pdi_page()*. Rotating such objects by multiples of 90 degrees inside the respective fitbox can be accomplished with the *orientate* option of these functions. The following example generates some text at an angle of 45° degrees:

```
p.fit_textline("Rotated text", 50.0, 700.0, "rotate=45");
```

Cookbook A full code sample can be found in the *Cookbook* topic `text_output/rotated_text`.

Rotation for vector graphics can be achieved by applying the general coordinate transformation functions *PDF_translate()* and *PDF_rotate()*. The following example creates a rotated rectangle with lower left corner at (200, 100). It translates the coordinate origin to the desired corner of the rectangle, rotates the coordinate system, and places the rectangle at (0, 0). The *save/restore* nesting makes it easy to continue placing objects in the original coordinate system after the rotated rectangle is done:

```
p.save();
  p.translate(200, 100);          /* move origin to corner of rectangle*/
  p.rotate(45.0);                /* rotate coordinates */
  p.rect(0.0, 0.0, 75.0, 25.0);  /* draw rotated rectangle */
  p.stroke();
p.restore();
```

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 (text easily appears bottom-up), 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. The *topdown* feature has been designed to make it quite natural for PDFlib users to work in a top-down coordinate system. 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 for a page can be activated with the *topdown* option of *PDF_begin_page_ext()*:

```
p.begin_page_ext(595.0, 842.0, "topdown");
```

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.

Cookbook A full code sample can be found in the *Cookbook* topic `general/metric_topdown_coordinates`.

3.2.2 Page Size

Cookbook A full code sample can be found in the *Cookbook* topic `pagination/page_sizes`.

Standard page formats. Absolute values and symbolic page size names may be used for the *width* and *height* options in *PDF_begin/end_page_ext()*. The latter are called `<format>.width` and `<format>.height`, where `<format>` is one of the standard page formats (in lowercase, e.g. `a4.width`).

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. The page size limits for Acrobat are shown in Table 3.2. In PDF 1.6 and above the *userunit* option in *PDF_begin/end_page_ext()* can be used to specify a global scaling factor for the page.

Table 3.2 Minimum and maximum page size of Acrobat

PDF viewer	minimum page size	maximum page size
Acrobat 4 and above	1/24" = 3 pt = 0.106 cm	200" = 14400 pt = 508 cm
Acrobat 7 and above with the userunit option	3 user units	14 400 user units The maximum value 75 000 for userunit allows page sizes up to 14 400 * 75 000 = 1 080 000 000 points = 381 km

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):

- ▶ **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 */
p.begin_page_ext(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.

3.2.3 Direct Paths and Path Objects

A path is a shape made of an arbitrary number of straight lines, rectangles, circles, Bézier curves, or elliptical segments. A path may consist of several disconnected sections, called subpaths. There are several operations which can be applied to a path:

- ▶ **Stroking** draws a line along the path, using client-supplied parameters (e.g., color, line width) for drawing.
- ▶ **Filling** paints the entire region enclosed by the path, using client-supplied parameters for filling.
- ▶ **Clipping** reduces the imageable area for subsequent drawing operations by replacing the current clipping area (which is unlimited 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 useful.

Direct Paths. Using the path functions `PDF_moveto()`, `PDF_lineto()`, `PDF_rect()` etc. you can construct a direct path which will immediately be written to the current page or another content stream (e.g. a template or Type 3 glyph description). Immediately after constructing the path it must be processed with one of `PDF_stroke()`, `PDF_fill()`, `PDF_clip()` and related functions. These functions will consume and delete the path. The only way to use a path multiply is with `PDF_save()` and `PDF_restore()`.

It is an error to construct a direct path without applying any of the above operations to it. PDFlib's scoping system ensures that clients obey to this restriction. If you want to set any appearance properties (e.g. color, line width) of a path you must do so before starting any drawing operations. These rules can 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:

```
p.setcolor("stroke", "rgb", 1, 0, 0, 0);
p.moveto(100, 100);
p.lineto(200, 100);
p.stroke();
```

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.

Cookbook A full code sample can be found in the *Cookbook* topic `graphics/starter_graphics`.

Path objects. Path objects are more convenient and powerful alternative to direct paths. Path objects encapsulate all drawing operations for constructing the path. Path objects can be created with `PDF_add_path_point()` or extracted from an image file which includes an image clipping path (see below). `PDF_add_path_point()` supports several convenience options to facilitate path construction. Once a path object has been created it can be used for different purposes:

- ▶ The path object can be used on the page description with `PDF_draw_path()`, i.e. filled, stroked, or used as a clipping path.
- ▶ Path objects can be used as wrap shapes for Textflow: the text will be formatted so that it wraps inside or outside of an arbitrary shape (see Section 8.2.10, »Wrapping Text around Paths and Images«, page 217).
- ▶ Text can also be placed on a path, i.e. the characters follow the lines and curves of the path (see Section 8.1.7, »Text on a Path«, page 199).
- ▶ Path objects can be placed in table cells.

Unlike direct paths, path objects can be used multiply until they are explicitly destroyed with `PDF_delete_path()`. Information about a path can be retrieved with `PDF_info_path()`. The following code fragment creates a simple path shape with a circle, strokes it at two different locations on the page, and finally deletes it:

```
path = p.add_path_point(-1, 0, 100, "move", "");
path = p.add_path_point(path, 200, 100, "control", "");
path = p.add_path_point(path, 0, 100, "circular", "");

p.draw_path(path, 0, 0, "stroke");
p.draw_path(path, 400, 500, "stroke");
p.delete_path(path);
```

Instead of creating a path object with individual drawing operations you can extract the clipping path from an imported image:

```
image = p.load_image("auto", "image.tif", "clippingpathname={path 1}");

/* create a path object from the image's clipping path */
path = (int) p.info_image(image, "clippingpath", "");
if (path == -1)
    throw new Exception("Error: clipping path not found!");

p.draw_path(path, 0, 0, "stroke");
```

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 7.3, »Placing Images and imported PDF Pages«, page 186). The general template idiom in PDFlib looks as follows:

```
/* define the template */
template = p.begin_template_ext(template_width, template_height, "");
...place marks on the template using text, vector, and image functions...
p.end_template_ext(0, 0);
...
p.begin_page(page_width, page_height);
/* use the template */
p.fit_image(template, 0.0, 0.0, "");
...more page marking operations...
p.end_page();
...
p.close_image(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:

- ▶ *PDF_load_image()*: images must be loaded outside of a template definition, but can be used within a template.

- ▶ *PDF_begin_pattern()*, *PDF_shading_pattern()*, *PDF_shading()*: patterns and shadings must be defined outside the template, but can be used within the template.
- ▶ *PDF_begin_item()* and *tag* option of various functions: structure elements cannot be created within a template.
- ▶ All interactive functions, since these must always be defined on the page where they should appear in the document, and cannot be generated as part of a template.

Cookbook A full code sample can be found in the *Cookbook* topic `general/repeated_contents`.

3.2.5 Referenced Pages from an external PDF Document

Cookbook A full code sample can be found in the *Cookbook* topic `pdfx/starter_pdfx5g`.

PDF documents can contain references to pages in external documents: the (scaled or rotated) reference page is not part of the document, it will be displayed and printed just like other page content. This can be used to reference re-used graphical contents (e.g. logos or cover pages) without including the corresponding PDF data. PDFlib supports strong references, i.e. references where the referenced page is identified via internal metadata. If the referenced page is not available or does not match the expected metadata, a proxy image will be displayed instead of the referenced page.

Using referenced pages in Acrobat. Although referenced pages (the technical term is *Reference XObjects*) have already been specified in PDF 1.4 (i.e. the file format of Acrobat 5), they require Acrobat 9 or above for proper display and printing. Referenced pages are a crucial component of PDF/X-5g and PDF/X-5pg. The generated (new) document is called the container document; the external PDF document with the referenced page is called the target file.

In order to use referenced pages with Acrobat 9 or X it is important to properly configure Acrobat as follows:

- ▶ *Edit, Preferences, General...*, *Page Display, Show reference XObject targets*: set to *Always* (the setting *Only PDF-X/5 compliant ones* doesn't work due to a bug in Acrobat).
- ▶ *Edit, Preferences, General...*, *Page Display, Location of referenced files*: enter the name of the directory where the target files live.
- ▶ *Edit, Preferences, General...*, *Security (Enhanced), Privileged Locations, Add Folder Path*: add the name of the directory where the container documents live. This must be done regardless of the *Enable Enhanced Security* setting.

The target page, whose file name and page number are specified inside the container PDF, will be displayed instead of the proxy if all of the following conditions are true:

- ▶ The container document is trusted according to Acrobat's configuration;
- ▶ The target file can be found in the specified directory;
- ▶ The target file does not require any password and can be opened without errors.
- ▶ The page number of the referenced page specified in the container document exists in the target file.
- ▶ PDF/X-5 only: the ID and certain XMP metadata entries in the target must match the corresponding entries in the container document.

If one or more of these conditions are violated, the proxy will silently be displayed instead of the target page. Acrobat will not issue any error message.

Proxy for the target page. PDFlib can use one of the following objects as placeholder (proxy) for the reference page:

- ▶ Another imported PDF page (e.g. a simplified version of the target). A PDF page which serves as a proxy for an external target must have the same page geometry as the target page.
- ▶ A template, e.g. a simple geometric shape such as a crossed-out rectangle. Templates will be adjusted to the size and aspect ratio of the target page.

The following code segment constructs a proxy template with a reference to an external page:

```
proxy = p.begin_template_ext(0, 0,  
    "reference={filename=target.pdf pagenumber=1 strongref=true}");  
if (proxy == -1)  
{  
    /* Error */  
}  
...construct template contents...  
p.end_template_ext(0, 0);
```

The proxy can be placed on the page as usual. It will carry the reference to the external target.

3.3 Encrypted PDF

3.3.1 PDF Security Features

PDF supports various security features which aid in protecting document contents. They are based on Acrobat's standard encryption handler which uses symmetric encryption. Adobe Reader and Adobe Acrobat 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.
- ▶ (PDF 1.6) Attachments can be encrypted even in otherwise unprotected documents.

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

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.

Unicode passwords. PDF 1.7 extension level 3 (Acrobat 9) supports Unicode passwords. While earlier versions support only passwords with characters in WinAnsi encoding, PDF 1.7 extension level 3 allows arbitrary Unicode characters for the user and master passwords.

Good and bad passwords. The strength of PDF encryption is not only determined by the length of the encryption key, but also by the length and quality of the password. It is widely known that names, plain words, etc. should not be used as passwords since these can easily be guessed or systematically tried using a so-called dictionary attack. Surveys have shown that a significant number of passwords are chosen to be the spouse's or pet's name, the user's birthday, the children's nickname etc., and can therefore easily be guessed.

While PDF encryption internally works with 40-256 bit key length, on the user level passwords of up to 32 characters are used up to PDF 1.7, and passwords with up to 127 UTF-8 bytes with PDF 1.7 extension level 3. The internal key which is used to encrypt the PDF document is derived from the user-supplied password by applying some complicated calculations. If the password is weak, the resulting protection will be weak as well, regardless of the key length. Even long keys and AES encryption are not very secure if short passwords are used.

3.3.2 Protecting Documents with PDFlib

Encryption algorithm and key length. When creating protected documents PDFlib will use the strongest possible encryption and key length which are possible with the PDF compatibility level chosen by the client:

- ▶ For PDF 1.3 (Acrobat 4) RC4 with 40-bit keys is used.
- ▶ For PDF 1.4 (Acrobat 5) RC4 with 128-bit keys is used.
- ▶ For PDF 1.5 (Acrobat 6) RC4 with 128-bit keys is used. This is the same key length as with PDF 1.4, but a slightly different encryption method will be used which requires Acrobat 6.
- ▶ For PDF 1.6 (Acrobat 7) and above the Advanced Encryption Standard (AES) with 128-bit keys will be used.
- ▶ For PDF 1.7 extension level 3 (Acrobat 9) the Advanced Encryption Standard (AES) with 256-bit keys will be used. This version also allows Unicode passwords, while all earlier algorithms support only passwords with characters in WinAnsi encoding.

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 32 characters are accepted. Empty passwords are not allowed.

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

Security recommendations. For maximum security we recommend the following:

- ▶ Use AES encryption unless you must support older viewers than Acrobat 7.
- ▶ Documents which have only a master password, but no user password, can always be cracked. You should therefore consider applying a user password (but of course the user password must be available to legitimate users of the document).
- ▶ Passwords should be at least eight characters long and should contain non-alphabetic characters. Words which can be found in dictionaries and the names of people or places should not be used as passwords.

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:

```
p.begin_document(filename, "masterpassword=abcd1234 permissions={noprint nocopy}");
```

Table 3.3 lists all supported access restriction keywords.

Cookbook A full code sample can be found in the *Cookbook* topic `general/permission_settings`.

Table 3.3 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 editing or cropping pages and creating or changing form fields.
<i>nocopy</i>	Acrobat will prevent copying and extracting text or graphics; the accessibility interface will be controlled by <code>noaccessible</code> .
<i>noannots</i>	Acrobat will prevent creating or changing annotations and form fields.
<i>noforms</i>	(PDF 1.4; implies <code>nomodify</code> and <code>noannots</code>) Acrobat will prevent form field filling.
<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; implies <code>nomodify</code>) Acrobat will prevent inserting, deleting, or rotating pages and creating bookmarks and thumbnails.
<i>nohighresprint</i>	(PDF 1.4) Acrobat will prevent high-resolution printing. If <code>noprint</code> isn't set, printing is restricted to the »print as image« feature which prints a low-resolution rendition of the page.
<i>plainmetadata</i>	(PDF 1.5) Keep XMP document metadata unencrypted even for encrypted documents.

Note When serving PDFs over the Web, clients can always produce a local copy of the document with their browser. There is no way for a PDF to prevent users from saving a local copy.

Encrypted file attachments. In PDF 1.6 and above file attachments can be encrypted even in otherwise unprotected documents. This can be achieved by supplying the `attachmentpassword` option to `PDF_begin_document()`.

3.4 Web-Optimized (Linearized) PDF

PDFlib can apply a process called linearization to PDF documents (linearized PDF is also called *Optimized* or *Fast Web View*). 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. This can be achieved with the *linearize* option in *PDF_begin_document()* as follows:

```
p.begin_document(outfilename, "linearize");
```

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 underlyingbyterange protocol is part of HTTP 1.1 and therefore implemented in all current Web servers.
- The user must use Acrobat as a Browser plugin, and have page-at-a-time download enabled in Acrobat (*Edit, Preferences, [General....] Internet, 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.

3.5 Working with Color

Note The *PDFlib API Reference* contains a detailed list of supported color spaces with descriptions.

Cookbook Code samples regarding color issues can be found in the *color* category of the *PDFlib Cookbook*. For an overview of using color spaces, see the *Cookbook* topic *color/starter_color*.

3.5.1 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 definition 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).

Cookbook Full code samples can be found in the *Cookbook* topics *graphics/fill_pattern* and *images/tiling_pattern*.

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.

3.5.2 Pantone, HKS, and custom 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.

PDFlib supports various built-in spot color libraries as well as custom (user-defined) spot colors. When a spot color name is requested with `PDF_makespotcolor()` PDFlib will first check whether the requested spot color can be found in one of its built-in libraries. If so, PDFlib will use built-in values for the alternate color. Otherwise the spot color is assumed to be a user-defined color, and the client must supply appropriate alternate color values (via the current color). 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, and for workflows which cannot deal with PDFlib's Lab alternate values for PANTONE colors.

PDFlib will automatically generate suitable alternate colors for built-in spot colors when a PDF/X or PDF/A conformance level has been selected (see Section 10.3, »PDF/X for Print Production«, page 247). For custom spot colors it is the user's responsibility to provide alternate colors which are compatible with the selected PDF/X or PDF/A 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.

Cookbook A full code sample can be found in the Cookbook topic `color/spot_color`.

PANTONE® colors. PANTONE colors are well-known and widely used on a world-wide basis. PDFlib fully supports the Pantone Matching System® (totalling ca. 26 000 swatches), plus the Pantone® Goe™ System which was introduced in 2008 with 2058 additional colors in coated/uncoated variants. All color swatch names from the digital color libraries listed in Table 3.4 can be used. Commercial PDFlib customers can request a text file with the full list of PANTONE spot color names from our support.



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. If a spot color is requested where the name starts with the *PANTONE* prefix, but the name does not represent a valid PANTONE color a warning is logged. The following code snippet demonstrates the use of a PANTONE color with a tint value of 70 percent:

```
spot = p.makespotcolor("PANTONE 281 U");
p.setcolor("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 PDF/X-1a mode.

Table 3.4 PANTONE spot color libraries built into PDFlib

color library name	sample color name	remarks
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 process uncoated EURO	PANTONE DE 35-1 U	introduced in May 2006
PANTONE pastel coated	PANTONE 9461 C	includes new colors introduced in 2006
PANTONE pastel uncoated	PANTONE 9461 U	includes new colors introduced in 2006
PANTONE metallic coated	PANTONE 871 C	includes new colors introduced in 2006
PANTONE color bridge CMYK PC	PANTONE 185 PC	replaces PANTONE solid to process coated
PANTONE color bridge CMYK EURO	PANTONE 185 EC	replaces PANTONE solid to process coated EURO
PANTONE color bridge uncoated	PANTONE 185 UP	introduced in July 2006
PANTONE hexachrome coated	PANTONE H 305-1 C	not recommended; will be discontinued
PANTONE hexachrome uncoated	PANTONE H 305-1 U	not recommended; will be discontinued
PANTONE solid in hexachrome coated	PANTONE 185 HC	
PANTONE solid to process coated	PANTONE 185 PC	replaced by PANTONE color bridge CMYK PC
PANTONE solid to process coated EURO	PANTONE 185 EC	replaced by PANTONE color bridge CMYK EURO
PANTONE Goe coated	PANTONE 42-1-1 C	2058 colors introduced in 2008
PANTONE Goe uncoated	PANTONE 42-1-1 U	2058 colors introduced in 2008

HKS® colors. The HKS color system is widely used in Germany and other European countries. PDFlib fully supports HKS colors. 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, 86 colors (*HKS 43 N*)
- ▶ HKS E (*Endlospapier*) for continuous stationary/coated, 88 colors (*HKS 43 E*)
- ▶ HKS Z (*Zeitungspapier*) for newsprint, 50 colors (*HKS 43 Z*)

Commercial PDFlib customers can request a text file with the full list of HKS spot color names from our support.

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. If a spot color is requested where the name starts with the HKS prefix, but the name does not represent a valid HKS color a warning is logged. The following code snippet demonstrates the use of an HKS color with a tint value of 70 percent:

```
spot = p.makespotcolor("HKS 38 E");  
p.setcolor("fill", "spot", spot, 0.7, 0, 0);
```

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

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

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

3.5.3 Color Management and ICC Profiles

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

Cookbook A full code sample can be found in the *Cookbook* topic `color/iccprofile_to_image`.

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()`.

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

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

Note ICC color profiles for common printing conditions are available for download from www.pdflib.com, as well as links to other freely available ICC profiles.

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.

¹ See www.color.org

- 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 *colordir* designates the directory where device-specific ICC profiles are stored by the operating system (typically *C:\WINNT\system32\pool\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.

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

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*:

```
p.set_parameter("errorpolicy", "return");
icchandle = p.load_iccprofile(...);
if (icchandle == -1)
{
    return;
}
p.set_value("setcolor:iccprofilecmyk", icchandle);
p.setcolor("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* options of *PDF_begin_page_ext()* and supplying an ICC profile handle as the corresponding value. The following examples will set the sRGB color space as the default RGB color space for text, images, and vector graphics:

```
/* sRGB is guaranteed to be always available */
icchandle = p.load_iccprofile("sRGB", "usage=iccbased");
p.begin_page_ext(595, 842, "defaultrgb=" + icchandle);
```

Defining output intents for PDF/X and PDF/A. 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 PDF/A any kind of profile can be specified as output intent. For details see Section 10.3, »PDF/X for Print Production«, page 247, and Section 10.4, »PDF/A for Archiving«, page 254.

3.6 Interactive Elements

Cookbook Code samples for creating interactive elements can be found in the interactive category of the PDFlib Cookbook.

3.6.1 Links, Bookmarks, and Annotations

This section explains how to create interactive elements such as bookmarks, form fields, and annotations. Figure 3.1 shows the resulting document with all interactive elements that we will create in this section. The document contains the following interactive elements:

- ▶ At the top right there is an invisible Web link to *www.kraxi.com* at the text *www.kraxi.com*. Clicking this area will bring up the corresponding Web page.
- ▶ A gray form field of type text is located below the Web link. Using JavaScript code it will automatically be filled with the current date.
- ▶ The red pushpin contains an annotation with an attachment. Clicking it will open the attached file.
- ▶ At the bottom left there is a form field of type button with a printer symbol. Clicking this button will execute Acrobat's menu item *File, Print*.
- ▶ The navigation page contains the bookmark »Our Paper Planes Catalog«. Clicking this bookmark will bring up a page of another PDF document.


In the next paragraphs we will show in detail how to create these interactive elements with PDFlib.

Web link. Let's start with a link to the Web site *www.kraxi.com*. This is accomplished in three steps. First, we fit the text on which the Web link should work. Using the *matchbox* option with *name=kraxi* we specify the rectangle of the text's fitbox for further reference.

Second, we create an action of type *URI* (in Acrobat: *Open a web link*). This will provide us with an action handle which subsequently can be assigned to one or more interactive elements.

Third, we create the actual link. A link in PDF is an annotation of type *Link*. The *action* option for the link contains the event name *activate* which will trigger the action, plus the *act* handle created above for the action itself. By default the link will be displayed

For special offers, visit our Web site at www.kraxi.com !

ORDER FORM				DATE	Sep 16 2004
ITEM	DESCRIPTION	QUANTITY	PRICE	AMOUNT	
1	 Long Distance Glider	0	0.00	0.00	
2	Giant Wing	0	0.00	0.00	
3	Cone Head Rocket	0	0.00	0.00	
4	Super Dart	0	0.00	0.00	
			Total:	0.00	




Fig. 3.1
Document with interactive elements

with a thin black border. Initially this is convenient for precise positioning, but we disabled the border with *linewidth=0*.

```
normalfont = p.load_font("Helvetica", "unicode", "");
p.begin_page_ext(pagewidth, pageheight, "topdown");

/* place the text line "Kraxi Systems, Inc." using a matchbox */
String optlist =
    "font=" + normalfont + " fontsize=8 position={left top} " +
    "matchbox={name=kraxi} fillcolor={rgb 0 0 1} underline";

p.fit_textline("Kraxi Systems, Inc.", 2, 20, optlist);

/* create URI action */
optlist = "url={http://www.kraxi.com}";
int act = p.create_action("URI", optlist);

/* create Link annotation on matchbox "kraxi" */
optlist = "action={activate " + act + "} linewidth=0 usematchbox={kraxi}";
/* 0 rectangle coordinates will be replaced with matchbox coordinates */
p.create_annotation(0, 0, 0, 0, "Link", optlist);

p.end_page_ext("");
```

For an example of creating a Web link on an image or on parts of a textflow, see Section 8.4, »Matchboxes«, page 237.

Cookbook A full code sample can be found in the *Cookbook* topic *interactive/link_annotations*.

Bookmark for jumping to another file. Now let's create the bookmark »Our Paper Planes Catalog« which jumps to another PDF file called *paper_planes_catalog.pdf*. First we create an action of Type *GoToR*. In the option list for this action we define the name of the target document with the *filename* option; the *destination* option specifies a certain part of the page which will be enlarged. More precisely, the document will be displayed on the second page (*page 2*) with a fixed view (*type fixed*), where the middle of the page is visible (*left 50 top 200*) and the zoom factor is 200% (*zoom 2*):

```
String optlist =
    "filename=paper_planes_catalog.pdf " +
    "destination={page 2 type fixed left 50 top 200 zoom 2}";

goto_action = p.create_action("GoToR", optlist);
```

In the next step we create the actual bookmark. The *action* option for the bookmark contains the *activate* event which will trigger the action, plus the *goto_action* handle created above for the desired action. The option *fontstyle bold* specifies bold text, and *textcolor {rgb 0 0 1}* makes the bookmark blue. The bookmark text »Our Paper Planes Catalog« is provided as a function parameter:

```
String optlist =
    "action={activate " + goto_action + "} fontstyle=bold textcolor={rgb 0 0 1}";

catalog_bookmark = p.create_bookmark("Our Paper Planes Catalog", optlist);
```

Clicking the bookmark will display the specified part of the page in the target document.

Cookbook A full code sample can be found in the *Cookbook topic* [interactive/nested_bookmarks](#).

Annotation with file attachment. In the next example we create a file attachment. We start by creating an annotation of type *FileAttachment*. The *filename* option specifies the name of the attachment, the option *mimetype image/gif* specifies its type (MIME is a common convention for classifying file contents). The annotation will be displayed as a pushpin (*iconname pushpin*) in red (*annotcolor {rgb 1 0 0}*) and has a tooltip (*contents {Get the Kraxi Paper Plane!}*). It will not be printed (*display noprint*):

```
String optlist =
    "filename=kraxi_logo.gif mimetype=image/gif iconname=pushpin " +
    "annotcolor={rgb 1 0 0} contents={Get the Kraxi Paper Plane!} display=noprint";

p.create_annotation(left_x, left_y, right_x, right_y, "FileAttachment", optlist);
```

Note that the size of the symbol defined with *iconname* does not vary; the icon will be displayed in its standard size in the top left corner of the specified rectangle.

3.6.2 Form Fields and JavaScript

Button form field for printing. The next example creates a button form field which can be used for printing the document. In the first version we add a caption to the button; later we will use a printer symbol instead of the caption. We start by creating an action of type *Named* (in Acrobat: *Execute a menu item*). Also, we must specify the font for the caption:

```
print_action = p.create_action("Named", "menuname=Print");
button_font = p.load_font("Helvetica-Bold", "unicode", "");
```

The *action* option for the button form field contains the *up* event (in Acrobat: *Mouse Up*) as a trigger for executing the action, plus the *print_action* handle created above for the action itself. The *backgroundcolor {rgb 1 1 0}* option specifies yellow background, while *bordercolor {rgb 0 0 0}* specifies black border. The option *caption Print* adds the text *Print* to the button, and *tooltip {Print the document}* creates an additional explanation for the user. The *font* option specifies the font using the *button_font* handle created above. By default, the size of the caption will be adjusted so that it completely fits into the button's area. Finally, the actual button form field is created with proper coordinates, the name *print_button*, the type *pushbutton* and the appropriate options:

```
String optlist =
    "action {up " + print_action + "} backgroundcolor=yellow " +
    "bordercolor=black caption=Print tooltip={Print the document} font=" +
    button_font;

p.create_field(left_x, left_y, right_x, right_y, "print_button", "pushbutton", optlist);
```

Now we extend the first version of the button by replacing the text *Print* with a little printer icon. To achieve this we load the corresponding image file *print_icon.jpg* as a template before creating the page. Using the *icon* option we assign the template handle *print_icon* to the button field, and create the form field similarly to the code above:

```
print_icon = p.load_image("auto", "print_icon.jpg", "template");
if (print_icon == -1)
```

```

{
    /* Error handling */
    return;
}
p.begin_page_ext(pagewidth, pageheight, "");
...
String optlist = "action={up " + print_action + "} icon=" + print_icon +
    " tooltip={Print the document} font=" + button_font;

p.create_field(left_x, left_y, right_x, right_y, "print_button", "pushbutton", optlist);

```

Cookbook A full code sample can be found in the *Cookbook* topic [interactive/form_pushbutton](#).

Simple text field. Now we create a text field near the upper right corner of the page. The user will be able to enter the current date in this field. We acquire a font handle and create a form field of type *textfield* which is called *date*, and has a gray background:

```

textfield_font = p.load_font("Helvetica-Bold", "unicode", "");
String optlist = "backgroundcolor={gray 0.8} font=" + textfield_font;
p.create_field(left_x, left_y, right_x, right_y, "date", "textfield", optlist);

```

By default the font size is *auto*, which means that initially the field height is used as the font size. When the input reaches the end of the field the font size is decreased so that the text always fits into the field.

Cookbook Full code samples can be found in the *Cookbook* topics in [teractive/form_textfield_layout](#) and [interactive/form_textfield_height](#).

Text field with JavaScript. In order to improve the text form field created above we automatically fill it with the current date when the page is opened. First we create an action of type *JavaScript* (in Acrobat: *Run a JavaScript*). The *script* option in the action's option list defines a JavaScript snippet which displays the current date in the *date* text field in the format month-day-year:

```

String optlist =
    "script={var d = util.printd('mmm dd yyyy', new Date()); "
    "var date = this.getField('date'); date.value = d;}";

show_date = p.create_action("JavaScript", optlist);

```

In the second step we create the page. In the option list we supply the *action* option which attaches the *show_date* action created above to the trigger event *open* (in Acrobat: *Page Open*):

```

String optlist = "action={open " + show_date + "}";
p.begin_page_ext(pagewidth, pageheight, optlist);

```

Finally we create the text field as we did above. It will automatically be filled with the current date whenever the page is opened:

```

textfield_font = p.load_font("Helvetica-Bold", "winansi", "");
String optlist = "backgroundcolor={gray 0.8} font=" + textfield_font;
p.create_field(left_x, left_y, right_x, right_y, "date", "textfield", optlist);

```

Cookbook A full code sample can be found in the *Cookbook* topic [interactive/form_textfield_fill_with_js](#).

Formatting Options for Text Fields. In Acrobat it is possible to specify various options for formatting the contents of a text field, such as monetary amounts, dates, or percentages. This is implemented via custom JavaScript code used by Acrobat. PDFlib does not directly support these formatting features since they are not specified in the PDF reference. However, for the benefit of PDFlib users we present some information below which will allow you to realize formatting options for text fields by supplying simple JavaScript code fragments with the *action* option of *PDF_create_field()*.

In order to apply formatting to a text field JavaScript snippets are attached to a text field as *keystroke* and *format* actions. The JavaScript code calls some internal Acrobat function where the parameters control details of the formatting.

The following sample creates two *keystroke* and *format* actions, and attaches them to a form field so that the field contents will be formatted with two decimal places and the EUR currency identifier:

```
keystroke_action = p.create_action("JavaScript",
    "script={AFNumber_Keystroke(2, 0, 3, 0, \"EUR \", true); }");

format_action = p.create_action("JavaScript",
    "script={AFNumber_Format(2, 0, 0, 0, \"EUR \", true); }");

String optlist = "font=" + font + " action={keystroke " + keystroke_action +
    " format=" + format_action + "}";
p.create_field(50, 500, 250, 600, "price", "textfield", optlist);
```

Cookbook A full code sample can be found in the *Cookbook* topic [interactive/form_textfield_input_format](#).

In order to specify the various formats which are supported in Acrobat you must use appropriate functions in the JavaScript code. Table 3.5 lists the JavaScript function names for the *keystroke* and *format* actions for all supported formats; the function parameters are described in Table 3.6. These functions must be used similarly to the example above.

Table 3.5 JavaScript formatting functions for text fields

format	JavaScript functions to be used for keystroke and format actions
number	AFNumber_Keystroke(nDec, sepStyle, negStyle, currStyle, strCurrency, bCurrencyPrepend) AFNumber_Format(nDec, sepStyle, negStyle, currStyle, strCurrency, bCurrencyPrepend)
percentage	AFPercent_Keystroke(ndec, sepStyle), AFPercent_Format(ndec, sepStyle)
date	AFDate_KeystrokeEx(cFormat), AFDate_FormatEx(cFormat)
time	AFTIME_Keystroke(tFormat), AFTIME_FormatEx(cFormat)
special	AFSpecial_Keystroke(psf), AFSpecial_Format(psf)

Table 3.6 Parameters for the JavaScript formatting functions

parameters	explanation and possible values	
nDec	Number of decimal places	
sepStyle	The decimal separator style:	
	0	1,234.56
	1	1234.56
	2	1.234,56
	3	1234,56
negStyle	Emphasis used for negative numbers:	
	0	Normal
	1	Use red text
	2	Show parenthesis
	3	both
strCurrency	Currency string to use, e.g. \u20AC for the Euro sign	
bCurrency-Prepend	false	do not prepend currency symbol
	true	prepend currency symbol
cFormat	A date format string. It may contain the following format placeholders, or any of the time formats listed below for tFormat :	
	d	day of month
	dd	day of month with leading zero
	ddd	abbreviated day of the week
	m	month as number
	mm	month as number with leading zero
	mmm	abbreviated month name
	mmmm	full month name
	yyyy	year with four digits
	yy	last two digits of year
tFormat	A time format string. It may contain the following format placeholders:	
	h	hour (0-12)
	hh	hour with leading zero (0-12)
	H	hour (0-24)
	HH	hour with leading zero (0-24)
	M	minutes
	MM	minutes with leading zero
	s	seconds
	ss	seconds with leading zero
	t	'a' or 'p'
	tt	'am' or 'pm'
psf	Describes a few additional formats:	
	0	Zip Code
	1	Zip Code + 4
	2	Phone Number
	3	Social Security Number

3.7 Georeferenced PDF

Cookbook A full code sample can be found in the *Cookbook topic* [interactive/starter_geospatial](#).

3.7.1 Using Georeferenced PDF in Acrobat

PDF 1.7ext3 allows geospatial reference information (world coordinates) to be added to PDF page contents. Geospatially referenced PDF documents can be used in Acrobat 9 and above for several purposes (only the first two features are available in Adobe Reader):

- ▶ display the coordinates of the map point under the mouse cursor: *Tools, Analysis/Analyze, Geospatial Location Tool* (in Acrobat X you may have to activate the *Analyze* toolbar using the button at the top of the *Tools* pane). You can copy the coordinates of the map point under the mouse cursor by right-clicking and selecting *Copy Coordinates to Clipboard*;
- ▶ search for a location on the map: *Tools, Analysis/Analyze, Geospatial Location Tool*, right-click and select *Find a Location*, and enter the desired coordinates;
- ▶ mark a location on the map: *Tools, Analysis, Geospatial Location Tool*, right-click and select *Mark Location*;
- ▶ measure distance, perimeter and area on geographic maps: *Tools, Analysis, Measuring Tool*;

Various settings for geospatial measuring can be changed in *Edit, Preferences, General..., Measuring (Geo)*, e.g. the preferred coordinate system for coordinate readouts.

Geospatial features in PDFlib are implemented with the following functions and options:

- ▶ One or more georeferenced areas can be assigned to a page with the *viewports* option of *PDF_begin/end_page_ext()*. Viewports allow different geospatial references (specified by the *georeference* option) to be used on different areas of the page, e.g. for multiple maps on the same page.
- ▶ The *georeference* option of *PDF_load_image()* can be used to assign an earth-based coordinate system to an image.
- ▶ The *georeference* option could as well be offered in *PDF_open_pdi_page()* and *PDF_begin_template_ext()* to assign an earth-based coordinate system to an imported PDF page or a template. Unfortunately, this doesn't work in Acrobat 9 and Acrobat X and therefore isn't supported.

3.7.2 Geographic and projected Coordinate Systems

A geographic coordinate system describes the earth in geographic coordinates, i.e. angular units of latitude and longitude. A projected coordinate system can be specified on top of a geographic coordinate system and describes the transformation of points in geographic coordinates to a two-dimensional (projected) coordinate system. The resulting coordinates are called Northing and Easting values; degrees are no longer required for projected coordinate systems. While geographic coordinate systems are in use for GPS and other global applications, projections are required for map-making and other applications with more or less local character.

For historical and mathematical reasons a variety of different coordinate systems is in use around the world. Both geographic and projected coordinate systems can be described using two well-established methods which are called EPSG and WKT.

EPSG. EPSG is a collection of thousands of coordinate systems which are referenced via numeric codes. EPSG is named after the defunct *European Petroleum Survey Group* and now maintained by the International Association of Oil and Gas Producers (OGP).

EPSG reference codes point to one of the coordinate systems in the EPSG database. The full EPSG database can be downloaded from the following location:

www.epsg.org

Well-known text (WKT). The WKT (*Well-Known Text*) system is descriptive and consists of a textual specification of all relevant parameters of a coordinate system. WKT is specified in the document *OpenGIS® Implementation Specification: Coordinate Transformation Services*, which has been published as Document 01-009 by the Open Geospatial Consortium (OGC). It is available at the following location:

www.opengeospatial.org/standards/ct

WKT has also been standardized in ISO 19125-1. Although both WKT and EPSG can be used in Acrobat (and are supported in PDFlib), Acrobat does not implement all possible EPSG codes. In particular, EPSG codes for geographic coordinate systems don't seem to be supported in Acrobat. In this case the use of WKT is recommended. The following Web site delivers the WKT corresponding to a particular EPSG code:

www.spatialreference.org/ref/epsg

3.7.3 Coordinate System Examples

Examples for geographic coordinate systems. The WGS84 (World Geodetic System) geographic coordinate system is the basis for GPS and many applications (e.g. OpenStreetMap). It can be expressed as follows in the *worldsystem* suboption of the *geo-reference* option:

```
worldsystem={type=geographic wkt={
GEOGCS["WGS_84",
    DATUM["WGS_1984", SPHEROID["WGS_84", 6378137, 298.257223563]],
    PRIMEM["Greenwich", 0],
    UNIT["degree", 0.01745329251994328]]
}}
```

The ETRS (European Terrestrial Reference System) geographic coordinate system is almost identical to WGS84. It can be specified as follows:

```
worldsystem={type=geographic wkt={
GEOGCS["ETRS_1989",
    DATUM["ETRS_1989", SPHEROID["GRS_1980", 6378137.0, 298.257222101]],
    PRIMEM["Greenwich", 0.0],
    UNIT["Degree", 0.0174532925199433]]
}}
```

Note EPSG codes for the WGS84 and ETRS systems are not shown here because Acrobat doesn't seem to support EPSG codes for geographic coordinate systems, but only for projected coordinate systems (see below).

Examples for projected coordinate systems. A projection is based on an underlying geographic coordinate system. In the following example we specify a projected coordinate system suitable for use with GPS coordinates.

In middle Europe the system called ETRS89 UTM zone 32 N applies. It uses the common UTM (Universal Mercator Projection), and can be expressed as follows in the *worldsystem* suboption of the *georeference* option:

```
worldsystem={type=projected wkt={
  PROJCS["ETRS_1989_UTM_Zone_32N",
    GEOGCS["GCS_ETRS_1989",
      DATUM["D_ETRS_1989", SPHEROID["GRS_1980", 6378137.0, 298.257222101],
      TOWGS84[0, 0, 0, 0, 0, 0, 0]],
      PRIMEM["Greenwich", 0.0],
      UNIT["Degree", 0.0174532925199433]],
    PROJECTION["Transverse_Mercator"],
    PARAMETER["False_Easting", 500000.0],
    PARAMETER["False_Northing", 0.0],
    PARAMETER["Central_Meridian", 9.0],
    PARAMETER["Scale_Factor", 0.9996],
    PARAMETER["Latitude_Of_Origin", 0.0],
    UNIT["Meter", 1.0]]
}}
```

The corresponding EPSG code for this coordinate system is 25832. As an alternative to WKT, the system above can also be specified via its EPSG code as follows:

```
worldsystem={type=projected epsg=25832}
```

3.7.4 Georeferenced PDF restrictions in Acrobat

We experienced the following shortcomings when working with georeferenced PDF in Acrobat 9 and Acrobat X:

- ▶ EPSG codes don't seem to work at all for geographic coordinate systems, but only for projected systems.
Workaround: use the corresponding WKT instead of the EPSG code.
- ▶ Attaching geospatial data to vector objects does not work. For this reason PDFlib does not support the *georeference* option for *PDF_open_pdi_page()* and *PDF_begin_template_ext()*, although in theory this should work according to the PDF Reference.
Workaround for creating vector-based maps: you can attach the geospatial data to the page, i.e. use the *viewports* option of *PDF_begin_page_ext()*.
- ▶ Overlapping maps: you can place multiple image-based maps on the same page. If the maps overlap and you display the coordinates of a point in the overlapping area, Acrobat will use the coordinates of the map which has been placed last (this makes sense since this is also the map which is visible). However, if both image handles are identical (i.e. retrieved with a single call to *PDF_load_image()*), Acrobat does no longer take into account the different image geometries: the coordinates of the first image will incorrectly be extended to the area of second image, resulting in wrong coordinate readouts.
Workaround: if you need multiple instances of the same image-based map on the same page, open the image multiply.
- ▶ The area measurement tool doesn't work correctly for geographic coordinate systems, but only for projected systems.



4 Unicode and Legacy Encodings

This chapter provides basic information about Unicode and other encoding schemes. Text handling in PDFlib heavily relies on the Unicode standard, but also supports various legacy and special encodings.

4.1 Important Unicode Concepts

Characters and glyphs. When dealing with text it is important to clearly distinguish the following concepts:

- ▶ *Characters* are the smallest units which convey information in a language. Common examples are the letters in the Latin alphabet, Chinese ideographs, and Japanese syllables. Characters have a meaning; they are semantic entities.
- ▶ *Glyphs* are different graphical variants which represent one or more particular characters. Glyphs have an appearance: they are representational entities.

There is no one-to-one relationship between characters and glyphs. For example, a ligature is a single glyph which is represented by two or more separate characters. On the other hand, a specific glyph may be used to represent different characters depending on the context (some characters look identical, see Figure 4.1).

BMP and PUA. The following terms will occur frequently in Unicode-based environments:

- ▶ The *Basic Multilingual Plane (BMP)* comprises the code points in the Unicode range U+0000...U+FFFF. The Unicode standard contains many more code points in the supplementary planes, i.e. in the range U+10000...U+10FFFF.
- ▶ A *Private Use Area (PUA)* is one of several ranges which are reserved for private use. PUA code points cannot be used for general interchange since the Unicode standard does not specify any characters in this range. The Basic Multilingual Plane includes a PUA in the range U+E000...U+F8FF. Plane fifteen (U+F0000...U+FFFFFD) and plane sixteen (U+100000...U+10FFFFD) are completely reserved for private use.

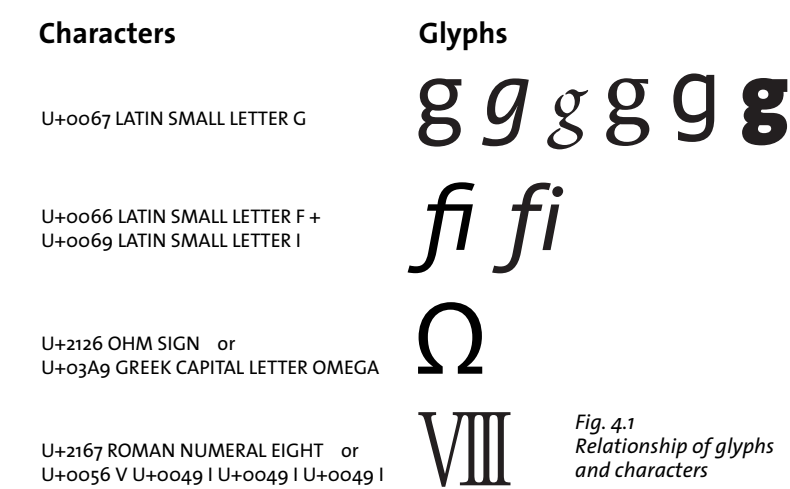


Fig. 4.1
Relationship of glyphs
and characters

Unicode encoding forms (UTF formats). The Unicode standard assigns a number (code point) to each character. In order to use these numbers in computing, they must be represented in some way. In the Unicode standard this is called an encoding form (formerly: transformation format); this term should not be confused with font encodings. Unicode defines the following encoding forms:

- ▶ **UTF-8:** This is a variable-width format where code points are represented by 1-4 bytes. ASCII characters in the range U+0000...U+007F are represented by a single byte in the range 00...7F. Latin-1 characters in the range U+00A0...U+00FF are represented by two bytes, where the first byte is always 0xC2 or 0xC3 (these values represent *Â* and *Ã* in Latin-1).
- ▶ **UTF-16:** Code points in the Basic Multilingual Plane (BMP) are represented by a single 16-bit value. Code points in the supplementary planes, i.e. in the range U+10000...U+10FFFF, are represented by a pair of 16-bit values. Such pairs are called surrogate pairs. A surrogate pair consists of a high-surrogate value in the range D800...DBFF and a low-surrogate value in the range DC00...DFFF. High- and low-surrogate values can only appear as parts of surrogate pairs, but not in any other context.
- ▶ **UTF-32:** Each code point is represented by a single 32-bit value.

Cookbook A full code sample can be found in the *Cookbook* topic `text_output/process_utf8`.

Unicode encoding schemes and the Byte Order Mark (BOM). Computer architectures differ in the ordering of bytes, i.e. whether the bytes constituting a larger value (16- or 32-bit) are stored with the most significant byte first (big-endian) or the least significant byte first (little-endian). A common example for big-endian architectures is PowerPC, while the x86 architecture is little-endian. Since UTF-8 and UTF-16 are based on values which are larger than a single byte, the byte-ordering issue comes into play here. An encoding scheme (note the difference to encoding form above) specifies the encoding form plus the byte ordering. For example, UTF-16BE stands for UTF-16 with big-endian byte ordering. If the byte ordering is not known in advance it can be specified by means of the code point U+FEFF, which is called Byte Order Mark (BOM). Although a BOM is not required in UTF-8, it may be present as well, and can be used to identify a stream of bytes as UTF-8. Table 4.1 lists the representation of the BOM for various encoding forms.

Table 4.1 Byte order marks for various Unicode encoding forms

Encoding form	Byte order mark (hex)	graphical representation in WinAnsi ¹
UTF-8	EF BB BF	ï»¿
UTF-16 big-endian	FE FF	þÿ
UTF-16 little-endian	FF FE	ÿþ
UTF-32 big-endian	00 00 FE FF	■ ■ þÿ
UTF-32 little-endian	FF FE 00 00	ÿþ ■ ■

1. The black square ■ denotes a null byte.

4.2 Single-Byte (8-Bit) Encodings

8-bit encodings (also called single-byte encodings) map a byte value 0x01-0xFF to a single character with a Unicode value in the BMP (i.e. U+0000...U+FFFF). They are limited to 255 different characters at a time since code 0 (zero) is reserved for the *.notdef* character U+0000. Table 4.2 lists the predefined encodings in PDFlib. It is important to keep in mind that certain scripts or languages have requirements which cannot be met by common fonts.

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

Table 4.2 Predefined encodings in PDFlib

code page	supported languages
winansi	identical to cp1252 (superset of iso8859-1)
macroman	Mac Roman encoding, the original Macintosh character set
macroman_apple	similar to macroman, but replaces currency with Euro and includes additional mathematical/greek symbols
ebcdic	EBCDIC code page 1047
ebcdic_37	EBCDIC code page 037
pdfdoc	PDFDocEncoding
iso8859-1	(Latin-1) Western European languages
iso8859-2	(Latin-2) Slavic languages of Central Europe
iso8859-3	(Latin-3) Esperanto, Maltese
iso8859-4	(Latin-4) Estonian, the Baltic languages, Greenlandic
iso8859-5	Bulgarian, Russian, Serbian
iso8859-6	Arabic
iso8859-7	Modern Greek
iso8859-8	Hebrew and Yiddish
iso8859-9	(Latin-5) Western European, Turkish
iso8859-10	(Latin-6) Nordic languages
iso8859-13	(Latin-7) Baltic languages
iso8859-14	(Latin-8) Celtic
iso8859-15	(Latin-9) Adds Euro as well as French and Finnish characters to Latin-1
iso8859-16	(Latin-10) Hungarian, Polish, Romanian, Slovenian
cp1250	Central European
cp1251	Cyrillic
cp1252	Western European (same as winansi)
cp1253	Greek
cp1254	Turkish
cp1255	Hebrew
cp1256	Arabic
cp1257	Baltic
cp1258	Viet Nam

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 (see Table 4.2):

- ▶ on IBM zSeries with MVS or USS it will be mapped to *ebcdic*;
- ▶ on IBM i5/iSeries it will be mapped to *ebcdic_37*;
- ▶ 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 for text fonts as follows:

- ▶ On Windows: the current system code page (see below for details)
- ▶ On Unix and Mac OS X: *iso8859-1* (except LWFN PostScript fonts on the Mac for which *auto* will be mapped to *macroman*)
- ▶ On IBM i5/iSeries: the current job's encoding (*IBMCCSID000000000000*)
- ▶ On IBM zSeries: *ebcdic* (=code page 1047).

For symbol fonts the keyword *auto* is mapped to *builtin* encoding (see Section 5.4.2, »Selecting an Encoding for symbolic Fonts«, page 123). While automatic encoding is convenient in many circumstances, using this method will make your PDFlib client programs inherently non-portable.

Encoding *auto* is used as the default encoding for Name strings (see Section 4.4.3, »Strings in non-Unicode-aware Language Bindings«, page 103) in non-Unicode-aware language bindings, since this is the most appropriate encoding for file names etc.

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

- ▶ 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 6.5.2, »Custom CJK Fonts«, page 168, for information on multi-byte Windows code pages):

```
font = p.load_font("Helvetica", "cp1250", "");
```

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

- ▶ On IBM i5/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:


```
font = p.load_font("Helvetica", "273", "");
```

- ▶ On IBM 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:

```
font = p.load_font("Helvetica", "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 3.1.3, »Resource Configuration and File Search«, page 56) 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
...
```

If no Unicode value has been specified PDFlib will search for a suitable Unicode value in its internal tables. A Unicode value can be specified instead of a glyph name:

```
% 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.
- ▶ All Unicode values in an encoding or codepage file must be smaller than U+FFFF.

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.

4.3 Chinese, Japanese, and Korean Encodings

Historically, a wide variety of CJK encoding schemes have been developed by diverse standards bodies and companies. 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.

Predefined CMaps for common CJK encodings. The predefined CJK CMaps are listed in Table 4.3. As can be seen from the table, they support most CJK encodings used on Mac, Windows, and Unix systems, as well as several vendor-specific encodings, e.g. Shift-JIS, EUC, and ISO 2022 for Japanese, GB and Big5 for Chinese, and KSC for Korean. Unicode is supported for all locales as well.

Note Unicode-aware language bindings support only Unicode CMaps (UCS2 or UTF16). Other CMaps can not be used (see Section 4.4.2, »Strings in Unicode-aware Language Bindings«, page 103).

CJK text encoding for standard CMaps. The client is responsible for supplying text encoded such that it matches the requested CMap. PDFlib checks whether the supplied text conforms to the requested CMap, and will raise an exception for bad text input which doesn't conform to the selected CMap.

CMap configuration. In order to create Chinese, Japanese, or Korean (CJK) text output with one of the predefined CMaps PDFlib requires the corresponding CMap files for processing the incoming text and mapping CJK encodings to Unicode. The CMap files are available in a separate package. They should be installed as follows:

- ▶ On Windows the CMap files will be found automatically if you place them in the *resource/cmap* directory within the PDFlib installation directory.
- ▶ On other systems you can place the CMap files at any convenient directory, and must manually configure the CMap files by setting the *SearchPath* at runtime:

```
p.set_parameter("SearchPath", "/path/to/resource/cmap");
```

As an alternative method for configuring access to the CJK CMap files you can set the *PDFLIBRESOURCEFILE* environment variable to point to a UPR configuration file which contains a suitable *SearchPath* definition.

Note On MVS the CMap files must be installed from an alternate package which contains CMaps with shortened file names.

Code pages for custom CJK fonts. On Windows PDFlib supports any CJK code page installed on the system. On other platforms the code pages listed in Table 4.4 can be used. These code pages will be mapped internally to the corresponding CMap (e.g. *cp932* will be mapped to *goms-RKSJ-H/V*). Because of this mapping the appropriate CMaps must be configured (see above).

Table 4.3 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
	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
Japanese	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
	90ms-RKSJ-H 90ms-RKSJ-V	Microsoft Code Page 932 (charset 128), JIS X 0208 character set with NEC and IBM extensions
	90msp-RKSJ-H 90msp-RKSJ-V	Same as 90ms-RKSJ-H, but replaces half-width Latin characters with proportional forms
	90pv-RKSJ-H	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
	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
Korean	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

Table 4.4 CJK code pages (must be used with textformat=auto or textformat=bytes)

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

4.4 String Handling in PDFlib

4.4.1 Content Strings, Hypertext Strings, and Name Strings

PDF and operating system requirements impose different string handling in PDFlib depending on the purpose of a string. The PDFlib API uses the string types below. All relevant parameters and options are marked as content string, hypertext string, or name string in the PDFlib API Reference.

Content strings, hypertext strings, and name strings can be used with Unicode and 8-bit encodings. Non-Unicode CJK CMaps can only be used in non-Unicode-aware language bindings. The details of string handling depend on the language binding, and are discussed in Section 4.4.2, »Strings in Unicode-aware Language Bindings«, page 103 and Section 4.4.3, »Strings in non-Unicode-aware Language Bindings«, page 103.

Content strings. Content strings are used to create genuine page content (page descriptions) according to the encoding chosen by the user for a particular font. All *function* parameters with the name *text* in the PDFlib API Reference for the page content functions fall in this class. Since content strings are represented with glyphs from a particular font, the range of usable characters depends on the font/encoding combination. Examples:

The *text* parameters of *PDF_show()*, *PDF_fit_textline()*, *PDF_add_textflow()*.

Hypertext strings. Hypertext strings are used for interactive features such as bookmarks and annotations, and are explicitly labeled *Hypertext string* in the function descriptions. Many parameters and options of the functions for interactive features fall in this class, as well as some others. The range of characters which can be displayed depends on external factors, such as the fonts available to Acrobat and the operating system.

Examples:

fieldname option of *PDF_add_table_cell()*
name option of *PDF_define_layer()*
destname option of *PDF_create_action()*
text parameter of *PDF_create_bookmark()*

Name strings. 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 language bindings which are not Unicode-aware.

Examples:

filename parameters of *PDF_begin_document()* and *PDF_create_pvf()*
fontname parameter of *PDF_load_font()*
profilename parameter of *PDF_load_iccprofile()*

File names are a special case: the option *filenamehandling* specifies how PDFlib converts filenames supplied to the API to something which can be used with the local file system.

4.4.2 Strings in Unicode-aware Language Bindings

If a development environment supports the string data type and uses Unicode internally we call the binding Unicode-aware. The following PDFlib language bindings are Unicode-aware:

- ▶ C++
- ▶ COM
- ▶ .NET
- ▶ Java
- ▶ Objective-C
- ▶ Python
- ▶ REALbasic
- ▶ RPG
- ▶ Tcl

String handling in these environments is straightforward: all strings will automatically be provided to the PDFlib kernel as Unicode strings in native 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:

- ▶ The PDFlib language wrapper applies all required conversions so that client-supplied 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-aware languages. It must be converted to UTF-16 before.
- ▶ Using *unicode* encoding for the contents of a page is the easiest way to deal with encodings in Unicode-aware languages, but 8-bit encodings and single-byte text for symbol fonts can also be used if so desired.
- ▶ Non-Unicode CMaps for Chinese, Japanese, and Korean text (see Section 4.3, «Chinese, Japanese, and Korean Encodings», page 99) 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. The distinction between the string types in the function descriptions is not relevant for Unicode-aware language bindings.

Unicode conversion functions. If you must deal with strings in other encodings than Unicode, you must convert them to Unicode before passing them to PDFlib. The language-specific sections in Chapter 2, «PDFlib Language Bindings», page 27, provide more details regarding useful Unicode string conversion methods provided by common language environments.

4.4.3 Strings in non-Unicode-aware Language Bindings

The following PDFlib language bindings are not Unicode-aware:

- ▶ C (no native string data type available)
- ▶ legacy C++ binding for compatibility with PDFlib 7 (see Section 2.5, «C++ Binding», page 33, for configuration information)
- ▶ Cobol (no native string data type available)
- ▶ Perl
- ▶ PHP
- ▶ Ruby

In language bindings which do not support a native string data type (i.e. C, Cobol) the length of UTF-16 strings must be supplied in a separate *length* parameter. Although Unicode text can be used in these languages, handling of the various string types is a bit more complicated:

Content strings. Content strings are strings used to create 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 and UTF-16 CMaps. For all other encodings the format will be *bytes*. In languages without a native string data type (see list above) 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 without a native string data type (see list above) the length of UTF-16 strings must be supplied in a separate *length* parameter.

Name strings. Name strings are interpreted slightly differently from page description strings. By default, name strings are interpreted in *host* encoding. However, if it starts with an UTF-8 BOM it will be interpreted as UTF-8 (or as EBCDIC UTF-8 if it starts with an EBCDIC UTF-8 BOM). If the *usehypertextencoding* parameter is *true*, the encoding specified in *hypertextencoding* will be applied to name strings as well. This can be used, for example, to specify font or file names in Shift-JIS. If *hypertextencoding=unicode* PDFlib expects a UTF-16 string which must be terminated by two null bytes.

In C the *length* parameter must be 0 for UTF-8 strings. If it is different from 0 the string will be interpreted as UTF-16. In all other non-Unicode-aware language bindings there is no *length* parameter available in the API functions, and name strings must always be supplied in UTF-8 format. In order to create Unicode name strings in this case you can use the *PDF_utf16_to_utf8()* utility function to create UTF-8 (see below).

Unicode conversion functions. In non-Unicode-aware language bindings PDFlib offers the *PDF_utf16_to_utf8()*, *PDF_utf8_to_utf16()* and related conversion functions which can be used to convert between UTF-8, UTF-16, and UTF-32 strings.

The language-specific sections in Chapter 2, »PDFlib Language Bindings«, page 27, provide more details regarding useful Unicode string conversion methods provided by common language environments.

Text format for content and hypertext strings. Unicode strings in PDFlib can be supplied in the UTF-8, UTF-16, or UTF-32 formats with any byte ordering. The choice of format can be controlled with the *textformat* parameter for all text on page descriptions, and the *hypertextformat* parameter for interactive elements. Table 4.5 lists the values which are supported for both of these parameters. The default for the *[hyper]textformat* parameter is *auto*. Use the *usehypertextencoding* parameter to enforce the same behavior for name strings. The default for the *hypertextencoding* parameter is *auto*.

Although the *textformat* setting is in effect for all encodings, it will be most useful for *unicode* encoding. Table 4.6 details the interpretation of text strings for various combinations of encodings and *textformat* settings. If a code or Unicode value in a content

Table 4.5 Values for the textformat and hypertextformat parameters

[hyper]textformat	explanation
bytes	One byte in the string corresponds to one character. This is mainly useful for 8-bit encodings and symbolic fonts. A UTF-8 BOM at the start of the string will be evaluated and then removed.
utf8	Strings are expected in UTF-8 format. Invalid UTF-8 sequences will trigger an exception if glyphcheck=error, or will be deleted otherwise.
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	<p>Content strings: equivalent to bytes for 8-bit encodings and non-Unicode CMaps, and utf16 for wide-character addressing (unicode, glyphid, or a UCS2 or UTF16 CMap).</p> <p>Hypertext strings: UTF-8 and UTF-16 strings with BOM will be detected (in C UTF-16 strings must be terminated with a double-null). If the string does not start with a BOM, it will be interpreted as an 8-bit encoded string according to the hypertextencoding parameter.</p> <p>This setting will provide proper text interpretation in most environments which do not use Unicode natively.</p>

string cannot be represented with a suitable glyph in the selected font, the option `glyphcheck` controls the behavior of PDFlib (see »Glyph replacement«, page 119).

Table 4.6 Relationship of encodings and text format

[hypertext]encoding	textformat=bytes	textformat=utf8, utf16, utf16be, or utf16le
All string types:		
auto	see section »Automatic encoding«, page 96	
unicode and UCS2- or UTF16 CMaps	8-bit codes are Unicode values from U+0000 to U+00FF	any Unicode value, encoded according to the chosen text format ¹
any other CMap (not Unicode-based)	any single- or multibyte codes according to the selected CMap	PDFlib will throw an exception
Only content strings:		
8-bit and builtin	8-bit codes	Convert Unicode values to 8-bit codes according to the chosen encoding ¹ .
glyphid	8-bit codes are glyph ids from 0 to 255	Unicode values will be interpreted as glyph ids ² . Surrogate pairs will not be interpreted.

1. If the Unicode character is not available in the font, PDFlib will throw an exception or replace it subject to the `glyphcheck` option.
2. If the glyph id is not available in the font, PDFlib will throw an exception or replace it with glyph id 0 subject to the `glyphcheck` setting.

Strings in option lists. Strings within option lists require special attention since in non-Unicode-aware language bindings 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 op-

tions. 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 (*0xEF 0xBB 0xBF*) it will be interpreted as UTF-8. On EBCDIC-based systems: if the option starts with an EBCDIC UTF-8 BOM (*0x57 0x8B 0xAB*) it will be interpreted as EBCDIC UTF-8. If no BOM is found, string interpretation depends on the type of string:
- ▶ Content strings will be interpreted according to the applicable *encoding* option or the encoding of the corresponding font (whichever is present).
- ▶ Hypertext strings will be interpreted according to the *hypertextencoding* parameter or option.
- ▶ Name strings will be interpreted according to the *hypertext* settings if *usehypertextencoding=true*, and *host* encoding otherwise.

Note that the characters `{` and `}` require special handling within strings in option lists, and must be preceded by a `\` character if they are used within a string option. This requirement remains for legacy encodings such as Shift-JIS: all occurrences of the byte values *0x7B* and *0x7D* must be preceded with *0x5C*. For this reason the use of UTF-8 for options is recommended (instead of Shift-JIS and other legacy encodings).

4.5 Addressing Characters

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 several auxiliary methods for expressing text.

4.5.1 Escape Sequences

PDFlib supports a method for easily incorporating arbitrary values within text strings via a mechanism called *escape sequences* (this is actually a misnomer; *backslash substitution* might be a better term.). For example, the `\t` sequence in the default text of a text block can be used to include tab characters which may not be possible by direct keyboard input. Similarly, escape sequences are useful for expressing codes for symbolic fonts, or in literal strings for language bindings where escape sequences are not available.

An escape sequence is an instruction to replace a sequence with a single byte value. The sequence starts with the code for the backslash character `'\'` in the current encoding of the string. The byte values resulting from substituting escape sequences are listed in Table 4.7; some differ between ASCII and EBCDIC platforms. Only byte values in the range 0-255 can be expressed with escape sequences.

Unlike some programming languages, escape sequences in PDFlib always have fixed length depending on their type. Therefore no terminating character is required for the sequence.

Table 4.7 Escape sequences for byte values

sequence	length	Mac, Windows, Unix	EBCDIC platforms	common interpretation
<code>\f</code>	2	<code>oC</code>	<code>oC</code>	form feed
<code>\n</code>	2	<code>oA</code>	<code>15/25</code>	line feed
<code>\r</code>	2	<code>oD</code>	<code>oD</code>	carriage return
<code>\t</code>	2	<code>o9</code>	<code>o5</code>	horizontal tabulation
<code>\v</code>	2	<code>oB</code>	<code>oB</code>	line tabulation
<code>\\</code>	2	<code>5C</code>	<code>Eo</code>	backslash
<code>\xNN</code>	4	two hexadecimal digits specifying a byte value, e.g. <code>\xFF</code>		
<code>\NNN</code>	4	three octal digits specifying a byte value, e.g. <code>\377</code>		

Escape sequences will not be substituted by default; you must explicitly set the *escapesequence* parameter or option to *true* if you want to use escape sequences for strings:

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

Cookbook A full code sample can be found in the Cookbook topic `fonts/escape_sequences`.

Escape sequences will be evaluated in all content strings, hypertext strings, and name strings after BOM detection, but before converting to the target format. If *textformat=*

utf16, *utf16le* or *utf16be* escape sequences must be expressed as two byte values according to the selected format. Each character in the escape sequence will be represented by two bytes, where one byte has the value zero. If *textformat=utf8* the resulting code will not be converted to UTF-8.

If an escape sequence cannot be resolved (e.g. *\x* followed by invalid hex digits) an exception will be thrown. For content strings the behavior is controlled by the *glyph-check* and *errorpolicy* settings.

Be careful with Windows path names containing backslash characters when escape sequence are enabled.

4.5.2 Character References

Cookbook A full code sample can be found in the *Cookbook* topic `fonts/character_references`.

A character reference is an instruction to replace the reference sequence with a Unicode value. The reference sequence starts with the code of the ampersand character `'&'` in the current encoding, and ends with the code of the semicolon character `'–'`. There are several methods available for expressing the target Unicode values:

HTML character references. PDFlib supports all character entity references defined in HTML 4.0. Numeric character references can be supplied in decimal or hexadecimal notation. The full list of HTML character references can be found at the following location:

www.w3.org/TR/REC-html40/charset.html#h-5.3

Examples:

<code>&shy;</code>	U+00AD soft hyphen
<code>&euro;</code>	U+20AC Euro glyph (entity name)
<code>&lt;</code>	U+003C less than sign
<code>&gt;</code>	U+003E greater than sign
<code>&amp;</code>	U+0026 ampersand sign
<code>&Alpha;</code>	U+0391 Greek Alpha

Numerical character references. Numerical character references for Unicode characters are also defined in HTML 4.0. They require the hash character `'#'` and a decimal or hexadecimal number, where hexadecimal numbers are introduced with a lower- or uppercase `'X'` character. Examples:

<code>&#173;</code>	U+00AD soft hyphen
<code>&#xAD;</code>	U+00AD soft hyphen
<code>&#229;</code>	U+0229 letter a with small circle above (decimal)
<code>&#xE5;</code>	U+00E5 letter a with small circle above (hexadecimal)
<code>&#Xe5;</code>	U+00E5 letter a with small circle above (hexadecimal)
<code>&#x20AC;</code>	U+20AC Euro glyph (hexadecimal)
<code>&#8364;</code>	U+20AC Euro glyph (decimal)

Note Code points 128-159 (decimal) or 0x80-0x9F (hexadecimal) do not reference winansi code points. In Unicode they do not refer to printable characters, but control characters.

PDFlib-specific entity names. PDFlib supports custom character entity references for the following groups of Unicode control characters:

- ▶ Control characters for overriding the default shaping behavior listed in Table 6.4.
- ▶ Control characters for overriding the default bidi formatting listed in Table 6.5.

- ▶ Control characters for Textflow line breaking and formatting listed in Table 8.1.

Examples:

<code>&linefeed;</code>	U+000A linefeed control character
<code>&hortab;</code>	U+0009 horizontal tab
<code>&ZWNJ;</code>	U+200C ZERO WIDTH NON-JOINER

Glyph name references. Glyph names are drawn from the following sources:

- ▶ Common glyph names will be searched in an internal list
- ▶ Font-specific glyph names are searched in the current font. Character references of this class work only with content strings since they always require a font.

In order to identify glyph name references the actual name requires a period character '.' after the ampersand character '&'. Examples:

<code>&.three;</code>	U+0033 common glyph name for the digit 3
<code>&.mapleleaf;</code>	(PUA unicode value) custom glyph name from Carta font
<code>&.T.swash;</code>	(PUA unicode value) second period character is part of the glyph name

Character references with glyph names are useful in the following scenarios:

- ▶ Character references with font-specific glyph names are useful in contents strings to select alternate character forms (e.g. swash characters) and glyphs without any specific Unicode semantics (symbols, icons, and ornaments). Note that tabular figures and many other features are more easily implemented with OpenType features (see Section 6.3, »OpenType Layout Features«, page 152).
- ▶ Names from the Adobe Glyph List (including the *uniXXXX* and *u1XXXX* forms) plus certain common »misnamed« glyph names will always be accepted for content strings and hypertext strings.

Byte value references. Numerical values can also be supplied in character references which may be useful for addressing the glyphs in a symbol font. This variant requires an additional hash character '#' and a decimal or hexadecimal number, where hexadecimal numbers are introduced with a lower- or uppercase 'X' character. Example (assuming the Wingdings font):

<code>&.#x9F;</code>	bullet symbol in Wingdings font
<code>&.#159;</code>	bullet symbol in Wingdings font

Using character references. Character references will not be substituted by default; you must explicitly set the *charref* parameter or option to *true* in order to use character references in content strings, for example:

```
p.set_parameter("charref", "true");
font = p.load_font("Helvetica", "winansi", "");
if (font == -1) { ... }
p.setfont(font, 24);
p.show_xy("Price: 500&euro;", 50, 500);
```

Additional notes on using character references:

- ▶ Character references can be used in all content strings, hypertext strings, and name strings. As an exception, font-specific glyph name references work only with contents strings as noted above.

- ▶ Character references are not substituted in text with *builtin* encoding. However, you can use character references for symbolic fonts by using *unicode* encoding.
- ▶ Character references are not substituted in option lists, but they will be recognized in options with the *Unichar* data type; in this case the *'&'* and *';* decoration must be omitted. This recognition is always enabled; it is not subject to the *charref* parameter or option.
- ▶ In non-Unicode-aware language bindings character references must be expressed as two-byte values if *textformat=utf16*, *utf16be*, or *utf16le*. If *encoding=unicode* and *textformat=bytes* the character references must be expressed in ASCII (even on EBCDIC-based platforms).
- ▶ If a character reference cannot be resolved (e.g. *&#* followed by invalid decimal digits, or *&* followed by an unknown entity name) an exception will be thrown. For content strings the behavior is controlled by the *glyphcheck* and *errorpolicy* settings. With *glyphcheck=none* the reference sequence itself will appear in the generated output.

5 Font Handling

5.1 Font Formats

5.1.1 TrueType Fonts

TrueType file formats. PDFlib supports vector-based TrueType fonts, but not fonts based on bitmaps. TrueType font files are self-contained: they contain all required information in a single file. PDFlib supports the following file formats for TrueType fonts:

- ▶ Windows TrueType fonts (*.tff), including Western, symbolic, and CJK fonts;
- ▶ TrueType collections (*.ttc) with multiple fonts in a single file. TTC files are typically used for grouping CJK fonts, but Apple also uses them to package multiple members of a Western font family in a single file.
- ▶ End-user defined character (EUDC) fonts (*.tte) created with Microsoft's *eudcedit.exe* tool;
- ▶ On the Mac any TrueType font installed on the system (including .dfont) can also be used in PDFlib.



TrueType font names. If you are working with font files you can use arbitrary alias names (see »Sources of Font Data«, page 126). 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*).

5.1.2 OpenType Fonts

The OpenType font format combines PostScript and TrueType technology. It is implemented as an extension of the TrueType file format and offers a unified format. OpenType fonts may contain optional tables which can be used to enhance text output, e.g. ligatures and swash characters (see Section 6.3, »OpenType Layout Features«, page 152), as well as tables for complex script shaping (see Section 6.4, »Complex Script Output«, page 158).



While OpenType fonts offer a single container format which works on all platforms, it may be useful to understand the following OpenType flavors which sometimes lead to confusion:

- ▶ Outline format: OpenType fonts may contain glyph descriptions which are based on TrueType or PostScript. The PostScript flavor is also called Compact Font Format (CFF) or Type 2, and is usually used with the *.otf suffix. The Windows Explorer always displays OpenType fonts with the »O« logo.
- ▶ TrueType fonts and OpenType fonts with TrueType outlines are not easily distinguished since both may use the *.tff suffix. Because of this blurry distinction the Windows Explorer works with the following criterion: if a .tff font contains a digital signature it is displayed with the »O« logo; otherwise it is displayed with the »TT« logo. However, since a digital signature is not required in OpenType fonts this can-

not be used a reliable criterion for distinguishing plain old TrueType fonts and OpenType fonts.

- ▶ The CID (Character ID) architecture is used for CJK fonts. Modern CID fonts are packaged as OpenType *.otf fonts with PostScript outlines. From a practical standpoint they are indistinguishable from plain OpenType fonts. The Windows Explorer always displays OpenType CID fonts with the »O« logo.

Note that neither the file name suffix nor the logo displayed by the Windows Explorer says anything about the presence or absence of OpenType layout features in a font. See Section 6.3, »OpenType Layout Features«, page 152, for more information.

5.1.3 PostScript Type 1 Fonts

PostScript outline and metrics file formats. PostScript Type 1 fonts are always split in two parts: the actual outline data and the metrics information. PDFlib supports the following file formats for PostScript Type 1 outline and metrics data on all platforms:



- ▶ The platform-independent AFM (Adobe Font Metrics) and the Windows-specific PFM (Printer Font Metrics) format for metrics information.
- ▶ 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, i.e. LWFN (LaserWriter Font) outline fonts, are also supported. These are accompanied by a font suitcase (FOND resource, or FFIL) which contains metrics data (plus screen fonts, which will be ignored by PDFlib).
- ▶ When working with PostScript host fonts the LWFN file must be placed in the same directory as the font suitcase, and must be named according to the 5+3+3 rule.

PostScript font names. If you are working with font files you can use arbitrary alias names (see section »Sources of Font Data«, page 126). If you want to know the font's internal name there are several possibilities to determine it:

- ▶ 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.
- ▶ On Windows and Mac OS X or above you can double-click the font 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).

5.1.4 SING Fonts (Glyphlets)

SING fonts (*Smart Independent Glyphlets*) are technically an extension of the OpenType font format. SING fonts have been developed as a solution to the Gaiji problem with CJK text, i.e. custom glyphs which are not encoded in Unicode or any of the common CJK legacy encodings. For more details on the SING architecture you can download the *Adobe Glyphlet Development Kit (GDK) for SING Gaiji Architecture* at the following location:

www.adobe.com/devnet/opentype/gdk/topic.html

SING fonts usually contain only a single glyph (they may also contain an additional vertical variant). The Unicode value of this »main« glyph can be retrieved with PDFlib by requesting its glyph ID and subsequently the Unicode value for this glyph ID:

```
maingid = (int) p.info_font(font, "maingid", "");  
uv = (int) p.info_font(font, "unicode", "gid=" + maingid);
```

It is recommended to use SING fonts as fallback font with the *gaiji* suboption of the *forcechars* option of the *fallbackfonts* option of *PDF_load_font()*; see Section 6.5.3, »EUDC and SING Fonts for Gaiji Characters«, page 169, for more information.

Cookbook A full code sample can be found in the *Cookbook topic* fonts/starter_fallback.

The low-cost FontLab SigMaker tool can be used to generate SING fonts based on an existing image or glyph from another font:

www.fontlab.com/font-utility/sigmaker/

5.1.5 Type 3 Fonts

Unlike all other font formats, Type 3 fonts are not fetched from a disk file, but must be defined at runtime with standard PDFlib graphics 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.

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. However, Type 3 fonts are most often used for bitmapped glyphs since it is the only font format in PDF which supports raster images for glyphs.

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:

```
p.begin_font("Fuzzyfont", 0.001, 0.0, 0.0, 0.001, 0.0, 0.0, "");  
  
p.begin_glyph("circle", 1000, 0, 0, 1000, 1000);  
p.arc(500, 500, 500, 0, 360);  
p.fill();  
p.end_glyph();  
  
p.begin_glyph("ring", 400, 0, 0, 400, 400);  
p.arc(200, 200, 200, 0, 360);  
p.stroke();  
p.end_glyph();  
  
p.end_font();
```

Cookbook Full code samples can be found in the *Cookbook topics* fonts/starter_type3font, fonts/type3_bitmaptext, fonts/type3_rasterlogo, and fonts/type3_vectorlogo.

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 in text output must actually be defined in the font: if character code *x* is to be displayed with any text output function, and the encoding contains *glyphname* at position *x*, then *glyphname* must have been defined via *PDF_begin_glyph()*.
- ▶ Some PDF consumers require a glyph named *.notdef* if codes will be used for which the corresponding glyph names are not defined in the font. Acrobat 8 may even crash if a *.notdef* glyph is not present. 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.
- ▶ Type 3 fonts do not contain any typographic properties such as ascender, descender, etc. However, these can be set by using the corresponding options in *PDF_load_font()*.

5.2 Unicode Characters and Glyphs

5.2.1 Glyph IDs

A font is a collection of glyphs, where each glyph is defined by its geometric outline. PDFlib assigns a number to each glyph in the font. This number is called the glyph id or GID. GID 0 (zero) refers to the *.notdef* glyph in all font formats. The visual appearance of the *.notdef* glyph varies among font formats and vendors; typical implementations are the space glyph or a hollow or crossed-out rectangle. The highest GID is one less than the number of glyphs in the font which can be queried with the *numglyphs* keyword of *PDF_info_font()*.

The assignment of glyph IDs depends on the font format:

- ▶ Since TrueType and OpenType fonts already contain internal GIDs, PDFlib uses these GIDs.
- ▶ For CID-keyed OpenType CJK fonts CIDs will be used as GIDs.
- ▶ For other font types PDFlib numbers the glyphs according to the order of the corresponding outline descriptions in the font.

PDFlib supports glyph selection via GID as an alternative to Unicode and other encodings (see »Glyphid encoding«, page 123). Direct GID addressing is only useful for specialized applications, e.g. printing font overview tables by querying the number of glyphs and iterating over all glyph IDs.

5.2.2 Unicode Mappings for Glyphs

Unicode mappings. PDFlib assigns a unique Unicode value to each GID. This mapping process depends on the font format and is detailed in the sections below for the supported font types. Although a unique Unicode value will be assigned to each GID, the reverse is not necessarily true, i.e. a particular glyph can represent multiple Unicode values. Common examples in many TrueType and OpenType fonts are the empty glyph which represents U+0020 Space as well as U+00A0 No-Break Space, and a glyph which represents both U+2126 Ohm Sign and U+03A9 Greek Capital Letter Omega. If multiple Unicode values point to the same glyph in a font PDFlib will assign the first Unicode value found in the font.

Unmapped glyphs and the Private Use Area (PUA). In some situations the font may not provide a Unicode value for a particular glyph. In this case PDFlib assigns a value from the Unicode Private Use Area (PUA, see Section 4.1, »Important Unicode Concepts«, page 93) to the glyph. Such glyphs are called *unmapped glyphs*. The number of unmapped glyphs in a font can be queried with the *unmappedglyphs* keyword of *PDF_info_font()*. Unmapped glyphs will be represented by the Unicode replacement character U+FFFD in the font's ToUnicode CMap which controls searchability and text extraction. As a consequence, unmapped glyphs cannot be properly extracted as text from the generated PDF.

When PDFlib assigns PUA values to unmapped glyphs it uses ascending values from the following pool:

- ▶ The basis is the Unicode PUA range in the Basic Multilingual Plane (BMP), i.e. the range U+E000 - U+F8FF. Additional PUA values in plane 15 (U+F0000 to U+FFFFFD) are used if required.

- ▶ PUA values which have already been assigned by the font internally are not used when creating new PUA values.
- ▶ PUA values in the Adobe range U+F600-F8FF are not used.

The generated PUA values are unique within a font. The assignment of generated PUA values for the glyphs in a font is independent from other fonts.

Unicode mapping for TrueType, OpenType, and SING fonts. PDFlib keeps the Unicode mappings found in the font's relevant *cmap* table (the selection of the *cmap* depends on the encoding supplied to *PDF_load_font()*). If a single glyph is used for multiple Unicode values PDFlib will use the first Unicode value found in the font.

If the *cmap* does not provide any Unicode mapping for a glyph PDFlib checks the glyph names in the *post* table (if present in the font) and determines Unicode mappings based on the glyph names as described below for Type 1 fonts).

In some cases neither the *cmap* nor the *post* table provide Unicode values for all glyphs in the font. This is true for variant glyphs (e.g. swash characters), extended ligatures, and non-textual symbols outside the Unicode standard. In this case PDFlib assigns PUA values to the affected glyphs as described in »Unmapped glyphs and the Private Use Area (PUA)«, page 115.

Unicode mapping for Type 1 fonts. Type 1 fonts do not include explicit Unicode mappings, but assign a unique name to each glyph. PDFlib tries to assign a Unicode value based on this glyph name, using an internal mapping table which contains Unicode mappings for more than 7 000 common glyph names for a variety of languages and scripts. The mapping table includes ca. 4 200 glyph names from the Adobe Glyph List (AGL)¹. However, Type 1 fonts may contain glyph names which are not included in the internal mapping table; this is especially true for Symbol fonts. In this case PDFlib assigns PUA values to the affected glyphs as described in »Unmapped glyphs and the Private Use Area (PUA)«, page 115.

If the metrics for a Type 1 font are loaded from a PFM file and no PFB or PFA outline file is available, the glyph names of the font are not known to PDFlib. In this case PDFlib assigns Unicode values based on the encoding (charset) entry in the PFM file.

Unicode mapping for Type 3 fonts. Since Type 3 fonts are also based on glyph names, they are treated in the same way as Type 1 fonts. An important difference, however, is that the glyph names for Type 3 fonts are under user control (via the *glyphname* parameter of *PDF_begin_glyph()*). It is therefore strongly recommended to use appropriate glyph names from the AGL for user-defined Type 3 fonts. This makes sure that proper Unicode values will be assigned automatically by PDFlib, resulting in searchable text in the generated PDF documents.

5.2.3 Unicode Control Characters

Control characters are Unicode values which do not represent any glyph, but are used to convey some formatting information. PDFlib processes the following groups of Unicode control characters:

- ▶ The control characters for overriding the default shaping behavior (listed in Table 6.4) and those for overriding the default bidi formatting (listed in Table 6.5) control

1. The AGL can be found at partners.adobe.com/public/developer/en/opentype/glyphlist.txt

complex script shaping and OpenType layout feature processing in Textline and Textflow. After evaluating these control characters they will be removed.

- ▶ The formatting control characters for line breaking and Textflow formatting listed in Table 8.1. After evaluating these control characters they will be removed.
- ▶ Other Unicode control characters in the ranges U+0001-U+0019 and U+007F-U+009F will be replaced with the *replacement* character.

Even if a font contains a glyph for a control character the glyph will usually not be visible since PDFlib removes control characters (as an exception to this rule *&NBSP*; and *&SHY*; will not be removed). However, with *encoding=glyphid* control characters will be retained in the text and can produce visible output.

5.3 The Text Processing Pipeline

The client application provides text for page output to PDFlib. This text is encoded according to some application-specific encoding and format. However, PDFlib's internal processing is based on the Unicode standard, and the final text output requires font-specific glyph IDs. PDFlib therefore treats incoming strings for page contents in a text processing pipeline with three sections:

- ▶ normalize input codes to Unicode values; this process is restricted by the selected encoding.
- ▶ convert Unicode values to font-specific glyph IDs; this process is restricted by the available glyphs in the font.
- ▶ transform glyph IDs; this process is restricted by the output encoding.

These three sections of the text processing pipeline contain several subprocesses which can be controlled by options.

5.3.1 Normalizing Input Strings to Unicode

The following steps are performed for all encodings except *encoding=glyhid* and non-Unicode CMaps:

- ▶ Unicode-aware language bindings: if a single-byte encoding has been specified UTF-16 text is converted to single-byte text by dropping the high-order bytes.
- ▶ Windows: convert multi-byte text (e.g. *cp932*) to Unicode.
- ▶ Replace escape sequences (see Section 4.5.1, »Escape Sequences«, page 107) with the corresponding numerical values.
- ▶ Resolve character references and replace them with the corresponding Unicode values (see Section 4.5.2, »Character References«, page 108, and next section below).
- ▶ Single-byte encodings: convert single-byte text to Unicode according to the specified encoding.

See also Section 5.2.2, »Unicode Mappings for Glyphs«, page 115, for more details regarding the Unicode assignments for various font formats and types of characters.

Character references with glyph names. A font may contain glyphs which are not directly accessible because the corresponding Unicode values are not known in advance (since PDFlib assigns PUA values at runtime). As an alternative for addressing such glyphs, character references with glyph names can be used; see Section 4.5.2, »Character References«, page 108, for a syntax description. These references be replaced with the corresponding Unicode values.

If a character reference is used in a content string, PDFlib tries to find the specified glyph in the current font, and will replace the reference with the glyph's Unicode value. If a glyph with the specified name is not available in the font, PDFlib searches its internal glyph name table to determine a Unicode value. This Unicode value will be used again to check whether a suitable glyph is available in the font. If no such glyph can be found, the behavior is controlled by the *glyphcheck* and *errorpolicy* settings. Character references cannot be used with *glyhid* or *builtin* encoding.

5.3.2 Converting Unicode Values to Glyph IDs

The Unicode values determined in the previous section may have to be modified for several reasons. The steps below are performed for all encodings except *encoding=glyphid* and non-Unicode CMaps which are treated as follows:

- ▶ For non-Unicode CMaps: invalid code sequences always trigger an exception.
- ▶ For *encoding=glyphid*: invalid glyph IDs are replaced with the *replacementchar* (if *glyphcheck=replace*) or glyph ID 0 (*glyphcheck=none*). If *glyphcheck=error* an exception will be thrown.

Forced characters from fallback fonts. Replace Unicode values according to the *forcechars* suboption of the *fallbackfonts* option, and determine the glyph ID of the corresponding fallback font. For more information see Section 5.4.6, »Fallback Fonts«, page 133.

Convert to glyph IDs. Convert the remaining Unicode values to glyph IDs according to the mappings determined in Section 5.2.2, »Unicode Mappings for Glyphs«, page 115. If no corresponding glyph ID for a Unicode value was found in the font, the next steps depend on the *glyphcheck* option:

- ▶ *glyphcheck=none*: glyph ID 0 will be used. This means that the *.notdef* glyph will be used in the text output. If the *.notdef* glyph contains a visible shape (often a hollow or crossed-out rectangle) it makes the problematic characters visible on the PDF page, which may or may not be desired.
- ▶ *glyphcheck=replace* (which is the default): a warning message will be logged and PDFlib attempts to replace the unmapable Unicode value with the glyph replacement mechanism detailed below.
- ▶ *glyphcheck=error*: PDFlib raises an error. In case of *errorpolicy=return* this means that the function call will be terminated without creating any text output; *PDF_add/create_textflow()* will return -1 (in PHP: 0). In case of *errorpolicy=exception* an exception will be thrown.

Glyph replacement. If *glyphcheck=replace*, unmapable Unicode values will recursively be replaced as follows:

- ▶ The fallback fonts specified when loading the master font will be searched for glyphs for the Unicode value. This may involve an arbitrary number of fonts since more than one fallback font can be specified for each font. If a glyph is found in one of the fallback fonts it will be used.
- ▶ Select a typographically similar glyph according to the Unicode value from PDFlib's internal replacement table. The following excerpt from the internal list contains some of these replacements. If the first character in the list is unavailable in a font, it will be replaced with the second character:

U+00A0 (NO-BREAK SPACE)	U+0020 (SPACE)
U+00AD (SOFT HYPHEN)	U+002D (HYPHEN-MINUS)
U+2010 (HYPHEN)	U+002D (HYPHEN-MINUS)
U+03BC (GREEK SMALL LETTER MU)	U+00C5 (MICRO SIGN)
U+212B (ANGSTROM SIGN)	U+00B5 (LATIN CAPITAL LETTER A WITH RING ABOVE Å)
U+220F (N-ARY PRODUCT)	U+03A0 (GREEK CAPITAL LETTER PI)
U+2126 (OHM SIGN)	U+03A9 (GREEK CAPITAL LETTER OMEGA)

In addition to the internal table, the fullwidth characters U+FF01 to U+FF5E will be replaced with the corresponding ISO 8859-1 characters (i.e. U+0021 to U+007E) if the fullwidth variants are not available in the font.

- Decompose Unicode ligatures into their constituent glyphs (e.g. replace U+FB00 *Latin small ligature ff* with the sequence U+0066 *f*, U+0066 *f*).
- Select glyphs with the same Unicode semantics according to their glyph name. In particular, all glyph name suffixes separated with a period will be removed if the corresponding glyph is not available (e.g. replace *A.swash* with *A*; replace *g.alt* with *g*).

If none of these methods delivers a glyph for the Unicode value, the character specified in the *replacementchar* option will be used. If the corresponding glyph itself is not available in the font (or the *replacementchar* option was not specified), U+00A0 (NO-BREAK SPACE) and U+0020 (SPACE) will be tried. If these are still unavailable, glyph ID 0 (the missing glyph symbol) will be used.

For PDF/A-1, PDF/X-4 and PDF/X-5 input codes which map to glyph ID 0 will be skipped.

Cookbook A full code sample can be found in the *Cookbook* topic `fonts/glyph_replacement`.

5.3.3 Transforming Glyph IDs

The determined glyph IDs are not yet final since several transformations may have to be applied before final output can be created. The details of these transformations depend on the font and several options. The steps below will be performed for all encodings except non-Unicode CMaps with *keepnative=true*.

Vertical glyphs. For fonts in vertical writing mode some glyphs may be replaced by their vertical counterparts. This substitution requires a *vert* OpenType layout feature table in the font.

OpenType layout features. OpenType features can create ligatures, swash characters, small caps, and many other typographic variations by replacing one or more glyph IDs with other values. OpenType features are discussed in Section 6.3, »OpenType Layout Features«, page 152. OpenType layout features are relevant only for suitable fonts (see »Requirements for OpenType layout features«, page 154), and are applied according to the *features* option.

Complex script shaping. Shaping reorders the text and determines the appropriate variant glyph according to the position of a character (e.g. initial, middle, final, or isolated form of Arabic characters). Shaping is discussed in Section 6.4, »Complex Script Output«, page 158. It is relevant only for suitable fonts (see »Requirements for shaping«, page 160, and is applied according to the *shaping* option.

5.4 Loading Fonts

5.4.1 Selecting an Encoding for Text Fonts

Fonts can be loaded explicitly with the `PDF_load_font()` function or implicitly by supplying the `fontname` and `encoding` options to certain functions such as `PDF_add/create_textflow()` or `PDF_fill_textblock()`. Regardless of the method used for loading a font, a suitable encoding must be specified. The encoding determines

- ▶ in which text formats PDFlib expects the supplied text;
- ▶ which glyphs in a font can be used;
- ▶ how text on the page and the glyph data in the font will be stored in the PDF output document.

PDFlib's text handling is based on the Unicode standard¹, almost identical to ISO 10646. Since most modern development environments support the Unicode standard our goal is to make it as easy as possible to use Unicode strings for creating PDF output. However, developers who don't work with Unicode are not required to switch their application to Unicode since legacy encodings can be used as well.

The choice of encoding depends on the font, the available text data, and some programming aspects. In the remainder of this section we will provide an overview of the different classes of encodings as an aid for selecting a suitable encoding.

Unicode encoding. With `encoding=unicode` you can pass Unicode strings to PDFlib. This encoding is supported for all font formats. Depending on the language binding in use, the Unicode string data type provided by the programming language (e.g. Java) can be used, or byte arrays containing Unicode in one of the UTF-8, UTF-16, or UTF-32 formats with little- or big-endian byte ordering (e.g. C).

With `encoding=unicode` all glyphs in a font can be addressed; complex script shaping and OpenType layout features are supported. PDFlib checks whether the font contains a glyph for a requested Unicode value. If no glyph is available, a substitute glyph can be pulled from the same or another font (see Section 5.4.6, »Fallback Fonts«, page 133).

In non-Unicode-aware language bindings PDFlib expects UTF-16 encoded text by default. However, you can supply single-byte strings by specifying `textformat=bytes`. In this case the byte values represent the characters U+0001 - U+00FF, i.e. the first Unicode block with Basic Latin characters (identical to ISO 8859-1). However, using character references Unicode values outside this range can also be specified in single-byte text.

Some font types in PDF (Type 1, Type 3, and OpenType fonts based on glyph names) support only single-byte text. However, PDFlib takes care of this situation to make sure that more than 255 different characters can be used even for these font types.

The disadvantage of `encoding=unicode` is that text in traditional single- or multi-byte encodings (except ISO 8859-1) cannot be used.

Single-byte 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 255 different characters at a time (the value 0 is not available). This type of encoding is supported for all font formats. PDFlib checks whether the font contains glyphs which match the selected encoding. If a minimum number of usable glyphs is not reached, PDFlib will log a warning message. If no usable glyph at all for the selected encoding is available in the font, font

¹. See www.unicode.org

loading will fail with the message *font doesn't support encoding*. PDFlib checks whether the font contains a glyph for a requested input value. If no glyph is available, a substitute glyph can be pulled from the same or another font (see Section 5.4.6, »Fallback Fonts«, page 133).

In non-Unicode-aware language bindings PDFlib expects single-byte encoded text by default. However, you can supply UTF-8 or UTF-16 strings by specifying *textformat= utf8* or *utf16*.

8-bit encodings are discussed in detail in Section 4.2, »Single-Byte (8-Bit) Encodings«, page 95. They can be pulled 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 a *system encoding*. This feature is available on Windows, IBM iSeries and zSeries.

The disadvantage of single-byte encodings is that only a limited set of characters and glyphs is available. For this reason complex script shaping and OpenType layout features are not supported for single-byte encodings.

Builtin encoding. Among other scenarios, you can specify *encoding=builtin* to use single-byte codes for non-textual glyphs from symbolic fonts. The format of a font's internal encoding depends on the font type:

- ▶ TrueType: the encoding is created based on the font's symbolic cmap, i.e. the (3, 0) entry in the cmap table.
- ▶ OpenType fonts can contain an encoding in the CFF table.
- ▶ PostScript Type 1 fonts always contain an encoding.
- ▶ For Type 3 fonts the encoding is defined by the first 255 glyphs in the font.

If the font does not contain any builtin encoding font loading fails (e.g. OpenType CJK fonts). You can use the *symbolfont* key in *PDF_info_font()*. If it returns *false*, the font is a text font which can also be loaded with one of the common single-byte encodings. This is not possible if the *symbolfont* key returns *true*. The glyphs in such symbolic fonts can only be used if you know the corresponding code for each glyph (see Section 5.4.2, »Selecting an Encoding for symbolic Fonts«, page 123).

In non-Unicode-aware language bindings PDFlib expects single-byte formatted text by default. This has the advantage that you can use the single-byte values which have traditionally been used to address some symbolic fonts; this is not possible with other encodings. However, you can also supply text in a Unicode format, e.g. with *textformat=utf16*.

The disadvantage of *encoding=builtin* is that in single-byte encoded text character references cannot be used.

Multi-byte encodings. This encoding type is supported for CJK fonts, i.e. TrueType and OpenType CID fonts with Chinese, Japanese, or Korean characters. A variety of encoding schemes has been developed for use with these scripts, e.g. Shift-JIS and EUC for Japanese, GB and Big5 for Chinese, and KSC for Korean. Multi-byte encodings are defined by

the Adobe CMaps or Windows codepages (see Section 4.3, »Chinese, Japanese, and Korean Encodings«, page 99).

These traditional encodings are only supported in non-Unicode-aware language bindings with the exception of Unicode CMaps which are equivalent to *encoding=unicode*.

In non-Unicode-aware language bindings PDFlib expects multi-byte encoded text by default (*textformat=bytes*).

With multi-byte encodings the text will be written to the PDF output exactly as supplied by the user if the *keepnative* option is *true*.

The disadvantage of multi-byte encodings is that PDFlib checks the input text only for valid syntax, but does not check whether a glyph for the supplied text is available in the font. Also, it is not possible to supply Unicode text since PDFlib cannot convert the Unicode values to the corresponding multi-byte sequences. Finally, character references, OpenType layout features and complex script shaping cannot be used.

Glyphid encoding. PDFlib supports *encoding=glyphid* for all font formats. With this encoding all glyphs in a font can be addressed, using the numbering scheme explained in Section 5.2.1, »Glyph IDs«, page 115. Numerical glyph IDs run from 0 to a theoretical maximum value of 65 535 (but fonts with such a large number of glyphs are not available). The maximum glyph ID value can be queried with the *maxcode* key in *PDF_info_font()*.

In non-Unicode-aware language bindings PDFlib expects double-byte encoded text by default (*textformat=utf16*).

PDFlib checks whether the supplied glyph ID is valid for the font. Complex script shaping and OpenType layout features are supported.

Since glyph IDs are specific to a particular font and in some situations are even created by PDFlib *encoding=glyphid* is generally not suited for regular text output. The main use of this encoding is for printing complete font tables with all glyphs.

5.4.2 Selecting an Encoding for symbolic Fonts

Symbolic fonts are fonts which contain symbols, logos, pictograms or other non-textual glyphs. They raise several issues which are not relevant for text fonts. The underlying problem is that by design the Unicode standard does not generally encode symbolic glyphs (although there are exceptions to this rule, e.g. the glyphs in the common ZapfDingbats font). In order to make symbolic fonts fit for use in Unicode-oriented workflows, TrueType and OpenType fonts usually assign Unicode values in the Private Use Area (PUA) to their glyphs. For lack of Unicode mapping tables, PostScript Type 1 cannot do this, and generally use the codes of Latin characters to select their glyphs. In all font formats the symbolic glyphs usually have custom glyph names.

This situation has the following consequences regarding the selection of glyphs from symbolic fonts:

- ▶ Symbolic TrueType and OpenType fonts are best loaded with *encoding=unicode*. If you know the PUA values assigned to the glyphs you can supply these values in the text in order to select symbolic glyphs. This requires advance knowledge of the PUA assignments in the font.
- ▶ Since PDFlib assigns PUA values for symbolic PostScript Type 1 fonts internally, these PUA values are not known in advance.
- ▶ If you prefer to work with 8-bit codes for addressing the glyphs in a symbolic font you can load the font with *encoding=builtin* and supply the 8-bit codes in the text. For

example, the digit 4 (code 0x34) will select the check mark symbol in the ZapfDingbats font.

In order to use symbolic fonts with *encoding=unicode* suitable Unicode values must be used for the text:

- ▶ The characters in the *Symbol* font all have proper Unicode values.
- ▶ The characters in the *ZapfDingbats* font have Unicode values in the range U+2007 - U+27BF.
- ▶ Microsoft's symbolic fonts, e.g. Wingdings and Webdings, use PUA Unicode values in the range U+FO20 - U+FOFF (although the *charmap* application presents them with single-byte codes).
- ▶ For other fonts the Unicode values for individual glyphs in the font must be known in advance or must be determined at runtime with *PDF_info_font()*, e.g. for PostScript Type 1 fonts by supplying the glyph name.

Control characters. The Unicode control characters in the range U+0001 - U+001F which are listed in Table 8.1 are supported in Textflow even with *encoding=builtin*. Codes < 0x20 will be interpreted as control characters if the symbolic font does not contain any glyph for the code. This is true for the majority of symbolic fonts.

Since the code for the a linefeed characters differs between ASCII and EBCDIC it is recommended to avoid the literal character 0x0A on EBCDIC systems, and use the PDFlib escape sequence `\n` with the option *escapesequence=true* instead. Note that the `\n` must arrive at the PDFlib API, e.g. in C the sequence `\\n` is required.

Character references. Character references are supported for symbolic fonts. However, symbolic fonts generally do not include any glyph for the ampersand character U+0026 '&' which introduces character references. The code 0x26 cannot be used either since it could be mapped to an existing glyph in the font. For these reasons symbolic fonts should be loaded with *encoding=unicode* if character references must be used. Character references do not work with *encoding=builtin*.

5.4.3 Example: Selecting a Glyph from the Wingdings Symbol Font

Since there are many different ways of selecting characters from a symbol font and some will not result in the desired output, let's take a look at an example.

Understanding the characters in the font. First let's collect some information about the target character in the font, using the Windows *charmap* application (see Figure 5.1):

- ▶ *Charmap* displays the glyphs in the Wingdings font, but does not provide any Unicode access in the *Advanced view*. This is a result of the fact that the font contains symbolic glyphs for which no standardized Unicode values are registered. Instead, the glyphs in the font use dummy Unicode values in the Private Use Area (PUA). The *charmap* application does not reveal these values.
- ▶ If you look at the lower left corner of the *charmap* window or hover the mouse over the *smileface* character, the *Character code: 0x4A* is displayed. This is the glyph's byte code.

This code corresponds to the uppercase *J* character in the Winansi encoding. For example, if you copy the character to the clipboard the corresponding Unicode value U+004A, i.e. character *J* will result from pasting the clipboard contents to a text-only

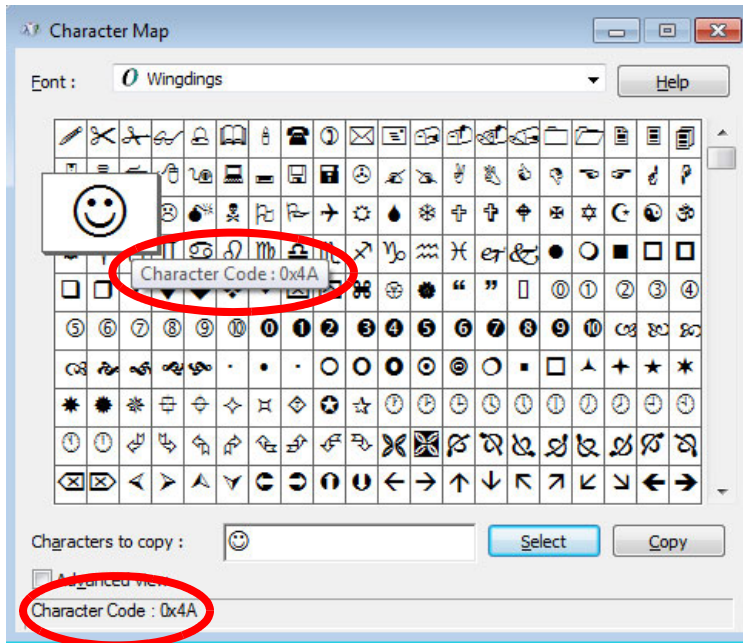


Fig. 5.1
Windows character map
with the Wingdings font

application. Nevertheless, this is not the character's Unicode value and therefore U+004A or *J* can not be used to select it in Unicode workflows.

- The Unicode character used internally in the font is not displayed in *charmap*. However, symbolic fonts provided by Microsoft use the following simple rule:

Unicode value = U+F000 + (character code displayed in charmap)

For the *smileface* glyph this yields the Unicode value U+F04A.

- The corresponding glyph name can be retrieved with a font editor and similar tools. In our example it is *smileface*.

You can use *PDF_info_font()* to query Unicode values, glyph names or codes, see Section 5.6.2, »Font-specific Encoding, Unicode, and Glyph Name Queries«, page 141.

Addressing the symbol character with PDFlib. Depending on the information which is available about the target character you can select the Wingdings *smileface* glyph in several ways:

- If you know the PUA Unicode value which is assigned to the character in the font you can use a numerical character reference (see »Numerical character references«, page 108):

```
&#xF04A;
```

If you work with *textformat=utf8* you can use the corresponding three-byte UTF-8 sequence:

```
\xEF\x81\x8A
```

Unicode values can not be used with the combination of *encoding=builtin* and *textformat=bytes*.

- If you know the character code you can use a byte value reference (see »Byte value references«, page 109):

&.#x4A;

In non-Unicode-aware language bindings the character code can be specified directly if *encoding=builtin* and *textformat=bytes*:

J
\x4A

- If you know the glyph name you can use a glyph name reference (see »Glyph name references«, page 109):

&.smileface;

Glyph names can not be used with the combination of *encoding=builtin* and *text-format=bytes*.

Table 5.1 lists methods for Unicode-aware language bindings such as Java and .NET.

Table 5.1 Addressing the smileface glyph in the Wingdings font with Unicode-aware language bindings (e.g. Java)

encoding	additional options	input string	visible result on the page
unicode		\uFo4A ¹	☺
	charref	o4A;	☺
	charref	&.#x4A;	☺
	charref	&.smileface;	☺
		J ²	(nothing)
	escapesequence	\x4A	(nothing)
builtin	(same as above with encoding=unicode)		

1. String syntax for U+F04A in Java and many other Unicode-aware languages
2. Winansi character for the byte code \x4A

Table 5.2 lists methods for non-Unicode-aware language bindings such as C.

5.4.4 Searching for Fonts

Sources of Font Data. As mentioned earlier, fonts can be loaded explicitly with the *PDF_load_font()* function or implicitly by supplying the *fontname* and *encoding* options to various text output functions. You can use a font’s native name or work with arbitrary custom names which will be used to locate the font data. Custom font names must be unique within a document. In *PDF_info_font()* this font name can be queried with the *apiname* key.

Subsequent calls to *PDF_load_font()* will return the same font handle if all options are identical to those provided in the first call to this function (a few options are treated differently; see PDFlib API Reference for details). Otherwise a new font handle will be created for the same font name. PDFlib supports the following sources of font data:

- Disk-based or virtual font files
- Fonts pulled from the Windows or Mac operating system (host fonts)

- ▶ The font name matches the name in a *FontAFM* or *FontPFM* resource which connects the font name with the name of a PostScript Type 1 font metrics file.
- ▶ The font name matches the name in a *HostFont* resource which connects the font name with the name of a font installed on the system.
- ▶ The font name matches the name of a Latin core font (see »Latin core fonts«, page 129).
- ▶ The name matches the name of a host font installed on the system (see Section 5.4.5, »Host Fonts on Windows and Mac OS X«, page 131).
- ▶ The font name matches the base name (i.e. without file name suffix) of a font file.

If no font was found, font loading stops with the following error message:

Font file (AFM, PFM, TTF, OTF etc.) or host font not found

Details regarding the resource categories can be found in Section 3.1.3, »Resource Configuration and File Search«, page 56. The following sections discuss font loading for the various classes of fonts in more detail.

TrueType and OpenType fonts. The font name must be connected to the name of the desired font file via the *FontOutline* resource:

```
p.set_parameter("FontOutline", "Arial=/usr/fonts/Arial.ttf");
font = p.load_font("Arial", "unicode", "embedding");
```



The alias font name to the left of the equal sign can be chosen arbitrarily:

```
p.set_parameter("FontOutline", "f1=/usr/fonts/Arial.ttf");
font = p.load_font("f1", "unicode", "embedding");
```

As an alternative to runtime configuration via *PDF_set_parameter()*, the *FontOutline* resource can be configured in a UPR file (see Section 3.1.3, »Resource Configuration and File Search«, page 56). In order to avoid absolute file names you can use the *SearchPath* resource category (again, the *SearchPath* resource category can alternatively be configured in a UPR file), for example:

```
p.set_parameter("SearchPath", "/usr/fonts");
p.set_parameter("FontOutline", "f1=Arial.ttf");
font = p.load_font("f1", "unicode", "");
```

TrueType Collections. In order to select a font which is contained in a TrueType Collection (TTC, see Section 6.5.2, »Custom CJK Fonts«, page 168) file you directly specify the name of the font:

```
p.set_parameter("FontOutline", "MS-Gothic=msgothic.ttc");
font = p.load_font("MS-Gothic", "unicode", "embedding");
```



The font name will be matched against the names of all fonts in the TTC file. Alternatively, to select the *n*-th font in a TTC file you can specify the number *n* with a colon after the font name. In this case the alias font name to the left of the equal sign can be chosen arbitrarily:

```
p.set_parameter("FontOutline", "f1=msgothic.ttc");
font = p.load_font("f1:0", "unicode", "");
```


PostScript Type 1 fonts. The font name must be connected to the name of the desired font metrics file via one of the *FontAFM* or *FontPFM* resource categories according to the type of the metrics file:



```
p.set_parameter("FontPFM", "lucidux=LuciduxSans.pfm");
font = p.load_font("lucidux", "unicode", "");
```

If *embedding* is requested for a PostScript font, its name must be connected to the corresponding font outline file (PFA or PFB) via the *FontOutline* resource category:

```
p.set_parameter("FontPFM", "lucidux=LuciduxSans.pfm");
p.set_parameter("FontOutline", "lucidux=LuciduxSans.pfa");
font = p.load_font("lucidux", "unicode", "embedding");
```

Keep in mind that for PostScript Type 1 fonts the *FontOutline* resource alone is not sufficient. Since a metrics file is always required an AFM or PFM file must be available in order to load the font.

The directories which will be searched for font metrics and outline files can be specified via the *SearchPath* resource category.

Latin 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 PDFlib so that no additional data are required (unless the font is to be embedded). The core fonts have the following names:

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

If a font name is not connected to any file name via resources, PDFlib will search the font in the list of Latin core fonts. This step will be skipped if the *embedding* option is specified or a *FontOutline* resource is available for the font name. The following code fragment requests one of the core fonts without any configuration:

```
font = p.load_font("Times-Roman", "unicode", "");
```

Core fonts found in the internal list are never embedded. In order to embed one of these fonts you must configure a font outline file.

Host fonts. If a font name is not connected to any file name via resources, PDFlib will search the font in the list of fonts installed on the Windows or Mac system. Fonts installed on the system are called *host fonts*. Host font names must be encoded in ASCII. On Windows Unicode can also be used. See Section 5.4.5, »Host Fonts on Windows and Mac OS X«, page 131, for more details on host fonts. Example:

```
font = p.load_font("Verdana", "unicode", "");
```

On Windows an optional font style can be added to the font name after a comma:

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

In order to load a host font with the name of one of the core fonts, the font name must be connected to the desired host font name via the *HostFont* resource category. The fol-

lowing fragment makes sure that instead of using the built-in core font data, the Symbol font metrics and outline data will be taken from the host system:

```
p.set_parameter("HostFont", "Symbol=Symbol");  
font = p.load_font("Symbol", "unicode", "embedding");
```

The alias font name to the left of the equal sign can be chosen arbitrarily; we simply used the name of the host font.

Extension-based search for font files. All font types except Type 3 fonts can be searched by using the specified font name as the base name (without any file suffix) of a font metrics or outline file. If PDFlib couldn't find any font with the specified name it will loop over all entries in the *SearchPath* resource category, and add all known file name suffixes to the supplied font name in an attempt to locate the font metrics or outline data. The details of the extension-based search algorithm are as follows:

- ▶ The following suffixes will be added to the font name, and the resulting file names tried one after the other to locate the font metrics (and outline in the case of TrueType and OpenType fonts):

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

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

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

If no font file was found, font loading stops with the following error message:

```
Font cannot be embedded (PFA or PFB font file not found)
```

- ▶ All candidate 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.

The following groups of statements will achieve the same effect with respect to locating the font outline file:

```
p.set_parameter("FontOutline", "Arial=/usr/fonts/Arial.ttf");  
font = p.load_font("Arial", "unicode", "");
```

and

```
p.set_parameter("SearchPath", "/usr/fonts");  
font = p.load_font("Arial", "unicode", "");
```

Standard CJK fonts. Acrobat supports various standard fonts for CJK text; see Section 6.5.1, »Standard CJK Fonts«, page 166, for more details and a list of font names. PDFlib will find a standard CJK font at the very beginning of the font search process if all of the following requirements are met:

- ▶ The specified font name matches the name of a standard CJK font;
- ▶ The specified encoding is the name of one of the predefined CMaps;

- The *embedding* option was not specified.

If one or more of these requirements is violated the font search will be continued. Standard CJK fonts found in the internal list are never embedded. In order to embed one of these you must configure a font outline file. Example:

```
font = p.load_font("KozGoPro-Medium", "90msp-RKSJ-H", "");
```

Type 3 fonts. Type 3 fonts must be defined at runtime by defining the glyphs with standard PDFlib graphics functions (see Section 5.1.5, »Type 3 Fonts«, page 113). If the font name supplied to *PDF_begin_font()* matches the font name requested with *PDF_load_font()* the font will be selected at the beginning of the font search (assuming that the font name didn't match the name of a standard CJK font). Example:

```
p.begin_font("PDFlibLogoFont", 0.001, 0.0, 0.0, 0.001, 0.0, 0.0, "");
```

```
...
```

```
p.end_font();
```

```
...
```

```
font = p.load_font("PDFlibLogoFont", "logoencoding", "");
```

5.4.5 Host Fonts on Windows and Mac OS X

On Mac and Windows systems PDFlib can access TrueType, OpenType, and PostScript fonts which have been installed in the operating system. We refer to such fonts as *host fonts*. Instead of manually configuring font files simply install the font in the system (usually by dropping it into the appropriate directory), and PDFlib will happily use it.

When working with host fonts it is important to use the exact (case-sensitive) font name. Since font names are crucial we mention some platform-specific methods for determining font names below. More information on font names can be found in Section 5.1.3, »PostScript Type 1 Fonts«, page 112, and Section 5.1.1, »TrueType Fonts«, page 111.

Finding host font names on Windows. You can find the name of an installed font by double-clicking the font file and taking note of the full font name which is displayed in the window title (on Windows Vista/7/8) or the first line of the resulting window (on Windows XP). 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 host font data from the Windows system you must use the translated form of the font name in PDFlib (e.g. *Arial Fett*), 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 (e.g. *Arial Bold*).

Note You can avoid this internationalization problem by appending font style names (e.g. »,Bold«, see below) to the font name instead of using localized font name variants.

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

1. See www.microsoft.com/typography/TrueTypeProperty21.mspx

Windows font style names. When loading 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, for example

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

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. Font style names only work with host fonts, but not for fonts configured via a font file.

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

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

The keywords are case-insensitive. The *italic* keyword can be specified alternatively or in addition to the above. If two style names are used both must be separated with a comma, for example:

```
font = p.load_font("Verdana,Bold,Italic", "unicode", "");
```

Numerical font weight values can be used as an equivalent alternative to font style names:

0 (none), 100 (thin), 200 (extralight), 300 (light), 400 (normal), 500 (medium), 600 (semibold), 700 (bold), 800 (extrabold), 900 (black)

The following example will select the bold variant of a font:

```
font = p.load_font("Verdana,700", "unicode", "");
```

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.

Note Do not confuse the Windows style name convention with the fontstyle option which looks similar, but works on a completely different basis.

Potential problem with host font access on Windows. We'd like to alert users to a potential problem with font installation on Windows. If you install fonts via the *File, Install new font...* menu item (as opposed to dragging fonts to the *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.

Host font names on the Mac. Using the *Font Book* utility, which is part of Mac OS X, you can find the names of installed host fonts. In order to programmatically create lists of host fonts we recommend Apple's freely available Font Tools¹. This suite of command-line utilities contains a program called *ftxinstalledfonts* which is useful for determining the exact names of all installed fonts. PDFlib supports several flavors of host font names:

- ▶ QuickDraw font names: these are old-style font names which have been in use for a long time on Mac OS systems, but are considered outdated. In order to determine QuickDraw font names issue the following command in a terminal window:

```
ftxinstalledfonts -q
```

- ▶ »Unique« font names: these are newer font names (in Mac OS supported by new ATS font functions) which can be encoded in Unicode, e.g. for East-Asian fonts. In order to determine unique font names issue the following command in a terminal window (in some cases the output contains entries with a '.' which must be removed):

```
ftxinstalledfonts -u
```

- ▶ PostScript font names. In order to determine PostScript font names issue the following command in a terminal window:

```
ftxinstalledfonts -p
```

Note The Leopard builds of PDFlib (for Mac OS X 10.5 and above) support all three kinds of host font names. Non-Leopard builds accept only QuickDraw font names.

Potential problems with host font access on the Mac. In our testing we found that newly installed fonts are sometimes not accessible for UI-less applications such as PDFlib until the user logs out from the console, and logs in again.

On Mac OS X 10.5 (Leopard) host fonts are not available to programs running in a terminal session from a remote computer. This is not a restriction of PDFlib, but also affects other programs such as Font Tools. This problem has been fixed in Mac OS X 10.5.6.

5.4.6 Fallback Fonts

Cookbook A full code sample can be found in the *Cookbook* topic `text_output/starter_fallback`.

Fallback fonts provide a powerful mechanism which deals with shortcomings in fonts and encodings. It can be used in many situations to facilitate text output since necessary font changes will be applied automatically by PDFlib. This mechanism augments a given font (called the base font) by merging glyphs from one or more other fonts into the base font. More precisely: the fonts are not actually modified, but PDFlib applies all necessary font changes in the PDF page description automatically. Fallback fonts offer the following features:

- ▶ Glyphs which are unavailable in the base font will automatically be searched in one or more fallback fonts. In other words, you can add glyphs to a font. Since multiple fallback fonts can be attached to the base font, you can effectively use all Unicode characters for which at least one of the fonts contains a suitable glyph.

¹. See developer.apple.com/textfonts/download

- ▶ Glyphs from a particular fallback font can be used to override glyphs in the base font, i.e. you can replace glyphs in a font. You can replace one or more individual glyphs, or specify one or more ranges of Unicode characters which will be replaced.

The size and vertical position of glyphs from a fallback font can be adjusted to match the base font. Somewhat surprisingly, the base font itself can also be used as a fallback font (with the same or a different encoding). This can be used to implement the following tricks:

- ▶ Using the base font itself as fallback font can be used to adjust the size or position of some or all glyphs in a font.
- ▶ You can add characters outside the actual encoding of the base font.

The fallback font mechanism is controlled by the *fallbackfonts* font loading option, and affects all text output functions. As with all font loading options, the *fallbackfonts* option can be provided in explicit calls to *PDF_load_font()*, or in option lists for implicit font loading. Since more than one fallback font can be specified for a base font, the *fallbackfonts* option expects a list of option lists (i.e. an extra set of braces is required).

PDF_info_font() can be used to query the results of the fallback font mechanism (see Section 5.6.3, »Querying Codepage Coverage and Fallback Fonts«, page 142).

Caveats. Note the following when working with fallback fonts:

- ▶ Not all font combinations result in typographically pleasing results. Care should be taken to use only fallback fonts in which the glyph design matches the glyph design of the base font.
- ▶ Font loading options for the fallback font(s) must be specified separately in the *fallbackfonts* option list. For example, if *embedding* is specified for the base font, the fallback fonts will not automatically be embedded.
- ▶ Fallback fonts work only if the fonts contain proper Unicode information. The replacement glyph(s) must have the same Unicode value as the replaced glyph.
- ▶ Script-specific shaping (options *shaping*, *script*, *locale*) and OpenType features (options *features*, *script*, *language*) will only be applied to glyphs within the same font, but not across glyphs from the base font and one or more fallback fonts.
- ▶ The underline/overline/strikeout features must be used with care when working with fallback fonts, as well as the *ascender* and similar typographic values. The underline thickness or position defined in the base font may not match the values in the fallback font. As a result, the underline position or thickness may jump in unpleasant ways. A simple workaround against such artefacts is to specify a unified value with the *underlineposition* and *underlinewidth* options of *PDF_fit_textline()* and *PDF_add/create_textflow()*. This value should be selected so that it works with the base font and all fallback fonts.

In the sections below we describe several important use cases of fallback fonts and demonstrate the corresponding option lists.

Add mathematical characters to a text font. As a very rough solution in case of missing mathematical glyphs you can use the following font loading option list for the *fallbackfonts* option to add mathematical glyphs from the Symbol font to a text font:

```
fallbackfonts={{fontname=Symbol encoding=unicode}}
```

Combine fonts for use with multiple scripts. In some situations the script of incoming text data is not known in advance. For example, a database may contain Latin, Greek, and Cyrillic text, but the available fonts cover only one of these scripts at a time. Instead of determining the script and selecting an appropriate font you can construct a font which chains together several fonts, effectively covering the superset of all scripts. Use the following font loading option list for the *fallbackfonts* option to add Greek and Cyrillic fonts to a Latin font:

```
fallbackfonts={
  {fontname=Times-Greek encoding=unicode embedding forcechars={U+0391-U+03F5}}
  {fontname=Times-Cyrillic encoding=unicode embedding forcechars={U+0401-U+0490}}
}
```

Extend 8-bit encodings. If your input data is restricted to a legacy 8-bit encoding you can nevertheless use characters outside this encoding, taking advantage of fallback fonts (where the base font itself serves as a fallback font) and PDFlib's character reference mechanism to address characters outside the encoding. Assuming you loaded the Helvetica font with *encoding=iso8859-1* (this encoding does not include the Euro glyph), you can use the following font loading option list for the *fallbackfonts* option to add the Euro glyph to the font:

```
fallbackfonts={{fontname=Helvetica encoding=unicode forcechars=euro}}
```

Since the input encoding does not include the Euro character you cannot address it with an 8-bit value. In order to work around this restriction use character or glyph name references, e.g. *&euro*; (see Section 4.5.2, »Character References«, page 108).

Use Euro glyph from another font. In a slightly different scenario the base font doesn't include a Euro glyph. Use the following font loading option list for the *fallbackfonts* option to pull the Euro glyph from another font:

```
fallbackfonts={{fontname=Helvetica encoding=unicode forcechars=euro textrise=-5}}}
```

We used the *textrise* suboption to slightly move down the Euro glyph.

Enlarge some or all glyphs in a font. Fallback fonts can also be used to enlarge some or all glyphs in a font without changing the font size. Again, the base font itself will be used as fallback font. This feature can be useful to make different font designs visually compatible without adjusting the *fontsize* in the code. Use the following font loading option list for the *fallbackfonts* option to enlarge all glyphs in the specified range to 120%:

```
fallbackfonts={
  {fontname=Times-Italic encoding=unicode forcechars={U+0020-U+00FF} fontsize=120%}
}
```

Add an enlarged pictogram. Use the following font loading option list for the *fallbackfonts* option to pull a symbol from the ZapfDingbats font:

```
fallbackfonts={
  {fontname=ZapfDingbats encoding=unicode forcechars=.a12 fontsize=150% textrise=-15%}
}
```

Again, we use the `fontsize` and `textrise` suboptions to adjust the symbol's size and position to the base font.

Replace glyphs in a CJK font. You can use the following font loading option list for the *fallbackfonts* option to replace the Latin characters in the ASCII range with those from another font:

```
fallbackfonts={  
  {fontname=Courier-Bold encoding=unicode forcechars={U+0020-U+007E}}  
}
```

Add Latin characters to an Arabic font. This use case is detailed in Section 6.4.5, »Arabic Text Formatting«, page 164.

Identify missing glyphs. The font *Unicode BMP Fallback SIL*, which is freely available, displays the hexadecimal value of each Unicode value instead of the actual glyph. This font can be very useful for diagnosing font-related problems in the workflow. You can use the following font loading option list for the *fallbackfonts* option to augment any font with this special fallback font to visualize missing characters:

```
fallbackfonts={{fontname={Unicode BMP Fallback SIL} encoding=unicode}}
```

Add Gaiji characters to a font. This use case is detailed in Section 6.5.3, »EUDC and SING Fonts for Gaiji Characters«, page 169.

5.5 Font Embedding and Subsetting

5.5.1 Font Embedding

PDF font embedding and font substitution in Acrobat. PDF documents can include font data in various formats to ensure proper text display. Alternatively, a font descriptor containing only the character metrics and some general font information (but not 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 and configured («Use Local Fonts»), or try to build a substitute font according to the font descriptor. The use of substitution fonts results in readable text, but the glyphs may look different from the original font. More importantly, Acrobat’s substitution fonts (*AdobeSansMM* and *AdobeSerifMM*) work only for Latin text, but not any other scripts or symbolic glyphs. Similarly, substitution fonts don’t work if complex script shaping or OpenType layout features have been used. For these reasons font embedding is generally recommended unless you know that the documents are displayed on the target systems acceptably even without embedded fonts. Such PDF files are inherently nonportable, but may be of use in controlled environments, such as corporate networks where the required fonts are known to be available on all workstations.

Embedding fonts with PDFlib. Font embedding is controlled by the *embedding* option when loading a font (although in some cases PDFlib enforces font embedding):

```
font = p.load_font("WarnockPro", "winansi", "embedding");
```

Table 5.3 lists different situations with respect to font usage, each of which imposes different requirements on the font and metrics files required by PDFlib. In addition to the requirements listed in Table 5.3 the corresponding CMap files (plus in some cases the Unicode mapping CMap for the respective character collection, e.g. *Adobe-Japan1-UCS2*) must be available in order to use a (standard or custom) CJK font with any of the standard CMaps.

Font embedding for fonts which are exclusively used for invisible text (mainly useful for OCR results) can be controlled with the *optimizeinvisible* option when loading the font.

Table 5.3 Different font usage situations and required files

font usage	font metrics file required?	font outline file required?
one of the 14 core fonts	no	only if embedding=true
TrueType, OpenType, or PostScript Type 1 host font installed on the Mac or Windows system	no	no
non-core PostScript fonts	yes	only if embedding=true
TrueType fonts	n/a	yes
OpenType and SING fonts	n/a	yes
standard CJK fonts ¹	no	no

1. See Section 6.5, «Chinese, Japanese, and Korean Text Output», page 166, for more information on CJK fonts.

Legal aspects of font embedding. It's important to note that mere possession of a font file may not justify embedding the font in PDF, even for holders of a legal font license. Many font vendors restrict embedding of their fonts. Some type foundries completely forbid PDF font embedding, others offer special online or embedding licenses for their fonts, while still others allow font embedding provided subsetting is applied to the font. Please check the legal implications of font embedding before attempting to embed fonts with PDFlib. PDFlib will honor embedding restrictions which may be specified in a TrueType or OpenType font. If the embedding flag in a TrueType font is set to *no embedding*¹, PDFlib will honor the font vendor's request, and reject any attempt at embedding the font.

5.5.2 Font Subsetting

In order to decrease the size of the PDF output, PDFlib can embed only those glyphs of 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
- ▶ Type 3 fonts (special handling required, see »Type 3 font subsetting«, page 139.)

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 (assuming *autosubsetting* is not specified):

- ▶ 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 (except Type 3 fonts which require special handling, see below). The default value is *true*.
- ▶ If *autosubsetting=true*: 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:

```
p.set_value("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.

- ▶ If *autosubsetting=true*: 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.

Embedding and subsetting TrueType fonts. 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*.

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

Specifying the initial font subset. Font subsets contain outline descriptions for all glyphs used in the document. This means that the generated font subsets will vary among documents since a different set of characters (and therefore glyphs) is generally used in each document. Different font subsets can be problematic when merging many small documents with embedded font subsets to a larger document: the embedded subsets cannot be removed since they are all different.

For this scenario PDFlib allows you to specify the initial contents of a font subset with the *initialsubset* option of *PDF_load_font()*. While PDFlib starts with an empty subset by default and adds glyphs as required by the generated text output, the *initialsubset* option can be used to specify a non-empty subset. For example, if you know that only Latin-1 text output will be generated and the font contains many more glyphs, you can specify the first Unicode block as initial subset:

```
initialsubset={U+0020-U+00FF}
```

This means that the glyphs for all Unicode characters in the specified range will be included in the subset. If this range has been selected so that it covers all text in the generated documents, the generated font subsets will be identical in all documents. As a result, when combining such documents later to a single PDF the identical font subsets can be removed with the *optimize* option of *PDF_begin_document()*.

Type 3 font subsetting. Type 3 fonts must be defined and therefore embedded before they can be used in a document (because the glyph widths are required). On the other hand, subsetting is only possible after creating all pages (since the glyphs used in the document must be known to determine the proper subset). In order to avoid this conflict, PDFlib supports widths-only Type 3 fonts. If you need subsetting for a Type 3 font you must define the font in two passes:

- ▶ The first pass with the *widthsonly* option of *PDF_begin_font()* must be done before using the font. It defines only the font and glyph metrics (widths); the font matrix in *PDF_begin_font()* as well as *wx* and the glyph bounding box in *PDF_begin_glyph()* must be supplied and must accurately describe the actual glyph metrics. Only *PDF_begin_glyph()* and *PDF_end_glyph()* are required for each glyph, but not any other calls for defining the actual glyph shape. If other functions are called between start and end of a glyph description, they will not have any effect on the PDF output, and will not raise any exception.
- ▶ The second pass must be done after creating all text in this font, and defines the actual glyph outlines or bitmaps. Font and glyph metrics will be ignored since they are already known from the first pass. After the last page has been created, PDFlib also knows which glyphs have been used in the document, and will only embed the required glyph descriptions to construct the font subset.

The same set of glyphs must be provided in pass 1 and pass 2. A Type 3 font with subsetting can only be loaded once with *PDF_load_font()*.

Cookbook A full code sample can be found in the *Cookbook* topic `fonts/type3_subsetting`.

5.6 Querying Font Information

`PDF_info_font()` can be used to query useful information related to fonts, encodings, Unicode, and glyphs. Depending on the type of query, a valid font handle may be required as parameter for this function. In all examples below we use the variables described in Table 5.4.

Table 5.4 Variables for use in the examples for `PDF_info_font()`

variable	comments
int uv;	Numerical Unicode value; as an alternative glyph name references without the & and ; decoration can be used in the option list, e.g. unicode=euro. For more details see the description of the Unichar option list data type in the PDFlib API Reference.
int c;	8-bit character code
int gid;	glyph id
int cid;	CID value
String gn;	glyph name
int gn_idx;	String index for a glyph name; if gn_idx is different from -1 the corresponding string can be retrieved as follows: gn = p.get_parameter("string", gn_idx);
String enc;	encoding name
int font;	valid font handle retrieved with <code>PDF_load_font()</code>

If the requested combination of keyword and option(s) is not available, `PDF_info_font()` will return -1. This must be checked by the client application and can be used to check whether or not a required glyph is present in a font.

Each of the sample code lines below can be used in isolation since they do not depend on each other.

5.6.1 Font-independent Encoding, Unicode, and Glyph Name Queries

Encoding queries. Encoding queries do not require any valid font handle, i.e. the value -1 (in PHP: 0) can be supplied for the `font` parameter of `PDF_info_font()`. Only glyph names known internally to PDFlib can be supplied in `gn`, but not any font-specific glyph names.

Query the 8-bit code of a Unicode character or a named glyph an 8-bit encoding:

```
c = (int) p.info_font(-1, "code", "unicode=" + uv + " encoding=" + enc);
c = (int) p.info_font(-1, "code", "glyphname=" + gn + " encoding=" + enc);
```

Query the Unicode value of an 8-bit code or a named glyph in an 8-bit encoding:

```
uv = (int) p.info_font(-1, "unicode", "code=" + c + " encoding=" + enc);
uv = (int) p.info_font(-1, "unicode", "glyphname=" + gn + " encoding=" + enc);
```

Query the registered glyph name of an 8-bit code or a Unicode value in an 8-bit encoding:

```
gn_idx = (int) p.info_font(-1, "glyphname", "code=" + c + " encoding=" + enc);
gn_idx = (int) p.info_font(-1, "glyphname", "unicode=" + uv + " encoding=" + enc);

/* retrieve the actual glyph name using the string index */
gn = p.get_parameter("string", gn_idx);
```

Unicode and glyph name queries. *PDF_info_font()* can also be used to perform queries which are independent from a specific 8-bit encoding, but affect the relationship of Unicode values and glyph names known to PDFlib internally. Since these queries are independent from any font, a valid font handle is not required.

Query the Unicode value of an internally known glyph name:

```
uv = (int) p.info_font(-1, "unicode", "glyphname=" + gn + " encoding=unicode");
```

Query the internal glyph name of a Unicode value:

```
gn_idx = (int) p.info_font(-1, "glyphname", "unicode=" + uv + " encoding=unicode");

/* retrieve the actual glyph name using the string index */
gn = p.get_parameter("string", gn_idx);
```

5.6.2 Font-specific Encoding, Unicode, and Glyph Name Queries

The following queries relate to a specific font which must be identified by a valid font handle. The *gn* variable can be used to supply internally known glyph names as well as font-specific glyph names. In all examples below the return value -1 means that the font does not contain the requested glyph.

Query the 8-bit codes for a Unicode value, glyph ID, named glyph, or CID in a font which has been loaded with an 8-bit encoding:

```
c = (int) p.info_font(font, "code", "unicode=" + uv);
c = (int) p.info_font(font, "code", "glyphid=" + gid);
c = (int) p.info_font(font, "code", "glyphname=" + gn);
c = (int) p.info_font(font, "code", "cid=" + cid);
```

Query the Unicode value for a code, glyph ID, named glyph, or CID in a font:

```
uv = (int) p.info_font(font, "unicode", "code=" + c);
uv = (int) p.info_font(font, "unicode", "glyphid=" + gid);
uv = (int) p.info_font(font, "unicode", "glyphname=" + gn);
uv = (int) p.info_font(font, "unicode", "cid=" + cid);
```

Query the glyph id for a code, Unicode value, named glyph, or CID in a font:

```
gid = (int) p.info_font(font, "glyphid", "code=" + c);
gid = (int) p.info_font(font, "glyphid", "unicode=" + uv);
gid = (int) p.info_font(font, "glyphid", "glyphname=" + gn);
gid = (int) p.info_font(font, "glyphid", "cid=" + cid);
```

Query the glyph id for a code, Unicode value, or named glyph in a font with respect to an arbitrary 8-bit encoding:

```
gid = (int) p.info_font(font, "glyphid", "code=" + c + " encoding=" + enc);
gid = (int) p.info_font(font, "glyphid", "unicode=" + uv + " encoding=" + enc);
gid = (int) p.info_font(font, "glyphid", "glyphname=" + gn + " encoding=" + enc);
```

Query the font-specific name of a glyph specified by code, Unicode value, glyph ID, or CID:

```
gn_idx = (int) p.info_font(font, "glyphname", "code=" + c);
gn_idx = (int) p.info_font(font, "glyphname", "unicode=" + uv);
gn_idx = (int) p.info_font(font, "glyphname", "glyphid=" + gid);
gn_idx = (int) p.info_font(font, "glyphname", "cid=" + cid);
```

```
/* retrieve the actual glyph name using the string index */
gn = p.get_parameter("string", gn_idx);
```

Checking glyph availability. Using *PDF_info_font()* you can check whether a particular font contains the glyphs you need for your application. As an example, the following code checks whether the Euro glyph is contained in a font:

```
/* We could also use "unicode=U+20AC" below */
if (p.info_font(font, "code", "unicode=euro") == -1)
{
    /* no glyph for Euro sign available in the font */
}
```

Cookbook A full code sample can be found in the *Cookbook* topic *fonts/glyph_availability*.

Alternatively, you can call *PDF_info_textline()* to check the number of unmapped characters for a given text string, i.e. the number of characters in the string for which no appropriate glyph is available in the font. The following code fragment queries results for a string containing a single Euro character (which is expressed with a glyph name reference). If one unmapped character is found this means that the font does not contain any glyph for the Euro sign:

```
String optlist = "font=" + font + " charref";

if (p.info_textline("&euro;", "unmappedchars", optlist) == 1)
{
    /* no glyph for Euro sign available in the font */
}
```

5.6.3 Querying Codepage Coverage and Fallback Fonts

PDF_info_font() can also be used to check whether a font is suited for creating text output in a certain language or script, provided the codepage is known which is required for the text. Codepage coverage is encoded in the OS/2 table of the font. Note that it is up to the font designer to decide what exactly it means that a font support a particular codepage. Even if a font claims to support a specific codepage this does not necessarily mean that it contains glyphs for all characters in this codepage. If more precise coverage information is required you can query the availability of all required characters as demonstrated in Section 5.6.2, »Font-specific Encoding, Unicode, and Glyph Name Queries«, page 141.

Checking whether a font supports a codepage. The following fragment checks whether a font supports a particular codepage:

```
String cp="cp1254";

result = (int) p.info_font(font, "codepage", "name=" + cp);

if (result == -1)
    System.err.println("Codepage coverage unknown");
else if (result == 0)
    System.err.println("Codepage not supported by this font");
else
    System.err.println("Codepage supported by this font");
```

Retrieving a list of all supported codepages. The following fragment queries a list of all codepages supported by a TrueType or OpenType font:

```
cp_idx = (int) p.info_font(font, "codepagelist", "");

if (cp_idx == -1)
    System.err.println("Codepage list unknown");
else
{
    System.err.println("Codepage list:");
    System.err.println(p.get_parameter("string", cp_idx));
}
```

This will create the following list for the common Arial font:

```
cp1252 cp1250 cp1251 cp1253 cp1254 cp1255 cp1256 cp1257 cp1258 cp874 cp932 cp936 cp949
cp950 cp1361
```

Query fallback glyphs. You can use *PDF_info_font()* to query the results of the fallback font mechanism (see Section 5.4.6, »Fallback Fonts«, page 133, for details on fallback fonts). The following fragment determines the name of the base or fallback font which is used to represent the specified Unicode character:

```
result = p.info_font(basefont, "fallbackfont", "unicode=U+03A3");
/* if result==basefont the base font was used, and no fallback font was required */
if (result == -1)
{
    /* character cannot be displayed with neither base font nor fallback fonts */
}
else
{
    idx = p.info_font(result, "fontname", "api");
    fontname = p.get_parameter("string", idx);
}
```



6 Text Output

6.1 Text Output Methods

PDFlib supports text output on several levels:

- ▶ Low-level text output with *PDF_show()* and similar functions;
- ▶ Single-line formatted text output with *PDF_fit_textline()*; This function also supports text on a path.
- ▶ Multi-line text formatted output with Textflow (*PDF_fit_textflow()* and related functions); The Textflow formatter can also wrap text inside or outside of vector-based shapes.
- ▶ Text in tables; the table formatter supports Textline and Textflow contents in table cells.

Low-level text output. functions like *PDF_show()* can be used to place text at a specific location on the page, without using any formatting aids. This is recommended only for applications with very basic text output requirements (e.g. convert plain text files to PDF), or for applications which already have full text placement information (e.g. a driver which converts a page in another format to PDF).

The following fragment creates text output with low-level functions:

```
font = p.load_font("Helvetica", "unicode", "");  
  
p.setfont(font, 12);  
p.set_text_pos(50, 700);  
p.show("Hello world!");  
p.continue_text("(says Java)");
```

Formatted single-line text output with Textlines. *PDF_fit_textline()* creates text output which consists of single lines and offers a variety of formatting features. However, the position of individual Textlines must be determined by the client application.

The following fragment creates text output with a Textline. Since font, encoding, and fontsize can be specified as options, a preceding call to *PDF_load_font()* is not required:

```
p.fit_textline(text, x, y, "fontname=Helvetica encoding=unicode fontsize=12");
```

See Section 8.1, »Placing and Fitting Textlines«, page 193, for more information about Textlines.

Multi-line text output with Textflow. *PDF_fit_textflow()* creates text output with an arbitrary number of lines and can distribute the text across multiple columns or pages. The Textflow formatter supports a wealth of formatting functions.

The following fragment creates text output with Textflow:

```
tf = p.add_textflow(tf, text, optlist);  
result = p.fit_textflow(tf, llx, lly, urx, ury, optlist);  
p.delete_textflow(tf);
```

See Section 8.2, »Multi-Line Textflows«, page 201, for more information about Textflow.

Text in tables. Textlines and Textflows can also be used to place text in table cells. See Section 8.3, »Table Formatting«, page 221, for more information about table features.

6.2 Font Metrics and Text Variations

6.2.1 Font and Glyph 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*.

Glyph metrics. PDFlib uses the glyph 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.

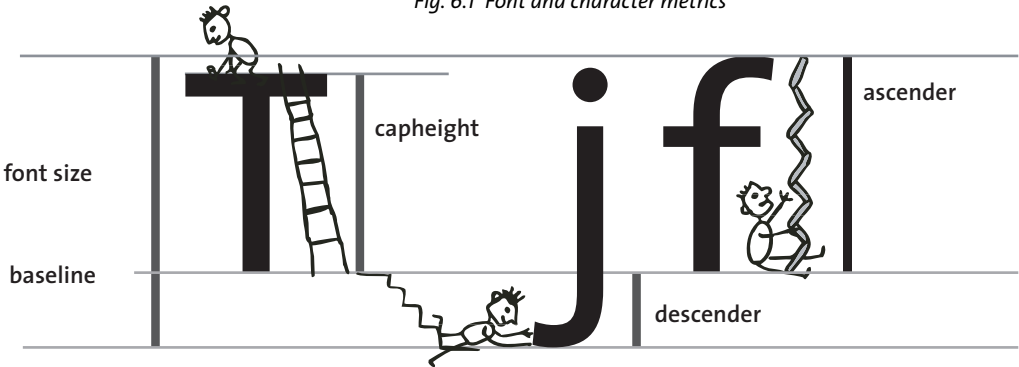
The *leading* (line spacing) specifies the vertical distance between the baselines of adjacent lines of text. By default it is set to the value of the font size. The *capheight* is the height of capital letters such as *T* or *H* in most Latin fonts. The *xheight* is the height of lowercase letters such as *x* 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 *xheight*, *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 *gaplen* property is only available in TrueType and OpenType fonts (it will be estimated for other font formats). The *gaplen* value is read from the font file, and specifies the difference between the recommended distance between baselines and the sum of ascender and descender.

PDFlib may have to estimate one or more of these values since they are not guaranteed to be present in the font or metrics file. In order to find out whether real or estimated values are used you can call *PDF_info_font()* to query the *xheight* with the option *faked*. The character metrics for a specific font can be queried from PDFlib as follows:

```
font = p.load_font("Times-Roman", "unicode", "");  
  
capheight = p.info_font(font, "capheight", "");  
ascender = p.info_font(font, "ascender", "");
```

Fig. 6.1 Font and character metrics



```
descender = p.info_font(font, "descender", "");  
xheight = p.info_font(font, "xheight", "");
```

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

Cookbook A full code sample can be found in the Cookbook topic `fonts/font_metrics_info`.

CPI calculations. While most fonts have varying character widths, so-called mono-spaced fonts use the same widths for all characters. In order to relate PDF font metrics to the characters per inch (CPI) measurements often used in high-speed print environments, some calculation examples for the mono-spaced Courier font may be helpful. In Courier, all characters have a width of 600 units with respect to the full character cell of 1000 units per point (this value can be retrieved from the corresponding AFM metrics file). For example, with 12 point text all characters will have an absolute width of

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

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.

6.2.2 Kerning

Some character combinations can lead to unpleasant appearance. For example, two characters *V* 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 information which specifies spacing adjustments for critical letter combinations. PDFlib uses kerning data from the following sources:

- TrueType and OpenType fonts: kerning pairs specified in the *kern* table;
- OpenType fonts: pair-based and class-based kerning data specified via the *kern* feature and the *GPOS* table;
- PostScript Type 1 fonts: kerning pairs specified in AFM and PFM files;
- kerning pairs for the PDF core fonts are provided by PDFlib internally.

Tele Vaso

No kerning

Tele Vaso

Kerning applied

Te Va

Character movement caused by kerning

Fig. 6.2 Kerning

There are two PDFlib controls for the kerning behavior:

- ▶ By default, kerning information in a font will be read when loading the font. If kerning is not required the *readkerning* option can be set to *false* in *PDF_load_font()*.
- ▶ Kerning for text output must be enabled with the *kerning* text appearance option which is supported by the text output functions.

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. Note that modern TrueType and OpenType fonts include special figures for this purpose which can be used with the *Tabular Figures* layout feature and the option *features={tnum}*.

Kerning will be applied in addition to any character spacing, word spacing, and horizontal scaling which may be active. PDFlib does not impose any limit for the number of kerning pairs in a font.

6.2.3 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 bold-italic styles by emboldening or slanting the base font. The aesthetic quality of artificial font styles does not match that 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 option for fonts other than the standard CJK fonts is not recommended. Also note that the fontstyle option may not work in PDF viewers other than Adobe Acrobat.

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 »Latin core fonts«, page 129). Font styles can not be applied to TrueType Collections (TTC).
- ▶ The encoding is *winansi* or *macroman* for TrueType fonts, or one of the predefined CJK CMaps listed in Table 4.3 (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()*:

```
font = p.load_font("HeiseiKakuGo-W5", "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.

Cookbook A full code sample can be found in the *Cookbook* topic `fonts/artificial_fontstyles`.

Simulated bold fonts. While *fontstyle* feature operates on a font, PDFlib supports an alternate mechanism for creating artificial bold text for individual text strings. This is controlled by the *fakebold* parameter or option.

Cookbook A full code sample can be found in the *Cookbook* topic `fonts/simulated_fontstyles`.

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 clockwise. 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 or option can be used to easily simulate one. This feature may be especially useful for CJK fonts. Typical values for the *italicangle* parameter or option are in the range -12 to -15 degrees.

Note The *italicangle* parameter or option is not supported for vertical writing mode.

Note PDFlib does not adjust the glyph width to the new bounding box of the slanted glyph. For example, when generated justified text the italicized glyphs may exceed beyond the fitbox.

Shadow text. PDFlib can create a shadow effect which will generate multiple instances of text where each instance is placed at a slightly different location. Shadow text can be created with the *shadow* option of *PDF_fit_textline()*. The color of the shadow, its position relative to the main text and graphics state parameters can be specified in suboptions.

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. The respective parameter names for *PDF_set_parameter()* can be used to switch the underline, overline, and strikeout feature on or off, as well as the corresponding options in the text output functions. The *underline-position* and *underlinewidth* parameters and options can be used for fine-tuning.

The current stroke color is used for drawing the bars. The current linecap parameter are ignored. The *decorationabove* option controls whether or not the line will be drawn on top of or below the text. Aesthetics alert: in most fonts underlining will touch descenders, and overlining will touch diacritical marks atop ascenders.

Cookbook A full code sample can be found in the *Cookbook* topic `text_output/starter_textline`.

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 the *PDFlib API Reference*, and can be set with the *textrendering* parameter or option.

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.

Cookbook Full code samples can be found in the *Cookbook* topics `text_output/text_as_clipping_path` and `text_output/invisible_text`.

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 `o` has been selected, both stroke and fill color may affect the text depending on the selected rendering mode.

Cookbook A full code sample can be found in the *Cookbook* topic `text_output/starter_textline`.

6.3 OpenType Layout Features

Cookbook Full code samples can be found in the Cookbook topics `text_output/starter_opentype` and `font/opentype_feature_tester`.

6.3.1 Supported OpenType Layout Features

PDFlib supports enhanced text output according to additional information in some fonts. These font extensions are called OpenType layout features. For example, a font may contain a *liga* feature which includes the information that the *f*, *f*, and *i* glyphs can be combined to form a ligature. Other common examples are small caps in the *smcp* feature, i.e. uppercase characters which are smaller than the regular uppercase characters, and old-style figures in the *onum* feature with ascenders and descenders (as opposed to lining figures which are all placed on the baseline). Although ligatures are a very common OpenType feature, they are only one of many dozen possible features. An overview of the OpenType format and OpenType feature tables can be found at

www.microsoft.com/typography/developers/opentype/default.htm

PDFlib supports the following groups of OpenType features:

- ▶ OpenType features for Western typography listed in Table 6.1; these are controlled by the *features* option.
- ▶ OpenType features for Chinese, Japanese, and Korean text output listed in Table 6.7; these are also controlled by the *features* option, and are discussed in more detail in Section 6.5.4, »OpenType Layout Features for advanced CJK Text Output«, page 170.
- ▶ OpenType features for complex script shaping and vertical text output; these are automatically evaluated subject to the *shaping* and *script* options (see Section 6.4, »Complex Script Output«, page 158). The *vert* feature is controlled by the *vertical* font option.
- ▶ OpenType feature tables for kerning; however, PDFlib doesn't treat *kerning* as OpenType feature because kerning data may also be represented with other means than OpenType feature tables. Use the *readkerning* font option and the *kerning* text option instead to control kerning (see Section 6.2.2, »Kerning«, page 148).

More detailed descriptions of OpenType layout features can be found at

www.microsoft.com/typography/otspec/featuretags.htm

Identifying OpenType features. You can identify OpenType feature tables with the following tools:

- ▶ The FontLab font editor is an application for creating and editing fonts. The free demo version (www.fontlab.com) displays and previews OpenType features
- ▶ DTL OTMaster Light (www.fonttools.org) is a free application for viewing and analyzing fonts, including their OpenType feature tables.
- ▶ Microsoft's free »font properties extension«¹ displays a list of OpenType features available in a font (see Figure 6.3).
- ▶ PDFlib's *PDF_info_font()* interface can also be used to query supported OpenType features (see »Querying OpenType features programmatically«, page 156).

¹ See www.microsoft.com/typography/TrueTypeProperty21.mspx

Table 6.1 Supported OpenType features for Western typography (Table 6.7 lists OpenType features for CJK text)

key-word	name	description
_none	all features disabled	Deactivate all OpenType features listed in Table 6.1 and Table 6.7.
afrc	alternative fractions	Replace figures separated by a slash with an alternative form.
c2pc	petite capitals from capitals	Turn capital characters into petite capitals.
c2sc	small capitals from capitals	Turn capital characters into small capitals.
case	case-sensitive forms	Shift various punctuation marks up to a position that works better with all-capital sequences or sets of lining figures; also changes oldstyle figures to lining figures.
dlig	discretionary ligatures	Replace a sequence of glyphs with a single glyph which is preferred for typographic purposes.
dnom	denominators	Replace figures which follow a slash with denominator figures.
frac	fractions	Replace figures separated by a slash with 'common' (diagonal) fractions.
hist	historical forms	Replace the default (current) forms with the historical alternates. Some letter forms were in common use in the past, but appear anachronistic today.
hlig	historical ligatures	This feature replaces the default (current) ligatures with the historical alternates.
liga	standard ligatures	Replace a sequence of glyphs with a single glyph which is preferred for typographic purposes.
lnum	lining figures	Change figures from oldstyle to the default lining form.
locl	localized forms	Enable localized forms of glyphs to be substituted for default forms. This feature requires the script and language options.
mgrk	mathematical Greek	Replace standard typographic forms of Greek glyphs with corresponding forms commonly used in mathematical notation.
numr	numerators	Replace figures which precede a slash with numerator figures and replace the typographic slash with the fraction slash.
onum	oldstyle figures	Change figures from the default lining style to oldstyle form.
ordn	ordinals	Replace default alphabetic glyphs with the corresponding ordinal forms for use after figures; commonly also creates the Numero (U+2116) character.
ornm	ornaments	Replace the bullet character and ASCII characters with ornaments.
pcap	petite capitals	Turn lowercase characters into petite capitals, i.e. capital letters which are shorter than regular small caps.
pnum	proportional figures	Replace monospaced (tabular) figures with figures which have proportional widths.
salt	stylistic alternates	Replace the default forms with stylistic alternates. These alternates don't always fit into a clear category like swash or historical.
sinf	scientific inferiors	Replace lining or oldstyle figures with inferior figures (smaller glyphs), primarily for chemical or mathematical notation).
smcp	small capitals	Turn lowercase characters into small capitals.
ss01 ... ss20	stylistic set 1-20	In addition to, or instead of, stylistic alternatives of individual glyphs (see salt feature), some fonts may contain sets of stylistic variant glyphs corresponding to portions of the character set, e.g. multiple variants for lowercase letters in a Latin font.

Table 6.1 Supported OpenType features for Western typography (Table 6.7 lists OpenType features for CJK text)

key-word	name	description
subs	subscript	Replace a default glyph with a subscript glyph.
sup	superscript	Replace lining or oldstyle figures with superior figures (primarily for footnote indication), and replace lowercase letters with superior letters (primarily for abbreviated French titles)
swsh	swash	Replace default glyphs with corresponding swash glyphs.
titl	titling	Replace default glyphs with corresponding forms designed for titling.
tnum	tabular figures	Replace proportional figures with monospaced (tabular) figures.
unic	unicase	Map upper- and lowercase letters to a mixed set of lowercase and small capital forms, resulting in a single case alphabet.
zero	slashed zero	Replace the glyph for the figure zero with an alternative form which uses a diagonal slash through the counter.

6.3.2 OpenType Layout Features with Textlines and Textflows

PDFlib supports OpenType layout features in the Textline and Textflow functions, but not in the low-level text output functions (*PDF_show()* etc.).

Requirements for OpenType layout features. A font for use with OpenType layout features must meet the following requirements:

- ▶ The font must be a TrueType (*.ttf), OpenType (*.otf) or TrueType Collection (*.ttc) font. For standard CJK fonts the corresponding font file must be available.
- ▶ The font file must contain a GSUB table with supported lookups for the OpenType feature(s) to be used in the text (see below).
- ▶ The font must be loaded with *encoding=unicode* or *glyphid*, or a Unicode CMap.
- ▶ The *readfeatures* option of *PDF_load_font()* must not be set to *false*.
- ▶ If the *fallbackfonts* option of *PDF_load_font()* was used, text in a single text run must not contain glyphs from the base font and a fallback font (or glyphs from different fallback fonts) at the same time.

Note PDFlib supports OpenType features with GSUB lookup types 1 (one-to-one substitution), 3 (alternate substitution table) and 4 (many-to-one substitution). Except for kerning PDFlib does not support OpenType features based on the GPOS table.

Caveats. Note the following when working with OpenType features:

- ▶ OpenType features (options *features*, *script*, *language*) will only be applied to glyphs within the same font, but not across glyphs from the base font and one or more fallback fonts if fallback fonts have been specified.
- ▶ Make sure to enable and disable features as you need them. Accidentally leaving OpenType features activated for all of the text may lead to unexpected results.

Enabling and disabling OpenType features. You can enable and disable OpenType features for pieces of text as required. Use the *features* text option to enable features by supplying their name, and enable them by prepending *no* to the feature name. For example, with inline option lists for Textflow feature control works as follows:

```
<features={liga}>ffi<features={noliga}
```

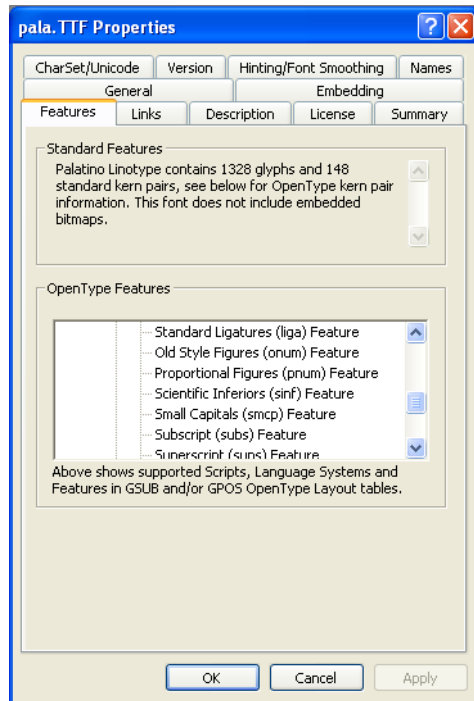


Fig. 6.3
Microsoft's font property extension displays
the list of OpenType features in a font

For Textlines you can enable OpenType features as follows:

```
p.fit_textline("ffi", x, y, "features={liga}");
```

OpenType features can also be enabled as Block properties for use with the PDFlib Personalization Server (PPS).

More than one feature can be applied to the same text, but the feature tables in the font must be prepared for this situation and must contain the corresponding feature lookups in the proper order. For example, consider the word *office*, and the ligature (*liga*) and small cap (*smcp*) features. If both features are enabled (assuming the font contains corresponding feature entries) you'd expect the small cap feature to be applied, but not the ligature feature. If this is correctly implemented in the font tables, PDFlib will generate the expected output, i.e. small caps without any ligature.

Disabling ligatures with control characters. Some languages disallow the use of languages in certain situations. Typographic rules for German and other languages prohibit the use of ligatures across composition boundaries. For example, the *f+i* combination in the word *Schilfinsel* must not be replaced with a ligature since it spans the boundaries between two combined words.

As described above, you can enable and disable ligatures and other OpenType feature processing with the *features* option. Disabling ligatures via options can be cumbersome in exceptional cases like the one described above. In order to offer simple ligature control you can disable ligatures with control characters in the text, avoiding the need for enabling/disabling features with multiple options. Inserting the character *Zero-width non-joiner* (U+200C, ‍ see also Table 6.4) between the constituent characters

will prevent them from being replaced by a ligature even if ligatures are enabled in the *features* option. For example, the following sequence will not create any *f+i* ligature:

```
<features={liga charref=true}>Schilf&zwj;insel
```

Script- and language-specific OpenType layout features. OpenType features may apply in all situations, or can be implemented for a particular script or even a particular script and language combination. For this reason the *script* and *language* text options can optionally be supplied along with the *features* option. They will have a noticeable effect only if the feature is implemented in a script- or language-specific manner in the font.

As an example, the ligature for the *f* and *i* glyphs is not available in some fonts if the Turkish language is selected (since the ligated form of *i* could be confused with the dotless *i* which is very common in Turkish). Using such a font the following Textflow option will create a ligature since no script/language is specified:

```
<features={liga}>fi
```

However, the following Textflow option list will not create any ligature due to the Turkish language option:

```
<script=latn language=TRK features={liga}>fi
```

The *locl* feature explicitly selects language-specific character forms. The *liga* feature contains language-specific ligatures. Some examples for language-specific features:

Variant character for Serbian:

```
<features={locl} script=cyrl language=SRB charref>&#x0431;
```

Variant figures for Urdu:

```
<features={locl} script=arab language=URD charref>&#x0662;&#x0663;&#x0664;&#x0665;
```

See Section 6.4.2, »Script and Language«, page 160, for supported script and language keywords.

Combining OpenType features and shaping. Shaping for complex scripts (see Section 6.4, »Complex Script Output«, page 158) heavily relies on OpenType font features which will be selected automatically. However, for some fonts it may make sense to combine OpenType features selected automatically for shaping with OpenType features which have been selected by the client application. PDFlib will first apply user-selected OpenType features (option *features*) before applying shaping-related features automatically (options *shaping*, *script* and *language*).

Querying OpenType features programmatically. You can query OpenType features in a font programmatically with *PDF_info_font()*. The following statement retrieves a space-separated list with all OpenType features which are available in the font and are supported by PDFlib:

```
result = (int) p.info_font(font, "featurelist", "");
if (result != -1)
{
    /* retrieve string containing space-separated feature list */
    featurelist = p.get_parameter("string", result);
}
else
```

```
{
    /* no supported features found */
}
```

Use the following statement to check whether PDFlib and the test font support a particular feature, e.g. ligatures (*liga*):

```
result = (int) p.info_font(font, "feature", "name=liga");
if (result == 1)
{
    /* feature supported by font and PDFlib */
}
```

6.4 Complex Script Output

Cookbook A full code sample can be found in the Cookbook topic `complex_scripts/starter_shaping`.

6.4.1 Complex Scripts

The Latin script basically places one character after the other in left-to-right order. Other writing systems have additional requirements for correct text output. We refer to such writing systems as complex scripts. PDFlib performs text processing for complex scripts for a variety of scripts including those listed in Table 6.2.

In this section we will discuss shaping for complex scripts in more detail. While most Western languages can be written by simply placing one character after the other from left to right, some writing systems (scripts) require additional processing:

- ▶ The Arabic and Hebrew scripts place text from right to left. Mixed text (e.g. Arabic with a Latin insert) contains both right-to-left and left-to-right segments. These segments must be reordered, which is referred to as the Bidi (bidirectional) problem.
- ▶ Some scripts, especially Arabic, use different character shapes depending on the position of the character (isolated, beginning/middle/end of a word).
- ▶ Mandatory ligatures replace sequences of characters.
- ▶ The position of glyphs must be adjusted horizontally and vertically.
- ▶ Indic scripts require reordering of some characters, i.e. characters may change their position in the text.
- ▶ Special word break and justification rules apply to some scripts.

Scripts which require one or more of these processing steps are called complex scripts. The process of preparing incoming logical text for proper presentation is called shaping (this term also includes reordering and Bidi processing). The user always supplies text in unshaped form and in logical order, while PDFlib performs the necessary shaping before producing PDF output.

Complex script shaping can be enabled with the *shaping* text option, which in turn requires the *script* option and optionally allows the *language* option. The following option list enables Arabic shaping (and Bidi processing):

```
shaping script=arab
```

Caveats. Note the following when working with complex script shaping:

- ▶ PDFlib does not automatically set the *shaping* and *script* options, but expects them to be supplied by the user.
- ▶ Script-specific shaping (options *shaping*, *script*, *language*) will only be applied to glyphs within the same font, but not across glyphs from different fonts. If fallback fonts are used, shaping will only be applied within text runs which contain text in the same (master or fallback) font.
- ▶ Since shaping may reorder characters in the text, care must be taken regarding attribute changes within a word. For example, if you use inline options in Textflow to colorize the second character in a word – what should happen when shaping swaps the first and second characters? For this reason, formatting changes should only be applied at word boundaries, but not within words.

Table 6.2 Complex scripts and keywords for the script option

writing system	script name	language/region (incomplete list)	script keyword
unspecified script	–		_none
automatic script detection	–	This keyword selects the script to which the majority of characters in the text belong, where _latn and _none are ignored.	_auto
European Alphabetic	Latin	many European and other languages	latn
	Greek	Greek	grek
	Cyrillic	Russian and many other Slavic languages	cyr1
Middle Eastern	Arabic	Arabic, Persian (Farsi), Urdu, Pashto and others	arab
	Hebrew	Hebrew, Yiddish and others	hebr
	Syriac	Syrian Orthodox, Maronite, Assyrian	syr1
	Thaana	Dhivehi/Maldives	thaa
South Asian (India)	Devanagari	Hindi and classical Sanskrit	deva
	Bengali	Bengali, Assamese	beng
	Gurmukhi	Punjabi	guru
	Gujarati	Gujarati	gujr
	Oriya	Oriya/Orissa	orya
	Tamil	Tamil/Tamil Nadu, Sri Lanka	tam1
	Telugu	Telugu/Andhra Pradesh	telu
	Kannada	Kannada/Karnataka	knda
	Malayalam	Malayalam/Kerala	mlym
Southeast Asian	Thai	Thai	thai
	Lao	Lao	»lao « ¹
	Khmer	Khmer (Cambodian)	khmr
East Asian	Han	Chinese, Japanese, Korean	hani
	Hiragana	Japanese	hira
	Katakana	Japanese	kana
	Hangul	Korean	hang
others	Other four-character codes according to the OpenType specification also work, but are not supported. The full list can be found at the following location: www.microsoft.com/typography/developers/OpenType/scripttags.aspx		

1. Note the trailing space character.

Requirements for shaping. A font for use with complex script shaping must meet the following requirements in addition to containing glyphs for the target script:

- ▶ It must be a TrueType or OpenType font with GDEF, GSUB, and GPOS feature tables and correct Unicode mappings appropriate for the target script. As an alternative to the OpenType tables, for the Arabic and Hebrew scripts, the font may contain glyphs for the Unicode presentation forms (e.g. Arabic Apple fonts are constructed this way). In this case internal tables will be used for the shaping process. For Thai text the font must contain contextual forms according to Microsoft, Apple, or Monotype Worldtype (e.g. used in some IBM products) conventions for Thai.
- ▶ If standard CJK fonts are to be used, the corresponding font file must be available.
- ▶ The font must be loaded with *encoding=unicode* or *glyphid*.
- ▶ The *monospace* and *vertical* options of *PDF_load_font()* must not be used, and the *readshaping* option must not be set to *false*.
- ▶ If the *fallbackfonts* option of *PDF_load_font()* was used, text in a single text run must not contain glyphs from a fallback font.

6.4.2 Script and Language

Script and language settings play a role the functional aspects listed below. They can be controlled with the following options:

- ▶ The *script* text option identifies the target script (writing system). It supports the four-letter keywords listed in Table 6.2. Examples:

```
script=latn
script=cyrl
script=arab
script=hebr
script=deva
script={lao }
```

With *script=_auto* PDFlib automatically assigns that script to which the majority of characters in the text belong. Since Latin text doesn't require shaping it will not be counted when determining the script automatically.

You can query the scripts used for some text with the *scriptlist* keyword of *PDF_info_textline()*.

- ▶ The *language* option specifies the natural language in which the text is written. It supports the three-character keywords listed in Table 6.3. Examples:

```
language=ARA
language=URD
language=ZHS
language=HIN
```

Complex script processing. Complex script processing (option *shaping*) requires the *script* option. The *language* option can additionally be supplied. It controls language-specific aspects of shaping, e.g. different figures for Arabic vs. Urdu. However, only few fonts contain language-specific script shaping tables, so in most cases specifying the *script* option will be sufficient, and shaping cannot be improved with the *language* option.

OpenType layout features. Fonts can implement OpenType layout features in a language-specific manner (see »Script- and language-specific OpenType layout features«,

page 156). While a few features may differ in behavior subject to the *script* and *language* options but can also be used without these options (e.g. *liga*), the *locl* feature only makes sense in combination with the *script* and *language* options.

Note While advanced line breaking for Textflow (see Section 8.2.9, »Advanced script-specific Line Breaking«, page 216) also applies language-specific processing, it is not controlled by the language option, but by the locale option which identifies not only languages, but also countries and regions.

Table 6.3 Keywords for the language option

key-word	language	key-word	language	key-word	language
_none	unspecified language	FIN	Finnish	NEP	Nepali
AFK	Afrikaans	FRA	French	ORI	Oriya
SQI	Albanian	GAE	Gaelic	PAS	Pashto
ARA	Arabic	DEU	German	PLK	Polish
HYE	Armenian	ELL	Greek	PTG	Portuguese
ASM	Assamese	GUJ	Gujarati	ROM	Romanian
EUQ	Basque	HAU	Hausa	RUS	Russian
BEL	Belarussian	IWR	Hebrew	SAN	Sanskrit
BEN	Bengali	HIN	Hindi	SRB	Serbian
BGR	Bulgarian	HUN	Hungarian	SND	Sindhi
CAT	Catalan	IND	Indonesian	SNH	Sinhalese
CHE	Chechen	ITA	Italian	SKY	Slovak
ZHP	Chinese phonetic	JAN	Japanese	SLV	Slovenian
ZHS	Chinese simplified	KAN	Kannada	ESP	Spanish
ZHT	Chinese traditional	KSH	Kashmiri	SVE	Swedish
COP	Coptic	KHM	Khmer	SYR	Syriac
HRV	Croatian	KOK	Konkani	TAM	Tamil
CSY	Czech	KOR	Korean	TEL	Telugu
DAN	Danish	MLR	Malayalam reformed	THA	Thai
NLD	Dutch	MAL	Malayalam traditional	TIB	Tibetan
DZN	Dzongkha	MTS	Maltese	TRK	Turkish ¹
ENG	English	MNI	Manipuri	URD	Urdu
ETI	Estonian	MAR	Marathi	WEL	Welsh
FAR	Farsi	MNG	Mongolian	JII	Yiddish

1. Some fonts wrongly use *TUR* for Turkish; PDFlib treats this tag as equivalent to *TRK*.

6.4.3 Complex Script Shaping

The shaping process selects appropriate glyph forms depending on whether a character is located at the start, middle, or end of a word, or in a standalone position. Shaping is a crucial component of Arabic and Hindi text formatting. Shaping may also replace a sequence of two or more characters with a suitable ligature. Since the shaping process determines the appropriate character forms automatically, explicit ligatures and Unicode presentation forms (e.g. Arabic Presentation Forms-A U+FB50) must not be used as input characters.

Since complex scripts require multiple different glyph forms per character and additional rules for selecting and placing these glyphs, shaping for complex scripts does not work with all kinds of fonts, but requires suitable fonts which contain the necessary information. Shaping works for TrueType and OpenType fonts which contain the required feature tables (see »Requirements for shaping«, page 160, for detailed requirements).

Shaping can only be done for characters in the same font because the shaping information is specific to a particular font. As it doesn't make sense, for example, to form ligatures across different fonts, complex script shaping cannot be applied to a word which contains characters from different fonts.

Override shaping behavior. In some cases users may want to override the default shaping behavior. PDFlib supports several Unicode formatting characters for this purpose. For convenience, these formatting characters can also be specified with entity names (see Table 6.4).

Table 6.4 Unicode control characters for overriding the default shaping behavior

<i>formatting character</i>	<i>entity name</i>	<i>Unicode name</i>	<i>function</i>
U+200C	ZWNJ	ZERO WIDTH NON-JOINER	prevent the two adjacent characters from forming a cursive connection
U+200D	ZWJ	ZERO WIDTH JOINER	force the two adjacent characters to form a cursive connection

6.4.4 Bidirectional Formatting

Cookbook A full code sample can be found in the Cookbook topic `complex_scripts/bidi_formatting`.

For right-to-left text (especially Arabic and Hebrew, but also some other scripts) it is very common to have nested sequences of left-to-right Latin text, e.g. an address or a quote in another language. These mixed sequences of text require bidirectional (Bidi) formatting. Since numerals are always written from left to right, the Bidi problem affects even text which is completely written in Arabic or Hebrew. PDFlib implements bidirectional text reordering according to the Unicode Bidi algorithm as specified in Unicode Standard Annex #9¹. Bidi processing does not have to be enabled with an option, but will automatically be applied as part of the shaping process if text in a right-to-left script with an appropriate *script* option is encountered.

1. See www.unicode.org/unicode/reports/tr9/

Note Bidi processing is not currently supported for multi-line Textflows, but only for Textlines (i.e. single-line text output).

Overriding the Bidi algorithm. While automatic Bidi processing will provide proper results in common cases, there are situations which require explicit user control. PDFlib supports several directional formatting codes for this purpose. For convenience, these formatting characters can also be specified with entity names (see Table 6.5). The bidirectional formatting codes are useful to override the default Bidi algorithm in the following situations:

- ▶ a right-to-left paragraph begins with left-to-right characters;
- ▶ there are nested segments with mixed text;
- ▶ there are weak characters, e.g. punctuation, at the boundary between left-to-right and right-to-left text;
- ▶ part numbers and similar entities containing mixed text.

Table 6.5 Directional formatting codes for overriding the bidirectional algorithm

formatting code	entity name	Unicode name	function
U+202A	LRE	LEFT-TO-RIGHT EMBEDDING (LRE)	start an embedded left-to-right sequence
U+202B	RLE	RIGHT-TO-LEFT EMBEDDING (RLE)	start an embedded right-to-left sequence
U+200E	LRM	LEFT-TO-RIGHT MARK (LRM)	left-to-right zero-width character
U+200F	RLM	RIGHT-TO-LEFT MARK (RLM)	right-to-left zero-width character
U+202D	LRO	LEFT-TO-RIGHT OVERRIDE (LRO)	force characters to be treated as strong left-to-right characters
U+202E	RLO	RIGHT-TO-LEFT OVERRIDE (RLO)	force characters to be treated as strong right-to-left characters
U+202C	PDF	POP DIRECTIONAL FORMATTING (PDF)	restore the bidirectional state to what it was before the last LRE, RLE, RLO, or LRO

Options for improved right-to-left document handling. The default settings of various formatting options and Acrobat behavior are targeted at left-to-right text output. Use the following options for right-to-left text formatting and document display:

- ▶ Place a Textline right-aligned with the following fitting option:
`position={right center}`
- ▶ Create a leader between the text and the left border:
`leader={alignment=left text=.`
- ▶ Use the following option of `PDF_begin/end_document()` to activate better right-to-left document and page display in Acrobat:
`viewerpreferences={direction=r2l}`

Dealing with Bidi text in your code. The following may also be useful when dealing with bidirectional text:

- ▶ You can use the `startx/starty` and `endx/endy` keywords of `PDF_info_textline()` to determine the coordinates of the logical start and end characters, respectively.

- ▶ You can use the *writingdirx* keyword of *PDF_info_textline()* to determine the dominant writing direction of text. This direction will be inferred from the initial characters of the text or from directional formatting codes according to Table 6.5 (if present in the text).
- ▶ You can use the *auto* keyword for the *position* option of *PDF_info_textline()* to automatically align Arabic or Hebrew text at the right border and Latin text at the left border. For example, the following Textline option list aligns right- or left-aligns the text on the baseline:

```
boxsize={width 0} position={auto bottom}
```

6.4.5 Arabic Text Formatting

Cookbook A full code sample can be found in the *Cookbook* topic `complex_scripts/arabic_formatting`.

In addition to Bidirectional formatting and text shaping as discussed above there are several other formatting aspects related to generating text output in the Arabic script.

Arabic ligatures. The Arabic script makes extensive use of ligatures. Many Arabic fonts contain two kinds of ligatures which are treated differently in PDFlib:

- ▶ Required ligatures (*rlig* feature) must always be applied, e.g. the Lam-Alef ligature and variants thereof. Required ligatures will be used if the *shaping* option is enabled with *script=arab*.
- ▶ Optional Arabic ligatures (*liga* and *dlig* features) are not used automatically, but can be enabled like other user-controlled OpenType features, i.e. *features={liga}*. Optional Arabic ligatures will be applied after complex script processing and shaping.

Latin ligatures in Arabic text. In Textlines the script-specific processing of OpenType features may produce unexpected results. For example, Latin ligatures don't work in combination with Arabic text within the same Textline. The reason is that the *script* option can be supplied only once for the contents of a Textline and affects both the *shaping* and *feature* options:

```
shaping script=arab features={liga}          WRONG, does not work with common fonts!
```

However, Arabic fonts typically don't contain Latin ligatures with an Arabic script designation, but only for the default or Latin script – but the *script* option cannot be changed within a single Textline. Because of this PDFlib will not find any Latin ligatures and will emit plain characters instead.

Avoiding ligatures. In some cases joining adjacent characters is not desired, e.g. for certain abbreviations. In such cases you can use the formatting characters listed in Table 6.4 to force or prevent characters from being joined. For example, the zero-width non-joiner character in the following example will prevent the characters from being joined in order to form a proper abbreviation:

```
&#x0623;&#x064A;&ZWJ;&#x0628;&#x064A;&ZWJ;&#x0625;&#x0645;
```

Tatweel formatting for Arabic text. You can stretch Arabic words by inserting one or more instances of the tatweel character U+o64o (also called kashida). While PDFlib does not automatically justify text by inserting tatweel characters, you can insert this character in the input text to stretch words.

Adding Latin characters to an Arabic font. Some Arabic fonts do not contain any glyphs for Latin characters, e.g. the Arabic fonts bundled with Apple Mac OS X. In this situation you can use the *fallbackfonts* option to merge Latin characters into an Arabic font. PDFlib will automatically switch between both fonts depending on the Latin or Arabic text input, i.e. you don't have to switch fonts in your application but can supply the mixed Latin/Arabic text with a single font specification.

You can use the following font loading option list for the *fallbackfonts* option to add Latin characters from the Helvetica font to the loaded Arabic font:

```
fallbackfonts={  
  {fontname=Helvetica encoding=unicode forcechars={U+0021-U+00FF}}  
}
```

6.5 Chinese, Japanese, and Korean Text Output

6.5.1 Standard CJK Fonts

Acrobat supports various 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 are listed in Table 6.6 along with applicable CMaps (see Section 4.3, »Chinese, Japanese, and Korean Encodings«, page 99, for more details on CJK CMaps).

Note Instead of Unicode CMaps (UCS2 or UTF16) the use of encoding=unicode is recommended for custom CJK fonts.

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 6.2.3, »Text Variations«, page 149).

Table 6.6 Acrobat's standard fonts and CMaps (encodings) for Japanese, Chinese, and Korean text

locale	font name	sample	supported CMaps (encodings)
Simplified Chinese	AdobeSongStd-Light ²	国际	GB-EUC-H, GB-EUC-V, GBpc-EUC-H, GBpc-EUC-V, GBK-EUC-H, GBK-EUC-V, GBKp-EUC-H, GBKp-EUC-V, GBK2K-H, GBK2K-V, UniGB-UCS2-H, UniGB-UCS2-V, UniGB-UTF16-H ¹ , UniGB-UTF16-V ¹
Traditional Chinese	AdobeMingStd-Light ²	中文	B5pc-H, B5pc-V, HKscs-B5-H, HKscs-B5-V, ETen-B5-H, ETen-B5-V, ETenms-B5-H, ETenms-B5-V, CNS-EUC-H, CNS-EUC-V, UniCNS-UCS2-H, UniCNS-UCS2-V, UniCNS-UTF16-H ¹ , UniCNS-UTF16-V ¹
Japanese	KozMinPro-Regular-Acro ³ KozGoPro-Medium ² KozMinProVI-Regular ²	日本語	83pv-RKSJ-H, goms-RKSJ-H, goms-RKSJ-V, gomsp-RKSJ-H, gomsp-RKSJ-V, gopv-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 ³ , UniJIS-UCS2-HW-V ³ , UniJIS-UTF16-H ¹ , UniJIS-UTF16-V ¹
Korean	AdobeMyungjoStd-Medium ²	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 ¹ , UniKS-UTF16-V ¹

1. Only available when generating PDF 1.5 or above
2. Only available when generating PDF 1.6 or above
3. The HW CMaps are not allowed for the KozMinPro-Regular-Acro and KozGoPro-Medium-Acro fonts because these fonts contain only proportional ASCII characters, but not any halfwidth forms.

Keeping native CJK legacy codes. If *keepnative=true*, native legacy character codes (e.g. Shift-JIS) according to the selected CMap will be written to the PDF output; otherwise the text will be converted to Unicode. The advantage of *keepnative=true* is that such fonts can be used for form fields without embedding (see description of the *keepnative* font loading option for in the PDFlib API Reference). If *keepnative=false* legacy codes will be converted to CID values which will subsequently be written to the PDF output. The advantage is that OpenType features can be used and that the Textflow formatter can be used. The visual appearance will be identical in both cases.

Horizontal and vertical writing mode. PDFlib supports both horizontal and vertical writing modes. Vertical writing mode can be requested in different ways (note that vertical writing mode is not supported for Type 1 fonts):

- ▶ For standard CJK fonts and CMaps the writing 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.
- ▶ Fonts with encodings other than a CMap can be used for vertical writing mode by supplying the *vertical* font option.
- ▶ Font names starting with an '@' character will always be processed in vertical mode.

Note The character spacing must be negative in order to spread characters apart in vertical writing mode.

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 6.6), and that Unicode-aware language bindings support only UCS2-compatible CMaps. The *KozMinPro-Regular-Acro* sample in Table 6.6 can be generated with the following code:

```
font = p.load_font("KozMinPro-Regular-Acro", "UniJIS-UCS2-H", "");
if (font == -1) { ... }
p.setfont(font, 24);
p.set_text_pos(50, 500);
p.show("\u65E5\u672C\u8A9E");
```

These statements locate one of the Japanese standard fonts, choosing a Unicode CMap (*UniJIS-UCS2-H*) with 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.

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 = p.load_font("KozMinPro-Regular-Acro", "UniJIS-UCS2-H", "monospace=1000");
```

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

6.5.2 Custom CJK Fonts

Note PDFlib GmbH offers the MS Gothic and MS Mincho fonts for free download at www.pdf-lib.com. PDFlib licensees are entitled to use these fonts without having to obtain a separate font license.

In addition to Acrobat's standard CJK fonts PDFlib supports custom CJK fonts (fonts outside the list in Table 6.6) in the TrueType (including TrueType Collections, TTC) and OpenType formats. Custom CJK fonts will be processed as follows:

- ▶ If the *embedding* option is *true*, the font will be converted to a CID font and embedded in the PDF output.
- ▶ CJK host font names on Windows can be supplied to *PDF_load_font()* as UTF-8 with initial BOM, or UTF-16. Non-Latin host font names are not supported on the Mac, though.
- ▶ The *keepnative* option is *false* by default. In order to avoid subtle problems in Acrobat we recommend to set *keepnative=false* if no font embedding is desired, and to set *embedding=true* if *keepnative=true* is desired.

Custom CJK font example with Japanese Shift-JIS text. The following C 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", "");
if (font == -1) { ... }
PDF_setfont(p, font, 24);
PDF_set_text_pos(p, 50, 500);

PDF_show2(p, "\x82\xA9\x82\xC8\x8A\xBF\x8E\x9A", 8);
```

Note that legacy encodings such as Shift-JIS are not supported in Unicode-aware language bindings.

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 5.4.4, »Searching for Fonts«, page 126):

```
font = p.load_font("Arial Unicode MS", "unicode", "");

p.setfont(font, 24);
p.set_text_pos(50, 500);

p.show("\u4e00\u500b\u4eba");
```

Accessing individual fonts in a TrueType Collection (TTC). TTC files contain multiple separate fonts. You can access each font by supplying its proper name. However, if you don't know which fonts are contained in a TTC file you can numerically address each font by appending a colon character and the number of the font within the TTC file (starting with 0). If the index is 0 it can be omitted. For example, the TTC file *msgothic.ttc* contains multiple fonts which can be addressed as follows in *PDF_load_font()* (each line contains equivalent font names):



msgothic:0	MS Gothic	msgothic:
msgothic:1	MS PGothic	
msgothic:2	MS UI Gothic	

Note that *msgothic* (without any suffix) will not work as a font name since it does not uniquely identify a font. Font name aliases (see »Sources of Font Data«, page 126) can be used in combination with TTC indexing. If a font with the specified index cannot be found, the function call will fail.

It is only required to configure the TTC font file once; all indexed fonts in the TTC file will be found automatically. The following code is sufficient to configure all indexed fonts in *msgothic.ttc* (see Section 5.4.4, »Searching for Fonts«, page 126):

```
p.set_parameter("FontOutline", "msgothic=msgothic.ttc");
```

6.5.3 EUDC and SING Fonts for Gaiji Characters

PDFlib supports Windows EUDC (end-user defined characters, *.tte) and SING fonts (*.gai) which can be used to access custom Gaiji characters for CJK text. Most conveniently fonts with custom characters are integrated into other fonts with the fallback font mechanism. Gaiji characters will commonly be provided in EUDC or SING fonts. Alternatively, Gaiji characters can also be supplied as Type 3 fonts, but this requires more programming effort.

Using fallback fonts for Gaiji characters. Typically, Gaiji characters will be pulled from Windows EUDC fonts or SING glyphlets, but the *fallbackfonts* option accepts any kind of font. Therefore this approach is not limited to Gaiji characters, but can be used for any kind of symbol (e.g. a company logo in a separate font). You can use the following font loading option list for the *fallbackfonts* option to add a user-defined (gaiji) character from an EUDC font to the loaded font:

```
fallbackfonts={
  {fontname=EUDC encoding=unicode forcechars=U+E000 fontsize=140% textrise=-20%}
}
```

Once a base font has been loaded with this fallback font configuration, the EUDC character can be used within the text without any need to change the font.

With SING fonts the Unicode value doesn't have to be supplied since it will automatically be determined by PDFlib:

```
fallbackfonts={
  {fontname=PDFlibWing encoding=unicode forcechars=gaiji}
}
```

Preparing EUDC fonts. 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. Since this file is invisible in Windows Explorer 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 (see Section 5.4.4, »Searching for Fonts«, page 126):

```
p.set_parameter("FontOutline", "EUDC=EUDC.TTE");
p.set_parameter("SearchPath", "...directory name...");
```

or place *EUDC.TTE* in the current directory.

As an alternative to this explicit font file configuration 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:

```
p.set_parameter("FontOutline", "EUDC=C:\Windows\fonts\EUDC.TTE");
```

- Integrate the EUDC font into any base font using the *fallbackfonts* option as described above. If you want to access the font directly, use the following call to load the font in PDFlib:

```
font = p.load_font("EUDC", "unicode", "");
```

as usual and supply the Unicode value(s) chosen in the first step to output the characters.

6.5.4 OpenType Layout Features for advanced CJK Text Output

As detailed in Section 6.3, »OpenType Layout Features«, page 152, PDFlib supports advanced typographic layout tables in OpenType and TrueType fonts. For example, OpenType features can be used to select alternative forms of the Latin glyphs with proportional widths or half widths, or to select alternate character forms. Table 6.7 lists OpenType features which are targeted at CJK text output.

The *vert* feature (vertical writing) will be automatically enabled for fonts with vertical writing mode (i.e. the *vertical* option has been supplied to *PDF_load_font()*), and disabled for fonts with horizontal writing mode.

Table 6.7 Supported OpenType layout features for Chinese, Japanese, and Korean text (Table 6.1 lists additional supported OpenType layout features for general use)

key-word	name	description
expt	expert forms	Like the JIS78 forms this feature replaces standard Japanese forms with corresponding forms preferred by typographers.
fwid	full widths	Replace glyphs set on other widths with glyphs set on full (usually em) widths. This may include Latin characters and various symbols.
hkna	horizontal Kana alternates	Replace standard Kana with forms that have been specially designed for only horizontal writing.
hngl	Hangul	Replace hanja (Chinese-style) Korean characters with the corresponding Hangul (syllabic) characters.
hojo	Hojo Kanji forms (JIS X 0212-1990)	Access the JIS X 0212-1990 glyphs (also called »Hojo Kanji«) if the JIS X 0213:2004 form is encoded as default.
hwid	half widths	Replace glyphs on proportional widths, or fixed widths other than half an em, with glyphs on half-em (en) widths.
ital	italics	Replace the Roman glyphs with the corresponding Italic glyphs.
jp04	JIS2004 forms	(Subset of the nlck feature) Access the JIS X 0213:2004 glyphs.
jp78	JIS78 forms	Replace default (JIS90) Japanese glyphs with the corresponding forms from JIS C 6226-1978 (JIS78).

Table 6.7 Supported OpenType layout features for Chinese, Japanese, and Korean text (Table 6.1 lists additional supported OpenType layout features for general use)

key-word	name	description
jp83	JIS83 forms	Replace default (JIS90) Japanese glyphs with the corresponding forms from JIS X 0208-1983 (JIS83).
jp90	JIS90 forms	Replace Japanese glyphs from JIS78 or JIS83 with the corresponding forms from JIS X 0208-1990 (JIS90).
locl	localized forms	Enable localized forms of glyphs to be substituted for default forms. This feature requires the script and language options.
nalt	alternate annotation forms	Replace default glyphs with various notational forms (e.g. glyphs placed in open or solid circles, squares, parentheses, diamonds or rounded boxes).
nlck	NLC Kanji forms	Access the new glyph shapes defined in 2000 by the National Language Council (NLC) of Japan for a number of JIS characters.
pkna	proportional Kana	Replace glyphs, Kana and Kana-related, set on uniform widths (half or full-width) with proportional glyphs.
pwid	proportional widths	Replace glyphs set on uniform widths (typically full or half-em) with proportionally spaced glyphs.
qwid	quarter widths	Replace glyphs on other widths with glyphs set on widths of one quarter of an em (half an en).
ruby	Ruby notation forms	Replace default Kana glyphs with smaller glyphs designed for use as (usually superscripted) Ruby.
smp1	simplified forms	Replace traditional Chinese or Japanese forms with the corresponding simplified forms.
tnam	traditional name forms	Replace simplified Japanese Kanji forms with the corresponding traditional forms. This is equivalent to the trad feature, but limited to the traditional forms considered proper for use in personal names.
trad	traditional forms	Replace simplified Chinese Hanzi or Japanese Kanji forms with the corresponding traditional forms.
twid	third widths	Replace glyphs on other widths with glyphs set on widths of one third of an em.
vert	vertical writing	Replace default forms with variants adjusted for vertical writing.
vkna	vertical Kana alternates	Replace standard Kana with forms that have been specially designed for only vertical writing.
vrt2	vertical alternates and rotation	(Overrides the vert feature which is a subset of the vrt2 feature) Replace some fixed-width (half-, third- or quarter-width) or proportional-width glyphs (mostly Latin or Katakana) with forms suitable for vertical writing (that is, rotated 90° clockwise).



7 Importing Images and PDF Pages

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. Placing images and PDF pages on an output page is discussed in Section 7.3, »Placing Images and imported PDF Pages«, page 186.

Cookbook Code samples regarding image issues can be found in the *images* category of the *PDFlib Cookbook*.

7.1 Importing Raster Images

7.1.1 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:

```
image = p.load_image("auto", "image.jpg", "");
if (image == -1)
    throw new Exception("Error: " + p.get_errmsg());

p.fit_image(image, 0.0, 0.0, "");
p.close_image(image);
```

The last parameter of `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 7.3, »Placing Images and imported PDF Pages«, page 186.

Cookbook A full code sample can be found in the *Cookbook* topic `images/starter_image`.

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.

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 with its native resolution (or 72 dpi if it doesn't contain any resolution informa-

tion) if the user coordinate system hasn't been scaled (since there are 72 default units to an inch).

Cookbook A full code sample can be found in the *Cookbook* topic `images/image_dimensions`. It shows how to get the dimensions of an image and how to place it with various sizes.

Color space of imported images. Except for adding or removing ICC profiles and applying a spot color according to the options provided in `PDF_load_image()`, PDFlib will generally try to preserve the native color space of an imported image. However, this is not possible for certain rare combinations, such as YCbCr in TIFF which will be converted to RGB.

PDFlib does not perform any conversion between RGB and CMYK. If such a conversion is required it must be applied to the image data before loading the image in PDFlib.

Multi-page images. PDFlib supports GIF, TIFF and JBIG2 images with more than one image, also known as multi-page image files. In order to use multi-page images use the `page` option in `PDF_load_image()`:

```
image = p.load_image("tiff", filename, "page=2");
```

The `page` option indicates that a multi-image file is to be used, and 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.

Cookbook A full code sample for converting all images in a multi-image TIFF file to a multi-page PDF file can be found in the *Cookbook* topic `images/multi_page_tiff`.

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). The primary use of inline images is for bitmap glyph descriptions in Type 3 fonts; inline images are not recommended for other situations.

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:

```
p.fit_image(image, 0, 0, "");  
p.close_image(image);
```

Inline images are only supported for `imagetype=ccitt`, `jpeg`, and `raw`. For other image types the `inline` option will silently be ignored.

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:

```
String 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 = p.load_image("tiff", filename, 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()`.

XMP metadata in images. Image files may contain XMP metadata. By default PDFlib ignores image metadata for images in the TIFF, JPEG, and JPEG 2000 image formats to reduce the output file size. However, the XMP metadata can be attached to the generated image in the output PDF document with the following option of `PDF_load_image()`:

```
metadata={keepxmp=true}
```

7.1.2 Supported Image File Formats

PDFlib deals with the image file formats described below. By default, PDFlib passes the compressed image data unchanged to the PDF output if possible 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*). Pass-through mode can be controlled with the *passthrough* option of `PDF_load_image()`.

If an image file can't be imported successfully `PDF_load_image()` will return an error code. If you need to know more details about the image failure, call `PDF_get_errmsg()` to retrieve a detailed error message.

PNG images. PDFlib supports all flavors of PNG images (ISO 15948). PNG images are handled in pass-through mode in most cases. If a PNG image contains transparency information, the transparency is retained in the generated PDF (see Section 7.1.4, «Image Masks and Transparency», page 178).

JPEG images. JPEG images (ISO 10918-1) are never decompressed, but some flavors may require transcoding for proper display in Acrobat. PDFlib automatically applies transcoding to certain types of JPEG images, but transcoding can also be controlled via the *passthrough* option of `PDF_load_image()`. Transcoding does not change the pixel count or color of an image, and does not introduce any visible compression/decompression artifacts. PDFlib supports the following JPEG image flavors:

- ▶ Grayscale, RGB (usually encoded as YCbCr), and CMYK color
- ▶ 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. PDFlib will also read clipping paths from JPEG images created with Adobe Photoshop.

JPEG images in the EXIF format as created by many digital cameras are treated as sRGB images; an sRGB ICC profile will be attached to the image unless the *honoriccprofile* option is *false* or another ICC profile has been assigned to the image with the *iccprofile* option.

JPEG 2000 images. JPEG 2000 images (ISO 15444-2) require PDF 1.5 or above, and are always handled in pass-through mode. PDFlib supports JPEG 2000 images as follows:

- ▶ JP2 and JPX baseline images (usually **.jp2* or **.jpf*) are supported, subject to the color space conditions below. All valid color depth values are supported.
The following color spaces are supported: sRGB, sRGB-grey, ROMM-RGB, sYCC, e-sRGB, e-sYCC, CIELab, ICC-based color spaces (restricted and full ICC profile), and CMYK. PDFlib will not alter the original color space in the JPEG 2000 image file.
- ▶ Images containing a soft mask can be used with the *mask* option to prepare a mask which can be applied to other images.
- ▶ External ICC profiles can not be applied to a JPEG 2000 image, i.e. the *iccprofile* option must not be used. ICC profiles contained in the JPEG 2000 image file will always be kept, i.e. the *honoriccprofile* option is always *true*.

*Note JPM compound image files according to ISO 15444-6 (usually *.jpm) are not supported.*

JBIG2 images. PDFlib supports single- and multi-page flavors of JBIG2 images (ISO 14492). JBIG2 images always contain black/white pixel data and require PDF 1.4 or above.

Due to the nature of JBIG2 compression, several pages in a multi-page JBIG2 stream may refer to the same global segments. If more than one page of a multi-page JBIG2 stream is converted the global segments can be shared among the generated PDF images. Since the calls to *PDF_load_image()* are independent from each other you must inform PDFlib in advance that multiple pages from the same JBIG2 stream will be converted. This works as follows:

- ▶ When loading the first page all global segments are copied to the PDF. Use the following option list for *PDF_load_image()*:

```
page=1 copyglobals=all
```

- ▶ When loading subsequent pages from the same JBIG2 stream the image handle<N> for page 1 must be provided so that PDFlib can create references to the global segments which have been copied with page 1. Use the following option list for *PDF_load_image()*:

```
page=2 imagehandle=<N>
```

The client application must make sure that the *copyglobals/imagehandle* mechanism is only applied to multiple pages which are extracted from the same JBIG2 image stream. Without the *copyglobals* options PDFlib will automatically copy all required data for the current page.

GIF images. PDFlib supports all GIF flavors (specifically GIF 87a and 89a) with interlaced and non-interlaced pixel data and all palette sizes. GIF images will always be re-compressed with Flate compression.

TIFF images. PDFlib imports almost all flavors of TIFF images:

- ▶ Compression schemes: uncompressed, CCITT (group 3, group 4, and RLE), ZIP (=Flate), PackBits (=RunLength), LZW, old-style and new-style JPEG, as well as some other rare compression schemes;
- ▶ Color space: black and white, grayscale, RGB, CMYK, CIELab, and YCbCr images;
- ▶ Color depth must be 1, 2, 4, 8, or 16 bits per color component. 16-bit images require PDF 1.5.

The following TIFF features will be processed when importing an image:

- ▶ TIFF files containing more than one image (see »Multi-page images«, page 174); use the *page* option to select an image within a TIFF file.
- ▶ Alpha channels or masks (see Section 7.1.4, »Image Masks and Transparency«, page 178) will be honored unless the *ignoremask* option is set. You can explicitly select an alpha channel with the *alphachannelname* option.
- ▶ PDFlib honors clipping paths in TIFF images created with Adobe Photoshop and compatible programs unless the *ignoreclippingpath* option is set.
- ▶ PDFlib honors ICC profiles in TIFF images unless the *honoriccprofile* option is set to *false*.
- ▶ The orientation tag which specifies the desired image orientation will be honored. It can be ignored (as many applications do) with the *ignoreorientation* option.

Some TIFF features (e.g., spot color) and combinations of features 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. 16-bit images will be treated as 5+5+5 plus 1 unused bit. 32-bit images will be treated as 3×8 bit images (the remaining 8 bits will be ignored).
- ▶ 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.



Fig. 7.1
Using a clipping path to separate
foreground and background

7.1.3 Clipping Paths

PDFlib supports clipping paths in TIFF and JPEG images created with Adobe Photoshop. An image file may contain multiple named paths. Using the *clippingpathname* option of *PDF_load_image()* one of the named paths can be selected and will be used as a clipping path: only those parts of the image inside the clipping path will be visible, other parts will remain invisible. This is useful to separate background and foreground, eliminate unwanted portions of an image, etc.

Alternatively, an image file may specify a default clipping path. If PDFlib finds a default clipping path in an image file it will automatically apply it to an image (see Figure 7.1). In order to prevent the default clipping path from being applied set the *honor-clippingpath* option in *PDF_load_image()* to *false*. If you have several instances of the same image and only some instances shall have the clipping path applied, you can supply the *ignoreclippingpath* option in *PDF_fit_image()* in order to disable the clipping path. When a clipping path is applied, the bounding box of the clipped image will be used as the basis for all calculations related to placing or fitting the image.

Cookbook A full code sample can be found in the *Cookbook* topic `images/integrated_clipping_path`.

7.1.4 Image Masks and Transparency

PDFlib supports three kinds of transparency information in images: implicit transparency with alpha channels, explicit transparency, and image masks.

Implicit transparency with alpha channels. Raster images may be partially transparent, i.e. the background shines through the image. This is useful, for example, to ignore the background of an image and display only the person or object in the foreground. Transparency information can be stored in a separate alpha channel or (in palette-based images) as a transparent palette entry. Transparent images are not allowed in PDF/A-1, PDF/X-1 and PDF/X-3. PDFlib processes transparency information in the following image formats:

- ▶ GIF image files may contain a single transparent color value (palette entry) which is respected by PDFlib.
- ▶ TIFF images may contain a single associated alpha channel which will be honored by PDFlib. Alternatively, a TIFF image may contain an arbitrary number of unassociated

channels which are identified by name. These channels may be used to convey transparency or other information. When unassociated channels are found in a TIFF image PDFlib will by default use the first channel as alpha channel. However, you can explicitly select an unassociated alpha channel by supplying its name:

```
image = p.load_image("tiff", filename, "alphachannelname={apple}");
```

- ▶ PNG images may contain an associated alpha channel which will automatically be used by PDFlib.
- ▶ As an alternative to a full alpha channel, PNG images may contain single transparent color values which will be honored by PDFlib. If multiple color values with an attached alpha value are given, only the first one with an alpha value below 50 percent is used.

Note In addition to a full alpha channel Photoshop can create transparent backgrounds in a proprietary format which is not understood by PDFlib. In order to use such transparent images with PDFlib you must save them in Photoshop in the TIFF file format and select Save Transparency in the TIFF options dialog box.

Sometimes it is desirable to ignore any implicit transparency which may be contained in an image file. PDFlib's transparency support can be disabled with the *ignoremask* option when loading the image:

```
image = p.load_image("tiff", filename, "ignoremask");
```

Explicit transparency. The explicit case requires two steps, both of which involve image operations. First, a grayscale image must be prepared for later use as a mask. This is accomplished by loading the mask image. The following kinds of images can be used for constructing a mask:

- ▶ PNG images
- ▶ TIFF images: the *nopassthrough* option for *PDF_load_image()* is recommended to avoid multi-strip images.
- ▶ raw image data

Pixel values of 0 (zero) 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 a transparency effect.

In the second step the mask is applied to another image with the *masked* option:

```
mask = p.load_image("png", maskfilename, "");
if (mask == -1)
    throw new Exception("Error: " + p.get_errmsg());

String optlist = "masked=" + mask;
image = p.load_image(type, filename, optlist)
if (image == -1)
    throw new Exception("Error: " + p.get_errmsg());

p.fit_image(image, x, y, "");
```

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

Note In some situations 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.

Note The mask must have the same orientation as the underlying image; otherwise it will be rejected. Since the orientation depends on the image file format and other factors it is difficult to detect. For this reason it is recommended to use the same file format and creation software for both mask and image.

Cookbook A full code sample can be found in the Cookbook topic `images/image_mask`.

Image masks and soft masks. Image masks are images with a bit depth of 1 (bitmaps) in which zero 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 colored with the current fill color.

Soft masks generalize the concept of image masks to masks with more than 1 bit. They blend the image against some existing background. PDFlib accepts all kinds of single-channel (grayscale) images as soft mask. Note that only real gray-scale images are suitable as a mask, but not images with indexed (palette-based) color. They can be used the same way as image masks. The following kinds of images can be used as image masks:

- PNG images
- JBIG2 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 mirrored 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 = p.load_image("tiff", maskfilename, "mask");
p.setcolor("fill", "rgb", 1.0, 0.0, 0.0, 0.0);
if (mask != -1)
{
    p.fit_image(mask, x, y, "");
}
```

If you want to apply a color to an image without the zero bit pixels being transparent you must use the *colorize* option (see Section 7.1.5, »Colorizing Images«, page 180).

7.1.5 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.

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.

In order to colorize an image with a spot color you must supply the *colorize* option when loading the image, and supply the respective spot color handle which must have been retrieved with *PDF_makespotcolor()*:

```
p.setcolor("fillstroke", "cmyk", 1, .79, 0, 0);
spot = p.makespotcolor("PANTONE Reflex Blue CV");

String optlist = "colorize=" + spot;
image = p.load_image("tiff", "image.tif", optlist);
if (image != -1)
{
    p.fit_image(image, x, y, "");
}
```

7.2 Importing PDF Pages with PDI

Note All functions described in this section require PDFlib+PDI. The PDF import library (PDI) is not contained in the PDFlib base product. Although PDI is integrated in all precompiled editions of PDFlib, a license key for PDI (or PPS, which includes PDI) is required to use it.

7.2.1 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 or PDF/A documents to create a new PDF/X or PDF/A file;
- ▶ copy the PDF/X or PDF/A output intent of a 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;
- ▶ use the pCOS interface to query arbitrary properties of a PDF document (see pCOS Path Reference for details).

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 11, »PPS and the PDFlib Block Plugin«, page 271).

7.2.2 Using PDI Functions with PDFlib

Cookbook Code samples regarding PDF import issues can be found in the pdf_import category of the PDFlib Cookbook.

General considerations. It is important to understand that PDI will only import the actual page contents, but not any interactive 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 interactive features can be generated with the corresponding PDFlib functions. PDFlib blocks will also be ignored when importing a page. Document structure from Tagged PDF will also be lost when importing a document.

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.

PDFlib+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+PDI uses the template feature (Form XObjects) for placing imported PDF pages on the output page. Documents which contain imported pages from other PDF documents can be processed with PDFlib+PDI again.

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;
String filename = "input.pdf";

if (p.begin_document(outfilename, "") == -1) {...}
...

doc = p.open_pdi_document(infile, "");
if (doc == -1)
    throw new Exception("Error: " + p.get_errmsg());

page = p.open_pdi_page(doc, pageno, "");
if (page == -1)
    throw new Exception("Error: " + p.get_errmsg());

/* dummy page size, will be modified by the adjustpage option */
p.begin_page_ext(20, 20, "");
p.fit_pdi_page(page, 0, 0, "adjustpage");
p.close_pdi_page(page);
...add more content to the page using PDFlib functions...
p.end_page_ext("");
p.close_pdi_document(doc);
```

The last parameter 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 7.3, »Placing Images and imported PDF Pages«, page 186.

Dimensions of imported PDF pages. Imported PDF pages are handled 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 Size«, page 66);
- ▶ rotation which has been applied to the page will be retained.

The *cloneboxes* option instructs PDFlib+PDI to copy all page boxes of the imported page to the generated output page, effectively cloning all page size aspects.

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.

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.

Importing georeferenced PDF with PDI. When importing georeferenced PDF with PDI the geospatial information will be kept if it has been created with one of the following methods (image-based geospatial reference):

- ▶ with PDFlib and the *georeference* option of *PDF_load_image()*
- ▶ by importing an image with geospatial information in Acrobat.

The geospatial information will be lost after importing a page if it has been created with one of the following methods (page-based geospatial reference):

- ▶ with PDFlib and the *viewports* option of *PDF_begin/end_page_ext()*
- ▶ by manually geo-registering a PDF page in Acrobat.

Imported PDF with OPI information. OPI information present in the input PDF will be retained in the output unmodified.

Optimization across multiple imported documents. While PDFlib itself creates highly optimized PDF output, imported PDF may contain redundant data structures which can be optimized. In addition, importing multiple PDFs may bloat the output file size if multiple files contain identical resources, e.g. fonts. In this situation you can use the *optimize* option of *PDF_begin_document()*. It will detect redundant objects in imported files, and remove them without affecting the visual appearance or quality of the generated output.

7.2.3 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.

PDI implements a repair mode for damaged PDFs so that even certain kinds of damaged documents can be opened. 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_document()* and *PDF_open_pdi_page()* return an error code. If you need to know more details about the failure you can query the reason with *PDF_get_errmsg()*. Alternatively, you can set the *errorpolicy* option or parameter to *true*, which will result in an exception if the document cannot be opened.

The following kinds of PDF documents will be rejected by default; however, they can be opened for querying information with pCOS (as opposed to importing pages) by setting the *infomode* option to *true*:

- ▶ Encrypted PDF documents without the corresponding password (exception to the *infomode* rule: PDF 1.6 documents created with the Distiller setting *Object Level Compression: Maximum*; these cannot be opened even in info mode).
- ▶ Tagged PDF when the *tagged* option in *PDF_begin_document()* is *true*.
- ▶ PDF/A or PDF/X documents which are incompatible to the PDF/A or PDF/X level of the current output document (e.g. trying to import PDF/A-1a into a PDF/A-1b document). See Section 10.3.4, »Importing PDF/X Documents with PDI«, page 252, and Section 10.4.3, »Importing PDF/A Documents with PDI«, page 258, for more information.

The following additional checks are done in *PDF_open_pdi_page()*:

- ▶ PDF documents which use a higher PDF version number than the PDF output document that is currently being generated can not be imported with PDI. 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 1.7ext 3 (Acrobat 9) and *PDF 1.7ext8* (Acrobat X) documents are compatible with PDF 1.7 output as far as PDI is concerned (note that Acrobat X encryption is not yet supported).

In PDF/A mode the input PDF version number will be ignored since PDF version headers must be ignored in PDF/A.

7.3 Placing Images and imported PDF Pages

The function `PDF_fit_image()` for placing raster images 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 the PDFlib API Reference.

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. Before calling any of the *fit* functions a call to `PDF_load_image()` or `PDF_open_pdi_document()` and `PDF_open_pdi_page()` must be issued. For the sake of simplicity these calls are not reproduced here.

Cookbook Code samples regarding images and imported PDF pages can be found in the `images` and `pdf_import` categories of the PDFlib Cookbook.

7.3.1 Simple Object Placement

Positioning an image at the reference point. By default, an object will be placed in its original size with the lower left corner at the reference point. In this example we will place an image with the bottom centered at the reference point. The following code fragment places the image with the bottom centered at the reference point (*o, o*).

```
p.fit_image(image, 0, 0, "position={center bottom}");
```

Similarly, you can use the *position* option with another combination of the keywords *left*, *right*, *center*, *top*, and *bottom* to place the object at the reference point.

Placing an image with scaling. The following variation will place the image while modifying its size:

```
p.fit_image(image, 0, 0, "scale=0.5");
```

This code fragment places the object with its lower left corner at the point (*o, o*) 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.



Cookbook A full code sample can be found in the Cookbook topic `images/starter_image`.

7.3.2 Placing an Object in a Box

In order to position an object an additional box with specified width and height can be used. Figure 7.2 shows the output of the examples described below. Note that the gray box or line is depicted for visualizing the box size only ; it is not part of the actual output.

Positioning an image in the box. We define a box and place an image within the box on the top right. The box has a width of 70 units and a height of 45 units and is placed at the reference point (0, 0). The image is placed on the top right of the box (see Figure 7.2a). Similarly, we can place the image at the center of the bottom. This case is depicted in Figure 7.2b. Note that the image will extend beyond the box if it is larger.

Fig. 7.2 Placing an image in a box subject to various positioning options

Generated output	Option list for PDF_fit_image
a) 	boxsize={70 45} position={right top}
b) 	boxsize={70 45} position={center bottom}

Using suitable fitting methods. Next, we will fit the object into the box by using various fitting methods. Let’s start with the default case, where the default fitting method is used and no clipping or scaling will be applied. The image will be placed at the center of the box, 70 units wide and 45 high. The box will be placed at reference point (0, 0). Figure 7.3a illustrates this simple case.

Decreasing the box width from 70 to 35 units doesn’t have any effect on the output. The image will remain in its original size and will exceed the box (see Figure 7.3b).

Fitting an image in the center of a box. In order to center an image within a pre-defined rectangle you don’t have to do any calculations, but can achieve this with suitable options. With *position=center* we place the image in the center of the box, 70 units wide and 45 high (*boxsize={70 45}*). Using *fitmethod=meet*, the image is proportionally re-sized until its height completely fits into the box (see Figure 7.3c).

Decreasing the box width from 70 to 35 units forces PDFlib to scale down the image until its width completely fits into the box (see Figure 7.3d).








This is the most common method of placing images. With *fitmethod=meet* it is guaranteed that the image will not be distorted, and that it will always be placed in the box as large as possible.

Completely fitting the image into a box. We can further fit the image so that it completely fills the box. This is accomplished with *fitmethod=entire*. However, this combination will rarely be useful since the image will most probably be distorted (see Figure 7.3e).

Clipping an image when fitting it into the box. Using another fit method (*fitmethod=clip*) we can clip the object if it exceeds the target box. We decrease the box size to a width and height of 30 units and position the image in its original size at the center of the box (see Figure 7.3f).

By positioning the image at the center of the box, the image will be cropped evenly on all sides. Similary, to completely show the upper right part of the image you can position it with *position={right top}* (see Figure 7.3g).

Fig. 7.3 Fitting an image into a box subject to various fit methods

Generated output	Option list for <code>PDF_fit_image()</code>
a) 	<code>boxsize={70 45} position=center</code>
b) 	<code>boxsize={35 45} position=center</code>
c) 	<code>boxsize={70 45} position=center fitmethod=meet</code>
d) 	<code>boxsize={35 45} position=center fitmethod=meet</code>
e) 	<code>boxsize={70 45} position=center fitmethod=entire</code>
f) 	<code>boxsize={30 30} position=center fitmethod=clip</code>
g) 	<code>boxsize={30 30} position={right top} fitmethod=clip</code>

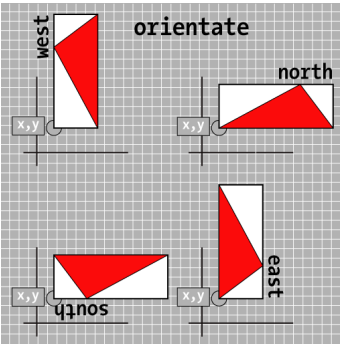
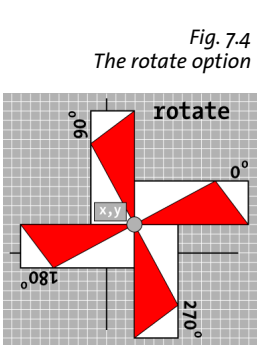
Adjusting an object to the page. Adjusting an object to a given page size can easily be accomplished by choosing the page as target box for placing the object. The following statement uses an A4-sized page with dimensions 595 x 842:

```
p.fit_image(image, 0, 0, "boxsize={595 842} position={left bottom} 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.

7.3.3 Orientating an Object

Placing an image with orientation. In our next example we orientate an image towards western direction (*orientate=west*). This means that the image is rotated by 90° counterclockwise and then the lower left corner of the rotated object is translated to the reference point (0, 0). The object will be rotated in itself (see Figure 7.6a). Since we have



not specified any fit method the image will be output in its original size and will exceed the box.

Fitting an image proportionally into a box with orientation. Our next goal is to orientate the image to the west with a predefined size. We define a box of the desired size and fit the image into the box with the image's proportions being unchanged (*fitmethod=meet*). The orientation is specified as *orientate=west*. By default, the image will be placed in the lower left corner of the box (see Figure 7.6b). Figure 7.6c shows the image orientated to the east, and Figure 7.6d the orientation to the south.

The *orientate* option supports the direction keywords *north*, *east*, *west*, and *south* as demonstrated in Figure 7.5.

Note that the *orientate* option has no influence on the whole coordinate system but only on the placed object.

Fig. 7.6 Orientating an image

Generated output	Option list for PDF_fit_image()
a)	<code>boxsize={70 45} orientate=west</code>
b)	<code>boxsize={70 45} orientate=west fitmethod=meet</code>
c)	<code>boxsize={70 45} orientate=east fitmethod=meet</code>
d)	<code>boxsize={70 45} orientate=south fitmethod=meet</code>
e)	<code>boxsize={70 45} position={center bottom} orientate=east fitmethod=clip</code>

Fitting an oriented image into a box with clipping. We orientate the image to the east (*orientate=east*) and position it centered at the bottom of the box (*position={center bottom}*). In addition, we place the image in its original size and clip it if it exceeds the box (*fitmethod=clip*) (see Figure 7.6e).

7.3.4 Rotating an Object

Rotating an object works similarly to orientation. However, it does not only affect the placed object but the whole coordinate system.

Placing an image with rotation. Our first goal is to rotate an image by 90° counterclockwise. Before placing the object the coordinate system will be rotated at the reference point (50, 0) by 90° counterclockwise. The rotated object's lower right corner (which is the unrotated object's lower left corner) will end up at the reference point. This case is shown in Figure 7.7a.

Since the rotation affects the whole coordinate system, the box will be rotated as well. Similarly, we can rotate the image by 30° counterclockwise (see Figure 7.7b). Figure 7.4 demonstrates the general behaviour of the *rotate* option.

Fitting an image with rotation. Our next goal is to fit the image rotated by 90° counterclockwise into the box while maintaining its proportions. This is accomplished using *fitmethod=meet* (see Figure 7.7c). Similarly, we can rotate the image by 30° counterclockwise and proportionally fit the image into the box (see Figure 7.7d).

Fig. 7.7 Rotating an image





Generated output	Option list for <code>PDF_fit_image()</code>
a) 	<code>boxsize={70 45} rotate=90</code>
b) 	<code>boxsize={70 45} rotate=30</code>
c) 	<code>boxsize={70 45} rotate=90 fitmethod=meet</code>
d) 	<code>boxsize={70 45} rotate=30 fitmethod=meet</code>

Fig. 7.8
Adjusting the page
size. Left to right:
exact, enlarge,
shrink



7.3.5 Adjusting the Page Size

Adjusting the page size to an image. In the next example 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 have exactly the object's size, or somewhat larger or smaller. When enlarging the page size (see Figure 7.8) some border will be kept around the image. If 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:

```
p.fit_image(image, 0, 0, "adjustpage");
```

The next code fragment increases the page size by 40 units in x and y direction, creating a white border around the object:

```
p.fit_image(image, 40, 40, "adjustpage");
```

The next code fragment decreases the page size by 40 units in x and y direction. The object will be clipped at the page borders, and some area within the object (with a width of 40 units) will be invisible:

```
p.fit_image(image, -40, -40, "adjustpage");
```

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.

Cloning the page boxes of an imported PDF page. You can copy all relevant page boxes (MediaBox, CropBox) etc. of an imported PDF page to the current output page. The *cloneboxes* option must be supplied to *PDF_open_pdi_page()* to read all relevant box values, and again in *PDF_fit_pdi_page()* to apply the box values to the current page:

```
/* Open the page and clone the page box entries */
inpage = p.open_pdi_page(indoc, 1, "cloneboxes");
...
/* Start the output page with a dummy page size */
p.begin_page_ext(10, 10, "");
...
/*
 * Place the imported page on the output page, and clone all
 * page boxes which are present in the input page; this will
 * override the dummy size used in begin_page_ext().
 */
p.fit_pdi_page(inpage, 0, 0, "cloneboxes");
```

Using this technique you can make sure that the pages in the generated PDF will have the exact same page size, cropping are etc. as the pages of the imported document. This is especially important for prepress applications.

7.3.6 Querying Information about placed Images and PDF Pages

Information about placed images. The *PDF_info_image()* function can be used to query image information. The supported keywords for this function cover general image information (e.g. width and height in pixels) as well as geometry information related to placing the image on the output page (e.g. width and height in absolute values after performing the fitting calculations).

The following code fragment retrieves both the pixel size and the absolute size after placing an image with certain fitting options:

```
String optlist = "boxsize={300 400} fitmethod=meet orientate=west";
p.fit_image(image, 0.0, 0.0, optlist);

imagewidth = (int) p.info_image(image, "imagewidth", optlist);
imageheight = (int) p.info_image(image, "imageheight", optlist);
System.err.println("image size in pixels: " + imagewidth + " x " + imageheight);

width = p.info_image(image, "width", optlist);
height = p.info_image(image, "height", optlist);
System.err.println("image size in points: " + width + " x " + height);
```

Information about placed PDF pages. The *PDF_info_pdi_page()* function can be used to query information about placed PDF pages. The supported keywords for this function cover information about the original page (e.g. its width and height) as well as geometry information related to placing the imported PDF on the output page (e.g. width and height after performing the fitting calculations).

The following code fragment retrieves both the original size of the imported page and the size after placing the page with certain fitting options:

```
String optlist = "boxsize={400 500} fitmethod=meet";
p.fit_pdi_page(page, 0, 0, optlist);

pagewidth = p.info_pdi_page(page, "pagewidth", optlist);
pageheight = p.info_pdi_page(page, "pageheight", optlist);
System.err.println("original page size: " + pagewidth + " x " + pageheight);

width = p.info_pdi_page(page, "width", optlist);
height = p.info_pdi_page(page, "height", optlist);
System.err.println("size of placed page: " + width + " x " + height);
```


8 Text and Table Formatting

8.1 Placing and Fitting Textlines

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 the PDFlib API Reference. 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 7.3, »Placing Images and imported PDF Pages«, page 186, for an introduction to image formatting.

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

`PDF_fit_textline()` uses a hypothetical 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 modified by the *matchbox* option which defines the text box.

In the examples below, the coordinates of the reference point are supplied as *x, y* parameters of `PDF_fit_textline()`. The fitbox for text lines is the area where text will be placed. It is defined as the rectangular area specified with the *x, y* parameters of `PDF_fit_textline()` and appropriate options (*boxsize, position, rotate*). The fitbox can be reduced to the left/right or top/bottom with the *margin* option.

Cookbook Code samples regarding text output issues can be found in the `text_output` category of the *PDFlib Cookbook*.

8.1.1 Simple Textline Placement

Positioning text at the reference point. By default, the text will be placed with the lower left corner at the reference point. However, in this example we want to place the text with the bottom centered at the reference point. The following code fragment places the text box with the bottom centered at the reference point (30, 20).

```
p.fit_textline(text, 30, 20, "position={center bottom}");
```

Figure 8.1 illustrates centered text placement. Similarly, you can use the *position* option with another combination of the keywords *left, right, center, top, and bottom* to place text at the reference point.

Fig. 8.1
Centered text

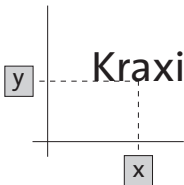
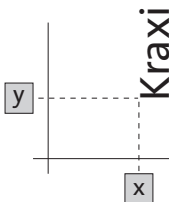


Fig. 8.2
Simple text with
orientation west



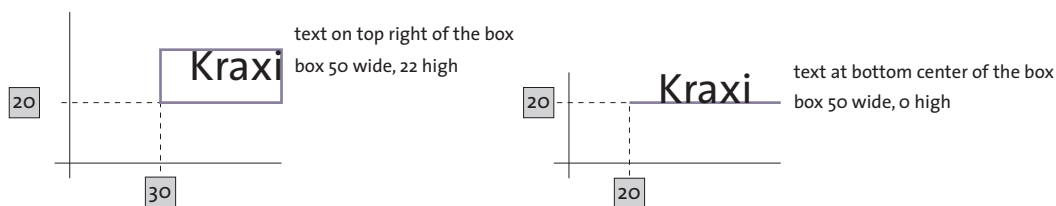


Fig. 8.3 Positioning text in a box

Placing text with orientation. Our next goal is to rotate text while placing its lower left corner (after the rotation) at the reference point. The following code fragment orientates the text to the west (90° counterclockwise) and then translates the lower left corner of the rotated text to the reference point (0, 0).

```
p.fit_textline(text, 0, 0, "orientate=west");
```

Figure 8.2 illustrates simple text placement with orientation.

8.1.2 Positioning Text in a Box

In order to position the text, an additional box with predefined width and height can be used, and the text can be positioned relative to this box. Figure 8.3 illustrates the general behaviour.



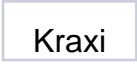

Positioning text in the box. We define a rectangular box and place the text within this box on the top right. The following code fragment defines a box with a width of 50 units and a height of 22 units at reference point (30, 20). In Figure 8.4a, the text is placed on the top right of the box.


Similarly, we can place the text at the center of the bottom. This case is illustrated in Figure 8.4b.

To achieve some distance between the text and the box we can add the *margin* option (see Figure 8.4c).

Note that the blue box or line depicted for visualizing the box size in the figures is not part of the actual output.

Fig. 8.4 Placing text in a box subject to various positioning options

Generated output	Option list for <code>PDF_fit_textline()</code>
a) 	<code>boxsize={50 22} position={right top}</code>
b) 	<code>boxsize={50 22} position={center bottom}</code>
c) 	<code>boxsize={50 22} position={center bottom} margin={0 3}</code>
d) 	<code>boxsize={50 0} position={center bottom}</code>

Generated output	Option list for <code>PDF_fit_textline()</code>
	<code>boxsize={0 35} position={left center} orientate=west</code>

Aligning text at a horizontal or vertical line. Positioning text along a horizontal or vertical line (i.e. a box with zero height or width) is a somewhat extreme case which may be useful nevertheless. In Figure 8.4d the text is placed with the bottom centered at the box. With a width of 50 and a height of 0, the box resembles to a horizontal line.

To align the text centered along a vertical line we will orientate it to the west and position it at the left center of the box. This case is shown in Figure 8.4e.

8.1.3 Fitting Text into a Box

In this section we use various fit methods to fit the text into the box. The current font and font size are assumed to be the same in all examples so that we can see how the font size and other properties will implicitly be changed by the different fit methods.

Let's start with the default case: no fit method will be used so that no clipping or scaling occurs. The text will be placed in the center of the box which is 100 units wide and 35 units high (see Figure 8.5a).

Decreasing the box width from 100 to 50 units doesn't have any effect on the output. The text will remain in its original font size and will exceed beyond the box (see Figure 8.5b).

Proportionally fitting text into a small box. Now we will completely fit the text into the box while maintaining its proportions. This can be achieved with the `fitmethod=auto` option. In Figure 8.5c the box is wide enough to keep the text in its original size completely so that the text will be fit into the box unchanged.

When scaling down the width of the box from 100 to 58, the text is too long to fit completely. The `auto` fit method will try to condense the text horizontally, subject to the `shrinklimit` option (default: 0.75). Figure 8.5d shows the text being shrunk down to 75 percent of its original length.

When decreasing the box width further down to 30 units the text will not fit even if shrinking is applied. Then the `meet` method will be applied. The `meet` method will decrease the font size until the text fits completely into the box. This case is shown in Figure 8.5e.


Fitting the text into the box with increased font size. You might want to fit the text so that it covers the whole width (or height) of the box but maintains its proportions. Using `fitmethod=meet` with a box larger than the text, the text will be increased until its width matches the box width. This case is illustrated in Figure 8.5f.

Completely fitting text into a box. We can further fit the text so that it completely fills the box. In this case, `fitmethod=entire` is used. However, this combination will rarely be used since the text will most probably be distorted (see Figure 8.5g).

Fitting text into a box with clipping. In another rare case you might want to fit the text in its original size and clip the text if it exceeds the box. In this case, `fitmethod=clip`

can be used. In Figure 8.5h the text is placed at the bottom left of a box which is not broad enough. The text will be clipped on the right.

Fig. 8.5 Fitting text into a box on the page subject to various options

Generated output	Option list for <code>PDF_fit_textline()</code>
a) 	<code>boxsize={100 35} position=center fontsize=12</code>
b) 	<code>boxsize={50 35} position=center fontsize=12</code>
c) 	<code>boxsize={100 35} position=center fontsize=12 fitmethod=auto</code>
d) 	<code>boxsize={58 35} position=center fontsize=12 fitmethod=auto</code>
e) 	<code>boxsize={30 35} position=center fontsize=12 fitmethod=auto</code>
f) 	<code>boxsize={100 35} position=center fontsize=12 fitmethod=meet</code>
g) 	<code>boxsize={100 35} position=center fontsize=12 fitmethod=entire</code>
h) 	<code>boxsize={50 35} position={left center} fontsize=12 fitmethod=clip</code>

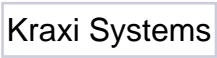
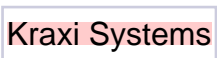
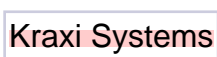
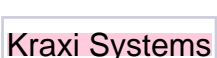
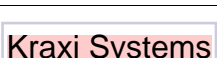
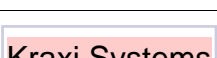
Vertically centering text. The text height in `PDF_fit_textline()` is the capheight, i.e. the height of the capital letter *H*, by default. If the text is positioned in the center of a box it will be vertically centered according to its capheight (see Figure 8.6a).

To specify another height for the text box we can use the Matchbox feature (see also Section 8.4. »Matchboxes«, page 237). The `matchbox` option of `PDF_fit_textline()` define the height of a Textline which is the capheight of the given font size, by default. The height of the matchbox is calculated according to its `boxheight` suboption. The `boxheight` suboption determines the extent of the text above and below the baseline. `matchbox={boxheight={capheight none}}` is the default setting, i.e. the top border of the matchbox will touch the capheight above the baseline, and the bottom border of the matchbox will not extend below the baseline.

To illustrate the size of the matchbox we will fill it with red color (see Figure 8.6b). Figure 8.6c vertically centers the text according to the *xheight* by defining a matchbox with a corresponding box height.

Figure 8.6d–f shows the matchbox (red) with various useful *boxheight* settings to determine the height of the text to be centered in the box (blue).

Fig. 8.6 Fitting text proportionally into a box according to different box heights

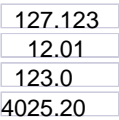
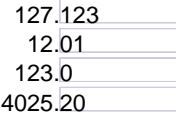
Generated output	Option list for PDF_fit_textline()
a) 	boxsize={80 20} position=center fitmethod=auto
b) 	boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={capheight none} fillcolor={rgb 1 0.8 0.8}}
c) 	boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={xheight none} fillcolor={rgb 1 0.8 0.8}}
d) 	boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={ascender none} fillcolor={rgb 1 0.8 0.8}}
e) 	boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={ascender descender} fillcolor={rgb 1 0.8 0.8}}
f) 	boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={fontsize none} fillcolor={rgb 1 0.8 0.8}}

8.1.4 Aligning Text at a Character

You might want to align text at a certain character, e.g. at the decimal point in a number. As shown in Figure 8.7a, the text is positioned at the center of the fitbox. Using *PDF_fit_textline()* with the *alignchar=.* option the numbers are aligned at the dot character.

You can omit the *position* option which places the dots in the center of the box. In this case, the default *position={left bottom}* will be used which places the dots at the reference point (see Figure 8.7b). In general, the alignment character will be placed with the lower right corner at the reference point.

Fig. 8.7 Aligning a Textline to the dot character


Generated output	Option list for PDF_fit_textline()
a) 	boxsize={70 8} position={center bottom} alignchar=.
b) 	boxsize={70 8} position={left bottom} alignchar=.

8.1.5 Placing a Stamp

Cookbook A full code sample can be found in the *Cookbook* topic `text_output/simple_stamp`.

As an alternative to rotated text, the stamp feature offers a convenient method for placing text diagonally in a box. The stamp function will automatically perform some sophisticated calculations to determine a suitable font size and rotation so that the text covers the box. To place a diagonal stamp, e.g. in the page background, use `PDF_fit_textline()` with the `stamp` option. With `stamp=ll2ur` the text will be placed from the lower left to the upper right corner of the fitbox. However, with `stamp=ul2lr` the text will be placed from the upper left to the lower right corner of the fitbox. As shown in Figure 8.8, `showborder=true` is used to illustrate the fitbox and the bounding box of the stamp.

Fig. 8.8 Fitting a text line like a stamp from the lower left to the upper right

Generated output	Option list for <code>PDF_fit_textline()</code>
	<code>fontsize=8 boxsize={160 50} stamp=ll2ur showborder=true</code>

8.1.6 Using Leaders

Leaders can be used to fill the space between the borders of the fitbox and the text. For example, dot leaders are often used as a visual aid between the entries in a table of contents and the corresponding page numbers.

Leaders in a table of contents. Using `PDF_fit_textline()` with the `leader` option and the `alignment={none right}` suboption, leaders are appended to the right of the text line, and repeated until the right border of the text box. There will be an equal distance between the rightmost leader and the right border, while the distance between the text and the leftmost leader may differ (see Figure 8.9a).

Cookbook A full code sample demonstrating the usage of dot leaders in a text line can be found in the *Cookbook* topic `text_output/leaders_in_textline`.

Cookbook A full code sample demonstrating the usage of dot leaders in a *Textflow* can be found in the *Cookbook* topic `text_output/dot_leaders_with_tabs`.

Leaders in a news ticker. In another use case you might want to create a news ticker effect. In this case we use a plus and a space character `»+ «` as leaders. The text line is placed in the center, and the leaders are printed before and after the text line (`alignment={left right}`). The left and right leaders are aligned to the left and right border, and might have a varying distance to the text (see Figure 8.9b).

Fig. 8.9 Fitting a text line using leaders

Generated output	Option list for PDF_fit_textline()
<div>a)</div> <div>Features of Giant Wing Description of Long Distance Glider..... Benefits of Cone Head Rocket.....</div>	<div>boxsize={200 10} leader={alignment={none right}}</div>
<div>b)</div> <div>+++++++ Giant Wing in purple!+++++++ ++ Long Distance Glider with sensational range! ++ +++++ Cone Head Rocket incredibly fast! +++++</div>	<div>boxsize={200 10} position={center bottom} leader={alignment={left right}} text={+ } }</div>

8.1.7 Text on a Path

Instead of placing text on a straight line you can also place text on an arbitrary path. PDFlib will place the individual characters along the path so that the text follows the curvature of the path. Use the *textpath* option of *PDF_fit_textline()* to create text on a path. The path must have been created earlier and is represented by a path handle. Path handles can be created by explicitly constructing a path with *PDF_add_path_point()* and related path object functions, or by retrieving a handle for the clipping path in an existing raster image. The following code fragment creates a simple path and places text on the path (see Figure 8.10):

```
/* Define the path in the origin */
path = p.add_path_point( -1,  0,  0, "move", "");
path = p.add_path_point(path, 100, 100, "control", "");
path = p.add_path_point(path, 200,  0, "circular", "");

/* Place text on the path */
p.fit_textline("Long Distance Glider with sensational range!", x, y,
    "textpath={path=" + path + "} position={center bottom}");

/* We also draw the path for demonstration purposes */
p.draw_path(path, x, y, "stroke");
```

Cookbook A full code sample can be found in the Cookbook topic `text_output/text_on_a_path`.

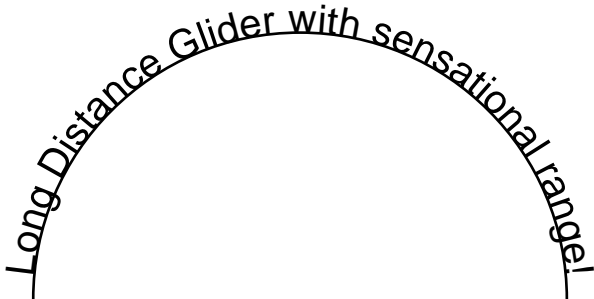


Fig. 8.10
Text on a path

Using an image clipping path for placing text. As an alternative to manually constructing a path object with the path functions you can extract the clipping path from an image and place text on the resulting path. The image must have been loaded with the *honorclippingpath* option, and the *clippingpathname* option must also be supplied to *PDF_load_image()* if the target path is not the image's default clipping path:

```
image = p.load_image("auto", "image.tif", "clippingpathname={path 1}");
```

```
/* create a path object from the image's clipping path */
path = (int) p.info_image(image, "clippingpath", "");
if (path == -1)
    throw new Exception("Error: clipping path not found!");
```

```
/* Place text on the path */
p.fit_textline("Long Distance Glider with sensational range!", x, y,
    "textpath={path=" + path + "} position={center bottom}");
```

Creating a gap between path and text. By default, PDFlib will place individual characters on the path, which means that there will no space between the glyphs and the path. If you want to create a gap between path and text you can simply increase the character boxes. This can easily be achieved with *boxheight* suboption of the *matchbox* option which specifies the vertical extension of the character boxes. The following option list takes the descenders into account (see Figure 8.10):

```
p.fit_textline("Long Distance Glider with sensational range!", x, y,
    "textpath={path=" + path + "} position={center bottom} " +
    "matchbox={boxheight={capheight descender}}");
```

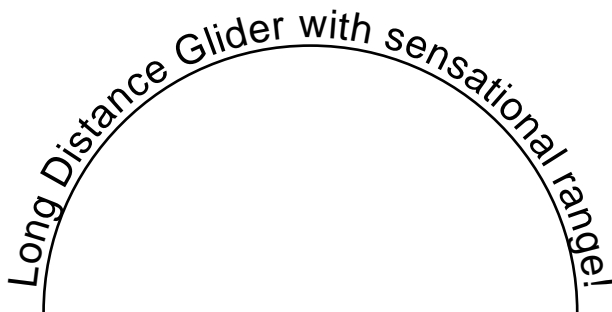


Fig. 8.11
Text on a path with an additional
gap between text and path

8.2 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, columns, 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 part 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 8.12 and Figure 8.13 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.

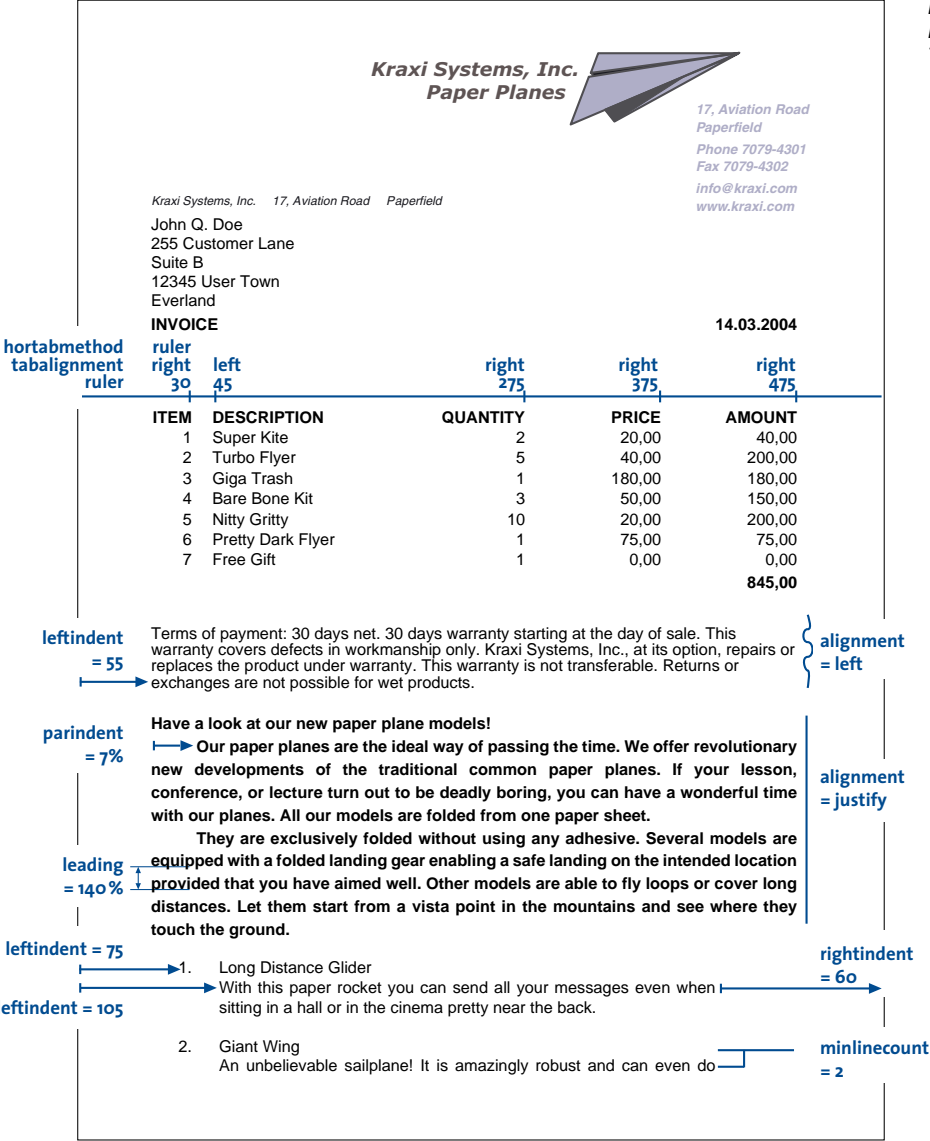


Fig. 8.12
Formatting
Textflows

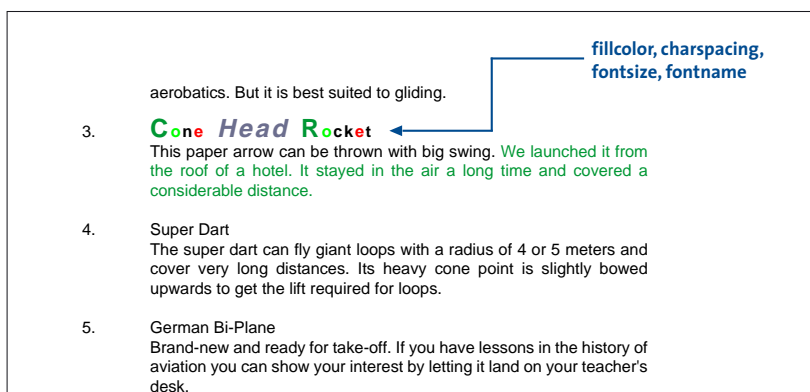


Fig. 8.13
Formatting
Textflows

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_add_textflow()` accepts portions of text and corresponding formatting options, creates a Textflow object, and returns a handle. As an alternative, the function `PDF_create_textflow()` analyzes the complete text in a single call, where the text may contain inline options for formatting control. These functions do 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 functions `PDF_add/create_textflow()` for creating Textflows support 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 when using `PDF_create_textflow()`. `PDF_info_textflow()` can be used to query formatting results and many other Textflow details. We will discuss Textflow placement using some common application examples. A complete list of Textflow options can be found in the *PDFlib API Reference*.

Many of the options supported by `PDF_add/create_textflow()` are identical to those of `PDF_fit_textline()`. It is therefore recommended to familiarize yourself with the examples in Section 8.1, »Placing and Fitting Textlines«, page 193. In the below sections we will focus on options related to multi-line text.

Cookbook Code samples regarding text output issues can be found in the `text_output` category of the *PDFlib Cookbook*.

8.2.1 Placing Textflows in the Fitbox

The fitbox for Textflow is the area where text will be placed. It is defined as the rectangular area specified with the `llx`, `lly`, `urx`, `ury` parameters of `PDF_fit_textflow()`.

Placing text in a single fitbox. Let's start with an easy example. The following code fragment uses two calls to `PDF_add_textflow()` to assemble a piece of bold text and a

piece of normal text. Font, font size, and encoding are specified explicitly. In the first call to `PDF_add_textflow()`, -1 is supplied, and the Textflow handle will be returned to be used in subsequent calls to `PDF_add_textflow()`, if required. `text1` and `text2` are assumed to contain the actual text to be printed.

With `PDF_fit_textflow()`, the resulting Textflow is placed in a fitbox on the page using default formatting options.

```
/* Add text with bold font */
tf = p.add_textflow(-1, text1, "fontname=Helvetica-Bold fontsize=9 encoding=unicode");
if (tf == -1)
    throw new Exception("Error: " + p.get_errmsg());

/* Add text with normal font */
tf = p.add_textflow(tf, text2, "fontname=Helvetica fontsize=9 encoding=unicode");
if (tf == -1)
    throw new Exception("Error: " + p.get_errmsg());

/* Place all text */
result = p.fit_textflow(tf, left_x, left_y, right_x, right_y, "");
if (!result.equals("_stop"))
    { /* ... */ }

p.delete_textflow(tf);
```

Placing text in two fitboxes on multiple pages. If the text placed with `PDF_fit_textflow()` doesn't completely fit into the fitbox, the output will be interrupted and the function will return the string `_boxfull`. PDFlib will remember the amount of text already placed, and will continue with the remainder of the text when the function is called again. In addition, it may be necessary to create a new page. The following code fragment demonstrates how to place a Textflow in two fitboxes per page on one or more pages until the text has been placed completely (see Figure 8.14).

Cookbook A full code sample can be found in the Cookbook topic text_output/starter_textflow.

```
/* Loop until all of the text is placed; create new pages as long as more text needs
 * to be placed. Two columns will be created on all pages.
 */
```

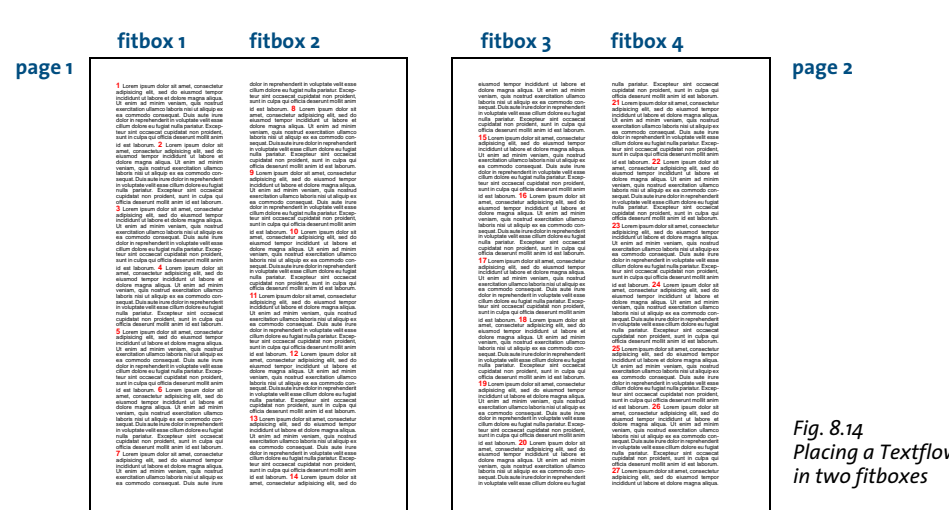


Fig. 8.14
Placing a Textflow
in two fitboxes

```

do
{
    String optlist = "verticalalign=justify linespreadlimit=120%";

    p.begin_page_ext(0, 0, "width=a4.width height=a4.height");

    /* Fill the first column */
    result = p.fit_textflow(tf, llx1, lly1, urx1, ury1, optlist);

    /* Fill the second column if we have more text*/
    if (!result.equals("_stop"))
        result = p.fit_textflow(tf, llx2, lly2, urx2, ury2, optlist);

    p.end_page_ext("");

    /* "_boxfull" means we must continue because there is more text;
     * "_nextpage" is interpreted as "start new column"
     */
} while (result.equals("_boxfull") || result.equals("_nextpage"));

/* Check for errors */
if (!result.equals("_stop"))
{
    /* "_boxempty" happens if the box is very small and doesn't hold any text at all.
     */
    if (result.equals( "_boxempty"))
        throw new Exception("Error: " + p.get_errmsg());
    else
    {
        /* Any other return value is a user exit caused by the "return" option;
         * this requires dedicated code to deal with.
         */
    }
}
p.delete_textflow(tf);

```

8.2.2 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_add_textflow()*. For example, we can indent the text 15 units from the left and 10 units from the right margin. The first line of each paragraph should be indented by an additional 20 units. The text should be justified against both margins, and the leading increased to 140%. Finally, we'll reduce the font size to 8 points. To achieve this, extend the option list for *PDF_add_textflow()* as follows (see Figure 8.15):

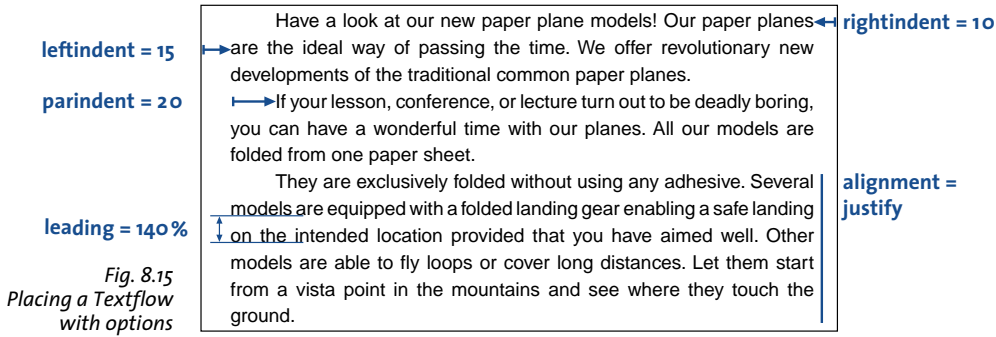
```

String optlist =
    "leftindent=15 rightindent=10 parindent=20 alignment=justify " +
    "leading=140% fontname=Helvetica fontsize=8 encoding=unicode";

```

8.2.3 Inline Option Lists and Macros

The text in Figure 8.15 is not yet perfect. The headline »Have a look at our new paper plane models!« should sit on a line of its own, should use a larger font, and should be centered. There are several ways to achieve this.

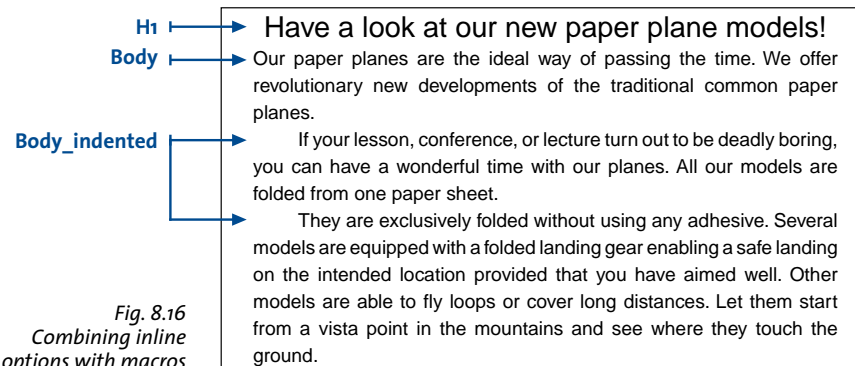


Inline option lists for PDF_create_textflow(). Up to now we provided formatting options in an option list supplied directly to the function. In order to continue the same way we would have to split the text, and place it in two separate calls, one for the headline and another one for the remaining text. However, in certain situations, e.g. with lots of formatting changes, this method might be pretty cumbersome.

For this reason, `PDF_create_textflow()` can be used instead of `PDF_add_textflow()`. `PDF_create_textflow()` interprets text and so-called inline options which are embedded directly 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.

Note Inline option lists are colorized 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. 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. 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.

Symbol characters and inline option lists. Symbolic characters can be used for Textflow even in combination with inline options lists. The code for the character which introduces an inline option list (by default: '⟨' U+003C) will not be interpreted as a symbol code within text for a font with *encoding=builtin*. In order to select the symbol glyph with the same code, the same workarounds can be used which are available for text fonts, i.e. by redefining the start character with the *begoptlistchar* option or by using the *textlen* option to specify the number of symbolic glyphs. Note that character references (e.g. <) can not be used as a workaround for the reasons discussed above.

Macros. 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 differently and occurs multiply in longer Textflows. In order to avoid starting each paragraph with the corresponding inline options, we can combine these in macros, and refer to the macros in the text via their names. As shown in Figure 8.16 we 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.

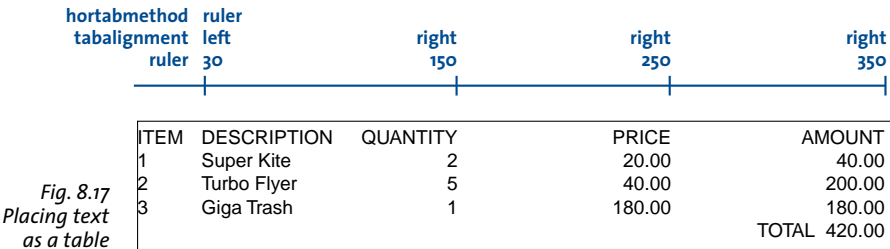
8.2.4 Tab Stops

In the next example we will place a simple 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
```

To place that simple table use the following option list in `PDF_add/create_textflow()`. The *ruler* option defines the tab positions, *tabalignment* specifies the alignment of tab stops, and *hortabmethod* specifies the method used to process tab stops (the result can be seen in Figure 8.17):

```
String optlist =
    "ruler      ={30    150  250  350} " +
    "tabalignment={left right right right} " +
    "hortabmethod=ruler leading=120% fontname=Helvetica fontsize=9 encoding=winansi";
```



Cookbook A full code sample can be found in the Cookbook topic `text_output/tabstops_in_textflow`.

Note PDFlib's table feature is recommended for creating complex tables (see Section 8.3, »Table Formatting«, page 221).

8.2.5 Numbered Lists and Paragraph Spacing

The following example demonstrates how to format a numbered list using the inline option `leftindent` (see Figure 8.18):

```
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.
```

Cookbook Full code samples for bulleted and numbered lists can be found in the Cookbook topics `text_output/bulleted_list` and `text_output/numbered_list`.

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. ↵
2. → Giant Wing: An unbelievable sailplane! It is amazingly robust and can even do
aerobatics. But it is best suited to gliding. ↵
```

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. 8.18
Numbered list



`leftindent = &indent` 
`parindent = - &indent` 

Fig. 8.19
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.

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 8.19 shows the *parindent* and *leftindent* options at work.

Setting the distance between two paragraphs. In many cases more distance between adjacent paragraphs is desired than between the lines within a paragraph. This can be achieved by inserting an extra empty line (which can be created with the *nextline* option), and specifying a suitable leading value for this empty line. This value is the distance between the baseline of the last line of the previous paragraph and the baseline of the empty line. The following example will create 80% additional space between the two paragraphs (where 100% equals the most recently set value of the font size):

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.
`<nextline leading=80><nextparagraph leading=100>`2. → Giant Wing: An unbelievable sailplane! It is amazingly robust and can even do aerobatics. But it is best suited to gliding.

Cookbook A full code sample can be found in the Cookbook topic `text_output/distance_between_paragraphs`.

8.2.6 Control Characters and Character Mapping

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 8.1 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>`

Table 8.1 Control characters and their meaning in Textflows

Unicode character	entity name	equiv. Text-flow option	meaning within Textflows
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, and may change its width according to formatting options such as justification.
U+202F	NNBSP, nnbsp	(none)	(narrow no-break space) fixed-width space character which will not break lines, and will not change its width according to formatting options
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, vartab 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 nextline; in addition, the parindent option will affect the next line.
U+000C	FF, formfeed	return	PDF_fit_textflow() will stop, and return the string _nextpage.

Mapping/removing characters or sequences of characters. The *charmapping* option can be used to map or remove some characters in the text to others. Let's start with an easy case where we will map all tabs in the text to space characters. The *charmapping* option to achieve this looks as follows:

```
charmapping={hortab space}
```

This command uses the symbolic character names *hortab* and *space*. You can find a list of all known character names in the *PDFlib API Reference*. To achieve multiple mappings at once you can use the following command which will replace all tabs and line break combinations with space characters:

```
charmapping={hortab space CRLF space LF space CR space}
```

The following command removes all soft hyphens:

```
charmapping={shy {shy 0}}
```

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}}
```

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. 8.20

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

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 */  
String optlist =  
    "fontname=Helvetica fontsize=9 encoding=winansi alignment=justify " +  
    "charmapping {CRLF {space -1}}"  
  
/* place textflow in fitbox */  
textflow = p.add_textflow(-1, text, optlist);  
if (textflow == -1)  
    throw new Exception("Error: " + p.get_errmsg());  
  
result = p.fit_textflow(textflow, left_x, left_y, right_x, right_y, "");  
if (!result.equals("_stop"))  
    { /* ... */ }  
  
p.delete_textflow(textflow);
```

Figure 8.20 shows Textflow output with the unmodified text and the improved version with the *charmapping* option.

8.2.7 Hyphenation

PDFlib does not automatically hyphenate text, but can break words at hyphenation opportunities which are explicitly marked 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 can be used if an encoding does not contain the soft hyphen character (e.g. *macroman*): `­`

`U+002D` will be used as hyphenation character. 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 word spacing 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 8.21 shows the generated text output with default 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 8.22, 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
```

8.2.8 Controlling the standard Linebreak Algorithm

PDFlib implements a sophisticated line-breaking algorithm. Table 8.2 lists Textflow options which control the line-breaking algorithm.

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

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. 8.21
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. 8.22
Justified text without soft hyphens, using default settings and a wide fitbox.

opportunity. In order to find out whether a quotation mark starts or ends a sequence, pairs of quotation marks are examined.

Table 8.2 Options for controlling the line-breaking algorithm

option	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 max-spacing options. Default: auto <ul style="list-style-type: none">auto The following methods are applied in order: shrink, spread, nofit, split.clip Same as nofit (see below), except that the long part at the right edge of the fitbox (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.
advanced-linebreak	(Boolean) Enable the advanced line breaking algorithm which is required for complex scripts. This is required for linebreaking in scripts which do not use space characters for designating word boundaries, e.g. Thai. The options locale and script will be honored. Default: false
avoidbreak	(Boolean) If true, avoid any line breaks until avoidbreak is reset to false. Default: false
charclass	(List of pairs, where the first element in each pair is a keyword, and the second element is either a unichar or a list of unichars) The specified unichars will be classified by the specified keyword to determine the line breaking behaviour of those character(s): <ul style="list-style-type: none">letter behave like a letter (e.g. a B)punct behave like a punctuation character (e.g. + / ; :)open behave like an open parenthesis (e.g. [)close behave like a close parenthesis (e.g.])default reset all character classes to PDFlib's builtin defaults Example: charclass={ close » open « letter {/ : =} punct & }

Table 8.2 Options for controlling the line-breaking algorithm

option	explanation
hyphenchar	(Unichar or keyword) Unicode value of the character which replaces a soft hyphen at line breaks. The value 0 and the keyword none completely suppress hyphens. Default: U+00AD (SOFT HYPHEN) if available in the font, U+002D (HYPHEN-MINUS) otherwise
locale	<p>(Keyword) The locale which will be used for localized linebreaking methods if advancedlinebreak=true. The keywords consists of one or more components, where the optional components are separated by an underscore character '_' (the syntax slightly differs from NLS/POSIX locale IDs):</p> <ul style="list-style-type: none">▶ A required two- or three-letter lowercase language code according to ISO 639-2 (see www.loc.gov/standards/iso639-2), e.g. en, (English), de (German), ja (Japanese). This differs from the language option.▶ An optional four-letter script code according to ISO 15924 (see www.unicode.org/iso15924/iso15924-codes.html), e.g. Hira (Hiragana), Hebr (Hebrew), Arab (Arabic), Thai (Thai).▶ An optional two-letter uppercase country code according to ISO 3166 (see www.iso.org/iso/country_codes/iso_3166_code_lists), e.g. DE (Germany), CH (Switzerland), GB (United Kingdom) <p>The keyword _none specifies that no locale-specific processing will be done.</p> <p>Specifying a locale is required for advanced line breaking for some scripts, e.g. Thai. Default: _none</p> <p>Examples: Thai, de_DE, en_US, en_GB</p>
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 horizscaling 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

An inline option list generally does not create a line break opportunity in order to allow option changes within words. However, when an option list is surrounded by space characters there is a line break opportunity at the beginning of the option list. If a line break occurs at the option list and *alignment=justify*, the spaces preceding the option list will be discarded. The spaces after the option list will be retained, and will appear at the beginning of the next line.

Preventing linebreaks. You can use the *charclass* option to prevent Textflow from breaking a line after specific characters. For example, the following option will prevent line breaks immediately after the / character:

```
charclass={letter /}
```

In order to prevent a sequence of text from being broken across lines you can bracket it with *avoidbreak...noavoidbreak*.

Cookbook A full code sample can be found in the Cookbook topic `text_output/avoid_linebreaking`.

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

Fig. 8.23 Justified text in a narrow fitbox with default settings

decrease the distance between words (minspacing option)

compress the line (shrink method, shrinklimit option)

force hyphenation (split method)

increase the distance between words (spread method, maxspacing option)

Formatting CJK text. The textflow engine is prepared to deal with CJK text, and properly treats CJK characters as ideographic glyphs as per the Unicode standard. As a result, CJK text will never be hyphenated. For improved formatting the following options are recommended when using Textflow with CJK text; they will disable hyphenation for inserted Latin text and create evenly spaced text output:

```
hyphenchar=none  
alignment=justify  
shrinklimit=100%  
spreadlimit=100%
```

Vertical writing mode is not supported in Textflow.

Justified text in a narrow fitbox. The narrower the fitbox, the more important are the options for controlling justified text. Figure 8.23 demonstrates the results of the various methods for justifying text in a narrow fitbox. The option settings in Figure 8.23 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 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 8.24a shows how to avoid forced hyphenation by compressing text down to *shrinklimit=50%*.

Fig. 8.24 Options for justified text in a narrow fitbox

Generated output	Option list for <code>PDF_fit_textflow()</code>
<div>a)</div> <div>passing the time. We offer revolutionary brand new developments of the traditional common paper planes. If your lesson, conference, or lecture turn out to</div> <div>←</div> <div>←</div>	<code>alignment=justify shrinklimit=50%</code>
<div>b)</div> <div>Our paper planes are the ideal way of passing the time. We offer revolutionary brand new developments of the</div> <div>←</div>	<code>alignment=justify spreadlimit=5</code>
<div>c)</div> <div>ments of the traditional common paper planes. If your lesson, conference, or lecture turn out to be deadly boring, you can have</div> <div>←</div>	<code>alignment=justify nofitlimit=50</code>

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 8.24b demonstrates a very large maximum character distance of 5 units using *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 8.24c shows the generated text output with a minimum text width of 50%.

8.2.9 Advanced script-specific Line Breaking

PDFlib implements an additional line breaking algorithm on top of the standard line breaking algorithm. This advanced line breaking algorithm is required for some scripts, and improves line breaking behavior for some other script/locale combinations even if it is not required. It can be enabled with the *advancedlinebreak* option. Since line breaking depends on the language of the text, the advanced line breaking algorithm honors the *script* option (see Table 6.2) and the *locale* option (see PDFlib API Reference). Advanced line breaking determines proper line break opportunities in the following situations:

- ▶ For scripts in which line breaking does not rely on the presence of space characters in the text, e.g. Thai. The following Textflow option list enables advanced line breaking for Thai:

```
<advancedlinebreak script=thai locale=tha>
```

- ▶ In script/locale combinations which require specific treatment of certain punctuation characters, e.g. the « and » guillemet characters used as quotation marks in French text. The following Textflow option list enables advanced line breaking for

French text. As a result, the guillemet characters surrounding a word will not be split apart from the word at the end of a line:

```
<advancedlinebreak script=latn locale=fr>
```

Note that the *locale* Textflow option is different from the *language* text option (see Table 6.3): although the *locale* option can start with the same three-letter language identifier, it can optionally contain one or two additional parts. However, these will rarely be required for PDFlib.

8.2.10 Wrapping Text around Paths and Images

The wrapping feature can be used to fill arbitrary shapes with text or wrap text around a path. By means of matchboxes, explicit rectangles/polygons/circles/curves or path objects you can specify wrapping areas for the Textflow. If an image contains an integrated clipping path you can wrap text around the image clipping path automatically.

Wrapping text around an image with matchbox. In the first example we will place an image within the Textflow and run the text around the whole image. First the image is loaded and placed in the box at the desired position. To refer to the image by name later, define a matchbox called *img* when fitting the image, and specify a margin of 5 units with the option list *matchbox={name=img margin=-5}* as follows:

```
result = p.fit_image(image, 50, 35,  
    "boxsize={80 46} fitmethod=meet position=center matchbox={name=img margin=-5}");
```

The Textflow is added. Then we place it using the *wrap* option with the image's matchbox *img* as the area to run around as follows (see Figure 8.25):

```
result = p.fit_textflow(textflow, left_x, left_y, right_x, right_y,  
    "wrap={usematchboxes={{img}}}")
```

Before placing the text you can fit more images using the same matchbox name. In this case the text will run around all images.

Cookbook A full code sample can be found in the Cookbook topic `text_output/wrap_text_around_images`.

Wrapping text around arbitrary paths. You can create a path object (see Section 3.2.3, »Direct Paths and Path Objects«, page 67) and use it as a wrap shape. The following fragment constructs a path object with a simple shape (a circle) and supplies it to the *wrap* option of *PDF_fit_textflow()*. The reference point for placing the path is expressed as percentage of the fitbox's width and height:

Fig. 8.25

Wrapping text around an image with matchbox

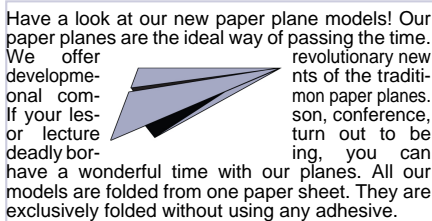
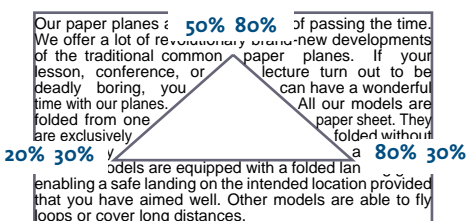


Fig. 8.26

Wrapping text around a triangular shape



```

path = p.add_path_point( -1, 0, 100, "move", "");
path = p.add_path_point(path, 200, 100, "control", "");
path = p.add_path_point(path, 0, 100, "circular", "");

/* Visualize the path if desired */
p.draw_path(path, x, y, "stroke");

result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
    "wrap={paths={" +
        "{path=" + path + " refpoint={100% 50%} }" +
        "}}");

p.delete_path(path);

```

Use the *inversefill* option to wrap text inside the path instead of wrapping the text around the path (i.e. the path serves as text container instead of creating a hole in the Textflow):

```

result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
    "wrap={inversefill paths={" +
        "{path=" + path + " refpoint={100% 50%} }" +
        "}}");

```

Wrapping text around an image clipping path. TIFF and JPEG images can contain an integrated clipping path. The path must have been created in an image processing application and will be evaluated by PDFlib. If a default clipping path is found in the image it will be used, but you can specify any other clipping path in the image with the *clippingpathname* option of *PDF_load_image()*. If the image has been loaded with a clipping path you can extract the path and supply it to the *wrap* option *PDF_fit_textflow()* as above. We also supply the *scale* option to enlarge the imported image clipping path:

```

image = p.load_image("auto", "image.tif", "clippingpathname={path 1}");

/* Create a path object from the image's clipping path */
path = (int) p.info_image(image, "clippingpath", "");
if (path == -1)
    throw new Exception("Error: clipping path not found!");

result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
    "wrap={paths={{path=" + path + " refpoint={50% 50%} scale=2}}});

p.delete_path(path);

```

Placing an image and wrapping text around it. While the previous section used only the clipping path of an image (but not the image itself), let's now place the image inside the fitbox of the Textflow and wrap the text around it. In order to achieve this we must again load the image with the *clippingpathname* option and place it on the page with *PDF_fit_image()*. In order to create the proper path object for wrapping the Textflow we call *PDF_info_image()* with the same option list as *PDF_fit_image()*. Finally, the reference point (the *x/y* parameters of *PDF_fit_image()*) must be supplied to the *refpoint* suboption of the *paths* suboption of the *wrap* option:

```

image = p.load_image("auto", "image.tif", "clippingpathname={path 1}");

/* Place the image on the page with some fitting options */

```

```
String imageoptlist = "scale=2";
p.fit_image(image, x, y, imageoptlist);

/* Create a path object from the image's clipping path, using the same option list */
path = (int) p.info_image(image, "clippingpath", imageoptlist);
if (path == -1)
    throw new Exception("Error: clipping path not found!");

result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
    "wrap={paths={{path=" + path + " refpoint={" + x + " " + y + " }}}}");

p.delete_path(path);
```

You can supply the same *wrap* option in multiple calls to *PDF_fit_textflow()*. This is useful if the placed image overlaps multiple Textflow fitboxes, e.g. for multi-column text.

Wrapping text around non-rectangular shapes. As an alternative to creating a path object as wrap shape you can specify path elements directly in Textflow options.

In addition to wrapping text around a rectangle specified by a matchbox you can define arbitrary graphical elements as wrapping shapes. For example, the following option list will wrap the text around a triangular shape (see Figure 8.26):

```
wrap={ polygons={ {50% 80% 20% 30% 80% 30% 50% 80%} } }
```

Note that the *showborder=true* option has been used to illustrate the margins of the shapes. The *wrap* option can contain multiple shapes. The following option list will wrap the text around two triangle shapes:

```
wrap={ polygons={ {50% 80% 20% 30% 80% 30% 50% 80%}
    {20% 90% 10% 70% 30% 70% 20% 90%} } }
```

Instead of percentages (relative coordinates within the fitbox) absolute coordinates on the page can be used.

Note It is recommended to set fixedleading=true when using shapes with segments which are neither horizontally nor vertically oriented.

Cookbook A full code sample can be found in the Cookbook topic text_output/wrap_text_around_polygons.

Filling non-rectangular shapes. The wrap feature can also be used to place the contents of a Textflow in arbitrarily shaped areas. This is achieved with the *addfitbox* or *inversefill* suboptions of the *wrap* option. Instead of wrapping the text around the specified shapes the text will be placed within one or more shapes. The following option list can be used to flow text into a rhombus shape, where the coordinates are provided as percentages of the fitbox rectangle (see Figure 8.27):

```
wrap={ addfitbox polygons={ {50% 100% 10% 50% 50% 0% 90% 50% 50% 100%} } }
```

Note that the *showborder=true* option has been again used to illustrate the margins of the shape. Without the *addfitbox* option the rhombus shape will remain empty and the text will be wrapped around it.

Fig. 8.27
Filling a rhombus
shape with text

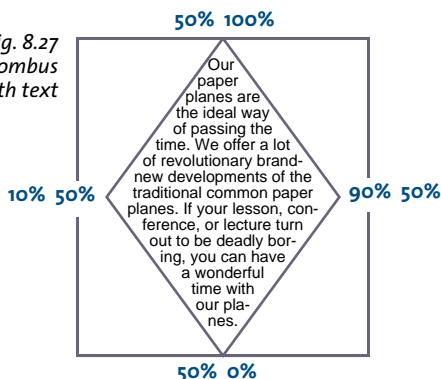
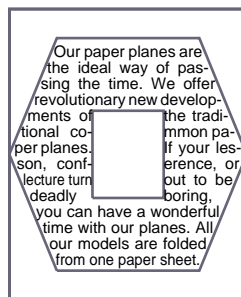


Fig. 8.28
Filling overlapping shapes



Filling overlapping shapes. In the next example we will fill a shape comprising two overlapping polygons, namely a hexagon with a rectangle inside. Using the *addfitbox* option the fitbox itself will be excluded from being filled, and the polygons in the subsequent list will be filled except in the overlapping area (see Figure 8.28):

```
wrap={ addfitbox polygons=
  { {20% 10% 80% 10% 100% 50% 80% 90% 20% 90% 0% 50% 20% 10%}
    {35% 35% 65% 35% 65% 65% 35% 65% 35% 35%} } }
```

Without the *addfitbox* option you will get the opposite effect: the previously filled area will remain empty, and the previously empty areas will be filled with text.

Cookbook A full code sample can be found in the *Cookbook* topic `text_output/fill_polygons_with_text`.

8.3 Table Formatting

The table formatting feature can be used to automatically format complex tables. Table cells may contain single- or multi-line text, images or PDF graphics. Tables are not restricted to a single fitbox, but can span multiple pages.

Cookbook Code samples regarding table issues can be found in the tables category of the PDFlib Cookbook.

General aspects of a table. The description of the table formatter is based on the following concepts and terms (see Figure 8.29):

- ▶ A *table* is a virtual object with a rectangular outline. It is comprised of horizontal *rows* and vertical *columns*.
- ▶ A *simple cell* is a rectangular area within a table, defined as the intersection of a row and a column. A *spanning cell* spans more than one column, more than one row, or both. The term *cell* will be used to designate both simple and spanning cells.
- ▶ The complete table may fit into one fitbox, or several fitboxes may be required. The rows of the table which are placed in one fitbox constitute a *table instance*. Each call to `PDF_fit_table()` will place one *table instance* in one fitbox (see Section 8.3.5, »Table Instances«, page 231).
- ▶ The *header* or *footer* is a group of one or more rows at the beginning or end of the table which are repeated at the top or bottom of each table instance. Rows which are neither part of the header nor footer are called *body rows*.


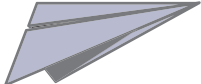

Our Paper Plane Models		
1 Giant Wing		 Amazingly robust!
Material	Offset print paper 220g/sqm	
Benefit	It is amazingly robust and can even do aerobatics. But it is best suited to gliding.	
2 Long Distance Glider		
Material	Drawing paper 180g/sqm	
Benefit	With this paper rocket you can send all your messages even when sitting in the cinema pretty near the back.	
3 Cone Head Rocket		 With big swing!
Material	Kent paper 200g/sqm	
Benefit	This paper arrow can be thrown with big swing. It stays in the air a long time.	

Fig. 8.29
Sample table

Fig. 8.29
Sample table

As an example, all aspects of creating the table in Figure 8.29 will be explained. A complete description of the table formatting options can be found in the *PDFlib API Reference*. Creating a table starts by defining the contents and visual properties of each table cell with `PDF_add_table_cell()`. Then you place the table using one or more calls to `PDF_fit_table()`.

When placing the table the size of its fitbox and the ruling and shading of table rows or columns can be specified. Use the Matchbox feature for details such as cell-specific shading (see Section 8.4, »Matchboxes«, page 237, for more information).

In this section the most important options for defining the table cells and fitting the table will be discussed. All examples demonstrate the relevant calls of `PDF_add_table_cell()` and `PDF_fit_table()` only, assuming that the required font has already been loaded.

Note Table processing is independent from the current graphics state. Table cells can be defined in document scope while the actual table placement must be done in page scope.

Cookbook A full code sample can be found in the Cookbook topic `tables/starter_table`.

8.3.1 Placing a Simple Table

Before we describe the table concepts in more detail, we will demonstrate a simple example for creating a table. The table contains six cells which are arranged in three rows and two columns. Four cells contain text lines, and one cell contains a multi-line Textflow. All cell contents are horizontally aligned to the left, and vertically aligned to the center with a margin of 4 points.

To create the table we first prepare the option list for the text line cells by defining the required options `font` and `fontsize` and a position of `{left center}` in the `fittextline` sub-option list. In addition, we define cell margins of 4 points. Then we add the text line cells one after the other in their respective column and row, with the actual text supplied directly in the call to `PDF_add_table_cell()`.

In the next step we create a Textflow, use the Textflow handle to assemble the option list for the Textflow table cell, and add that cell to the table.

Finally we place the table with `PDF_fit_table()` while visualizing the table frame and cell borders with black lines. Since we didn't supply any column widths, they will be calculated automatically from the supplied text lines plus the margins.

Cookbook A full code sample can be found in the Cookbook topic `tables/vertical_text_alignment`.

The following code fragment shows how to create the simple table. The result is shown in Figure 8.30a.

```
/* Text for filling a table cell with multi-line Textflow */
String tf_text = "It is amazingly robust and can even do aerobatics. " +
                 "But it is best suited to gliding.";

/* Define the column widths of the first and the second column */
int c1 = 80, c2 = 120;

/* Define the lower left and upper right corners of the table instance (fitbox) */
double llx=100, lly=500, urx=300, ury=600;

/* Load the font */
font = p.load_font("Helvetica", "unicode", "");
if (font == -1)
    throw new Exception("Error: " + p.get_errmsg());

/* Define the option list for the text line cells placed in the first column */
optlist = "fittextline={position={left center} font=" + font + " fontsize=8} margin=4" +
          " colwidth=" + c1;

/* Add a text line cell in column 1 row 1 */
```

```

tbl = p.add_table_cell(tbl, 1, 1, "Our Paper Planes", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());

/* Add a text line cell in column 1 row 2 */
tbl = p.add_table_cell(tbl, 1, 2, "Material", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());

/* Add a text line cell in column 1 row 3 */
tbl = p.add_table_cell(tbl, 1, 3, "Benefit", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());

/* Define the option list for a text line placed in the second column */
optlist = "fittextline={position={left center} font=" + font + " fontsize=8} " +
    "colwidth=" + c2 + " margin=4";

/* Add a text line cell in column 2 row 2 */
tbl = p.add_table_cell(tbl, 2, 2, "Offset print paper 220g/sqm", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());

/* Add a Textflow */
optlist = "font=" + font + " fontsize=8 leading=110%";
tf = p.add_textflow(-1, tf_text, optlist);

/* Define the option list for the Textflow cell using the handle retrieved above */
optlist = "textflow=" + tf + " margin=4 colwidth=" + c2;

/* Add the Textflow table cell in column 2 row 3 */
tbl = p.add_table_cell(tbl, 2, 3, "", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());

p.begin_page_ext(0, 0, "width=200 height=100");

/* Define the option list for fitting the table with table frame and cell ruling */
optlist = "stroke={{line=frame linewidth=0.8} {line=other linewidth=0.3}}";

/* Place the table instance */
result = p.fit_table(tbl, llx, lly, urx, ury, optlist);

/* Check the result; "_stop" means all is ok. */
if (!result.equals("_stop")) {
    if (result.equals("_error"))
        throw new Exception("Error: " + p.get_errmsg());
    else {
        /* Any other return value requires dedicated code to deal with */
    }
}

p.end_page_ext("");

/* This will also delete Textflow handles used in the table */
p.delete_table(tbl, "");

```

Fine-tuning the vertical alignment of cell contents. When we vertically center contents of various types in the table cells, they will be positioned with varying distance from the borders. In Figure 8.30a, the four text line cells have been placed with the following option list:

```
optlist = "fittextline={position={left center} font=" + font +  
          " fontsize=8} colwidth=80 margin=4";
```

The Textflow cell is added without any special options. Since we vertically centered the text lines, the *Benefit* line will move down with the height of the Textflow.

Fig. 8.30 Aligning text lines and Textflow in table cells

Generated output	
a)	Our Paper Planes
	Material Offset print paper 220g/sqm
	Benefit It is amazingly robust and can even do aerobatics. But it is best suited to gliding.
b)	Our Paper Planes
	Material Offset print paper 220g/sqm
	Benefit It is amazingly robust and can even do aerobatics. But it is best suited to gliding.

As shown in Figure 8.30b, we want all cell contents to have the same *vertical* distance from the cell borders regardless of whether they are Textflows or text lines. To accomplish this we first prepare the option list for the text lines. We define a fixed row height of 14 points, and the position of the text line to be on the top left with a margin of 4 points.

The *fontsize=8* option which we supplied before doesn't exactly represent the letter height but adds some space below and above. However, the height of an uppercase letter is exactly represented by the *capheight* value of the font. For this reason we use *fontsize={capheight=6}* which will approximately result in a font size of 8 points and (along with *margin=4*), will sum up to an overall height of 14 points corresponding to the *rowheight* option. The complete option list of *PDF_add_table_cell()* for our text line cells looks as follows:

```
/* option list for the text line cells */  
optlist = "fittextline={position={left top} font=" + font +  
          " fontsize={capheight=6}} rowheight=14 colwidth=80 margin=4";
```

To add the Textflow we use *fontsize={capheight=6}* which will approximately result in a font size of 8 points and (along with *margin=4*), will sum up to an overall height of 14 points as for the text lines above.

```
/* option list for adding the Textflow */  
optlist = "font=" + font + " fontsize={capheight=6} leading=110%";
```


In addition, we want the baseline of the *Benefit* text aligned with the first line of the Textflow. At the same time, the *Benefit* text should have the same distance from the top cell border as the *Material* text. To avoid any space from the top we add the Textflow cell using `fittextflow={firstlinedist=capheight}`. Then we add a margin of 4 points, the same as for the text lines:

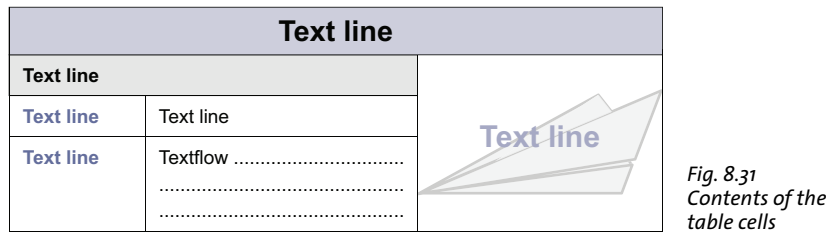
```
/* option list for adding the Textflow cell */
optlist = "textflow=" + tf + " fittextflow={firstlinedist=capheight} "
        "colwidth=120 margin=4";
```

Cookbook A full code sample can be found in the *Cookbook* topic `tables/vertical_text_alignment`.

8.3.2 Contents of a Table Cell

When adding cells to a table with `PDF_add_table_cell()`, you can specify various kinds of cell contents. Table cells can contain one or more content types at the same time. Additional ruling and shading is available, as well as matchboxes which can be used for placing additional content in a table cell.

For example, the cells of the paper plane table contain the elements illustrated in Figure 8.31.



Single-line text with Textlines. The text is supplied in the `text` parameter of `PDF_add_table_cell()`. In the `fittextline` option all formatting options of `PDF_fit_textline()` can be specified. The default fit method is `fitmethod=nofit`. The cell will be enlarged if the text doesn't completely fit into the cell. To avoid this, use `fitmethod=auto` to shrink the text subject to the `shrinklimit` option. If no row height is specified the formatter assumes twice the text size as height of the table cell (more precisely: twice the `boxheight`, which has the default value `{capheight none}` unless specified otherwise). The same applies to the row width for rotated text.

Multi-line text with Textflow. The Textflow must have been prepared outside the table functions and created with `PDF_create_textflow()` or `PDF_add_textflow()` before calling `PDF_add_table_cell()`. The Textflow handle is supplied in the `textflow` option. In the `fittextflow` option all formatting options of `PDF_fit_textflow()` can be specified.

The default fit method is `fitmethod=clip`. This means: First it is attempted to completely fit the text into the cell. If the cell is not large enough its height will be increased. If the text do not fit anyway it will be clipped at the bottom. To avoid this, use `fitmethod=auto` to shrink the text subject to the `minfontsize` option.

When the cell is too narrow the Textflow could be forced to split single words at undesired positions. If the `checkwordsplitting` option is `true` the cell width will be enlarged until no word splitting occurs any more.

Images and templates. Images must be loaded with *PDF_load_image()* before calling *PDF_add_table_cell()*. Templates must be created with *PDF_begin_template_ext()*. The image or template handle is supplied in the *image* option. In the *fitimage* option all formatting options of *PDF_fit_image()* can be specified. The default fit method is *fitmethod=meet*. This means that the image/template will be placed completely inside the cell without distorting its aspect ratio. The cell size will not be changed due to the size of the image/template.

Pages from an imported PDF document. The PDI page must have been opened with *PDF_open_pdi_page()* before calling *PDF_add_table_cell()*. The PDI page handle is supplied in the *pdipage* option. In the *fitpdipage* option all formatting options of *PDF_fit_pdi_page()* can be specified. The default fit method is *fitmethod=meet*. This means that the PDI page will be placed completely inside the cell without distorting its aspect ratio. The cell size will not be changed due to the size of the PDI page.

Path objects. Path objects must have been created with *PDF_add_path_point()* before calling *PDF_add_table_cell()*. The path handle is supplied in the *path* option. In the *fitpath* option all formatting options of *PDF_draw_path()* can be specified. The bounding box of the path will be placed in the table cell. The lower left corner of the inner cell box will be used as reference point for placing the path.

Annotations. Annotations in table cells can be created with the *annotationtype* option of *PDF_add_table_cell()* which corresponds to the *type* parameter of *PDF_create_annotation()* (but this function does not have to be called). In the *fitannotation* option all options of *PDF_create_annotation()* can be specified. The cell box will be used as annotation rectangle.

Form fields. Form fields in table cells can be created with the *fieldname* and *fieldtype* options of *PDF_add_table_cell()* which correspond to the *name* and *type* parameters of *PDF_create_field()* (but this function does not have to be called). In the *fitfield* option all options of *PDF_create_field()* can be specified. The cell box will be used as field rectangle.

Positioning cell contents in the inner cell box. By default, cell contents are positioned with respect to the cell box. The *margin* options of *PDF_add_table_cell()* can be used to specify some distance from the cell borders. The resulting rectangle is called the *inner cell box*. If any margin is defined the cell contents will be placed with respect to the inner cell box (see Figure 8.32). If no margins are defined, the inner cell box is identical to the cell box.

In addition, cell contents may be subject to further options supplied in the content-specific fit options, as described in section Section 8.3.4, »Mixed Table Contents«, page 228.

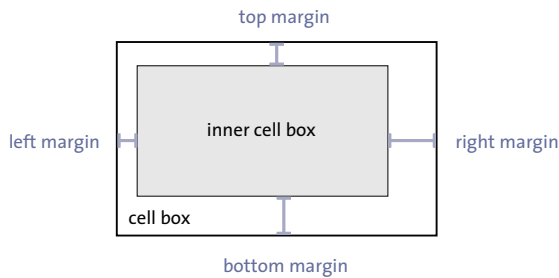


Fig. 8.32
Fitting contents in
the inner cell box

8.3.3 Table and Column Widths

When adding a cell to the table, you define the number of columns and/or rows spanned by the cell with the *colspan* and *rowspan* options. By default, a cell spans one column and one row. The total number of columns and rows in the table is implicitly increased by the respective values when adding a cell. Figure 8.33 shows an example of a table containing three columns and four rows.

row 1	cell spanning three columns		
row 2	cell spanning two columns		cell spanning three rows
row 3	simple cell	simple cell	
row 4	simple cell	simple cell	
	column 1	column 2	column 3

Fig. 8.33
Simple cells and cells spanning
several rows or columns

Furthermore you can explicitly supply the width of the first column spanned by the cell with the *colwidth* option. By supplying each cell with a defined first column width all those width values will implicitly add up to the total table width. Figure 8.34 shows an example.

1 1	colspan=3 colwidth=50		
1 2	colspan=2 colwidth=50		3 2 rowspan=3 colwidth=90
1 3	colspan=1 colwidth=50	2 3	
1 4	colspan=1 colwidth=50	2 4	

50

100

90

total table width of 240

Fig. 8.34
Column widths define
the total table width.

Alternatively, you can specify the column widths as percentages if appropriate. In this case the percentages refer to the width of the table's fitbox. Either none or all column widths must be supplied as percentages.

If some columns are combined to a column scaling group with the `colscalegroup` option of `PDF_add_table_cell()`, their widths will be adjusted to the widest column in the group (see Figure 8.35),

	column scaling group			
	Max. Load	Range	Weight	Speed
Giant Wing	12g	18m	14g	8m/s
Long Distance Glider	5g	30m	11.2g	5m/s
Cone Head Rocket	7g	7m	12.4g	6m/s

Fig. 8.35
The last four cells in the first row are in the same column scaling group. They will have the same widths.

If absolute coordinates are used (as opposed to percentages) and there are cells left without any column width defined, the missing widths are calculated as follows: First, for each cell containing a text line the actual width is calculated based on the column width or the text width (or the text height in case of rotated text). Then, the remaining table width is evenly distributed among the column widths which are still missing.

8.3.4 Mixed Table Contents

In the following sections we will create the sample table containing various kinds of contents as shown in Figure 8.36 step by step.

Cookbook A full code sample can be found in the *Cookbook* topic `tables/mixed_table_contents`.

As a prerequisite we need to load two fonts. We define the dimensions of the table's fit-box in terms of the coordinates of the lower left and upper right corners and specify the widths of the three table columns. Then, we start a new page with A4 size:

```
double llx = 100, lly = 500, urx = 360, ury = 600; // coordinates of the table

int c1 = 50, c2 = 120, c3 = 90; // widths of the three table columns

boldfont = p.load_font("Helvetica-Bold", "unicode", "");
normalfont = p.load_font("Helvetica", "unicode", "");

p.begin_page_ext(0, 0, "width=a4.width height=a4.height");
```

Step 1: Adding the first cell. We start with the first cell of our table. The cell will be placed in the first column of the first row and will span three columns. The first column has a width of 50 points. The text line is centered vertically and horizontally, with a margin of 4 points from all borders. The following code fragment shows how to add the first cell:

```
optlist = "fittextline={font=" + boldfont + " fontsize=12 position=center} " +
          "margin=4 colspan=3 colwidth=" + c1;

tbl = p.add_table_cell(tbl, 1, 1, "Our Paper Plane Models", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());
```


Step 2: Adding one cell spanning two columns. In the next step we add the cell containing the text line *1 Giant Wing*. It will be placed in the first column of the second row and spans two columns. The first column has a width of 50 points. The row height is 14 points. The text line is positioned on the top left, with a margin of 4 points from all borders. We use `fontsize={capheight=6}` to get a unique vertical text alignment as described in »Fine-tuning the vertical alignment of cell contents«, page 224.

Since the *Giant Wing* heading cell doesn't cover a complete row but only two of three columns it cannot be filled with color using on of the row-based shading options. We apply the Matchbox feature instead to fill the rectangle covered by the cell with a gray background color. (The Matchbox feature is discussed in detail in Section 8.4, »Matchboxes«, page 237.) The following code fragment demonstrates how to add the *Giant Wing* heading cell:

```
optlist = "fittextline={position={left top} font=" + boldfont +
          " fontsize={capheight=6}} rowheight=14 colwidth=" + c1 +
          " margin=4 colspan=2 matchbox={fillcolor={gray .92}}";

tbl = p.add_table_cell(tbl, 1, 2, "1 Giant Wing", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());
```

Fig. 8.36 Adding table cells with various contents step by step

Generated table		Generation steps	
Our Paper Plane Models			
1 Giant Wing			
Material	Offset print paper 220g/sqm		
Benefit	It is amazingly robust and can even do aerobatics. But it is best suited to gliding.		

Step 1: Add a cell spanning 3 columns

Step 2: Add a cell spanning 2 columns

Step 3: Add 3 more text line cells

Step 4: Add the Textflow cell

Step 5: Add the image cell with a text line

Step 6: Fitting the table

Step 3: Add three more Textline cells. The following code fragment adds the *Material*, *Benefit* and *Offset print paper...* cells. The *Offset print paper...* cell will start in the second column defining a column width of 120 points. The cell contents is positioned on the top left, with a margin of 4 points from all borders.

```
optlist = "fittextline={position={left top} font=" + normalfont +
          " fontsize={capheight=6}} rowheight=14 colwidth=" + c1 + " margin=4";

tbl = p.add_table_cell(tbl, 1, 3, "Material", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());

tbl = p.add_table_cell(tbl, 1, 4, "Benefit", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());

optlist = "fittextline={position={left top} font=" + normalfont +
          " fontsize={capheight=6}} rowheight=14 colwidth=" + c2 + " margin=4";

tbl = p.add_table_cell(tbl, 2, 3, "Offset print paper 220g/sqm", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());
```

Step 4: Add the Textflow cell. The following code fragment adds the *It is amazingly...* Textflow cell. To add a table cell containing a Textflow we first add the Textflow. We use `fontsize={capheight=6}` which will approximately result in a font size of 8 points and (along with `margin=4`), will sum up to an overall height of 14 points as for the text lines above.

```
tftext = "It is amazingly robust and can even do aerobatics. " +
        "But it is best suited to gliding.";

optlist = "font=" + normalfont + " fontsize={capheight=6} leading=110%";

tf = p.add_textflow(-1, tftext, optlist);
if (tf == -1)
    throw new Exception("Error: " + p.get_errmsg());
```

The retrieved Textflow handle will be used when adding the table cell. The first line of the Textflow should be aligned with the baseline of the *Benefit* text line. At the same time, the *Benefit* text should have the same distance from the top cell border as the *Material* text. Add the Textflow cell using `fittextflow={firstlinedist=capheight}` to avoid any space from the top. Then add a margin of 4 points, the same as for the text lines.

```
optlist = "textflow=" + tf + " fittextflow={firstlinedist=capheight} " +
        "colwidth=" + c2 + " margin=4";

tbl = p.add_table_cell(tbl, 2, 4, "", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());
```

Step 5: Add the image cell with a text line. In the fifth step we add a cell containing an image of the Giant Wing paper plane as well as the *Amazingly robust!* text line. The cell will start in the third column of the second row and spans three rows. The column width is 90 points. The cell margins are set to 4 points. For a first variant we place a TIFF image in the cell:

```
image = p.load_image("auto", "kraxi_logo.tif", "");
if (image == -1)
    throw new Exception("Error: " + p.get_errmsg());

optlist = "fittextline={font=" + boldfont + " fontsize=9} image=" + image +
        " colwidth=" + c3 + " rowspan=3 margin=4";

tbl = p.add_table_cell(tbl, 3, 2, "Amazingly robust!", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());
```

Alternatively, you could import the image as a PDF page. Make sure that the PDI page is closed only after the call to `PDF_fit_table()`.

```
int doc = p.open_pdi("kraxi_logo.pdf", "", 0);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());

page = p.open_pdi_page(doc, pageno, "");
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());
```

```
optlist = "fittextline={font=" + boldfont + " fontsize=9} pdipage=" + page +
         " colwidth=" + c3 + " rowspan=3 margin=4";

tbl = p.add_table_cell(tbl, 3, 2, "Amazingly robust!", optlist);
if (tbl == -1)
    throw new Exception("Error: " + p.get_errmsg());
```

Step 6: Fit the table. In the last step we place the table with *PDF_fit_table()*. Using *header=1* the table header will include the first line. The *fill* option and the suboptions *area=header* and *fillcolor={rgb 0.8 0.8 0.8}* specify the header row(s) to be filled with the supplied color. Using the *stroke* option and the suboptions *line=frame linewidth=0.8* we define a ruling of the table frame with a line width of 0.8. Using *line=other linewidth=0.3* a ruling of all cells is specified with a line width of 0.3.

```
optlist = "header=1 fill={{area=header fillcolor={rgb 0.8 0.8 0.8}}}" +
         "stroke={{line=frame linewidth=0.8} {line=other linewidth=0.3}}";

result = p.fit_table(tbl, llx, lly, urx, ury, optlist);

if (result.equals("_error"))
    throw new Exception("Error: " + p.get_errmsg());

p.end_page_ext("");
```

8.3.5 Table Instances

The rows of the table which are placed in one fitbox comprise a table instance. One or more table instances may be required to represent the full table. Each call to *PDF_fit_table()* will place one table instance in one fitbox. The fitboxes can be placed on the same page, e.g. with a multi-column layout, or on several pages.

The table in Figure 8.37 is spread over three pages. Each table instance is placed in one fitbox on one page. For each call to *PDF_fit_table()* the first row is defined as header and the last row is defined as footer.

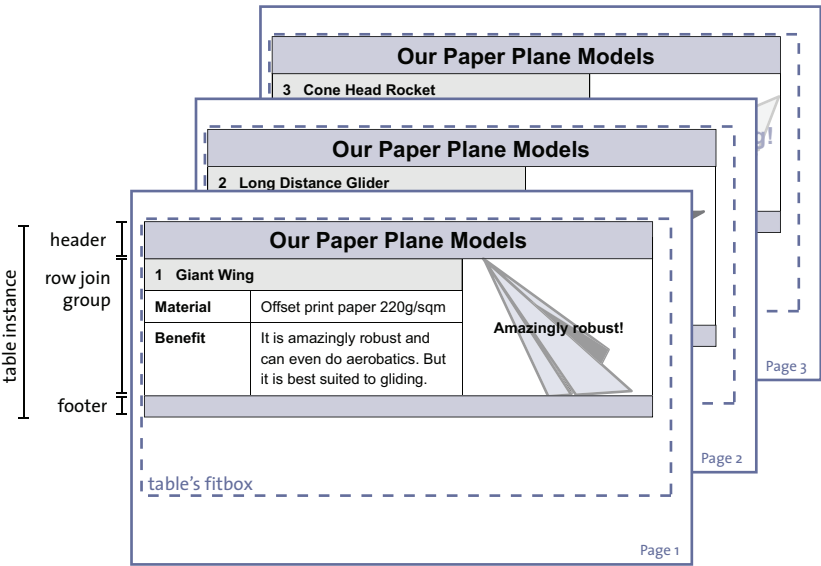


Fig. 8.37
Table broken into several
table instances placed in
one fitbox each.

The following code fragment shows the general loop for fitting table instances until the table has been placed completely. New pages are created as long as more table instances need to be placed.

```
do {
    /* Create a new page */
    p.begin_page_ext(0, 0, "width=a4.width height=a4.height");

    /* Use the first row as header and draw lines for all table cells */
    optlist = "header=1 stroke={{line=other}}";

    /* Place the table instance */
    result = p.fit_table(tbl, llx, lly, urx, ury, optlist);
    if (result.equals("_error"))
        throw new Exception("Error: " + p.get_errmsg());

    p.end_page_ext("");
} while (result.equals("_boxfull"));

/* Check the result; "_stop" means all is ok. */
if (!result.equals("_stop")) {
    if (result.equals("_error"))
        throw new Exception("Error: " + p.get_errmsg());
    else {
        /* Any other return value is a user exit caused by the "return" option;
         * this requires dedicated code to deal with. */
        throw new Exception ("User return found in Textflow");
    }
}
/* This will also delete Textflow handles used in the table */
p.delete_table(tbl, "");
```

Headers and footers. With the *header* and *footer* options of *PDF_fit_table()* you can define the number of initial or final table rows which will be placed at the top or bottom of a table instance. Using the *fill* option with *area=header* or *area=footer*, headers and footers can be individually filled with color. Header rows consist of the first *n* rows of the table definition and footer rows of the last *m* rows.

Headers and footers are specified per table instance in *PDF_fit_table()*. Consequently, they can differ among table instances: while some table instances include headers/footers, others can omit them, e.g. to specify a special row in the last table instance.

Joining rows. In order to ensure that a set of rows will be kept together in the same table instance, they can be assigned to the same row join group using the *rowjoingroup* option. The row join group contains multiple consecutive rows. All rows in the group will be prevented from being separated into multiple table instances.

The rows of a cell spanning these rows don't constitute a join group automatically.

Fitbox too low. If the fitbox is too low to hold the required header and footer rows, and at least one body row or row join group the row heights will be decreased uniformly until the table fits into the fitbox. However, if the required shrinking factor is smaller than the limit set in *vertshrinklimit*, no shrinking will be performed and *PDF_fit_table()* will return the string *_error* instead, or the respective error message. In order to avoid any shrinking use *vertshrinklimit=100%*.

Fitbox too narrow. The coordinates of the table’s fitbox are explicitly supplied in the call to `PDF_fit_table()`. If the actual table width as calculated from the sum of the supplied column widths exceeds the table’s fitbox, all columns will be reduced until the table fits into the fitbox. However, if the required shrinking factor is smaller than the limit set in `horshrinklimit`, no shrinking will be performed and `PDF_fit_table()` will return the string `_error` instead, or the respective error message. In order to avoid any shrinking use `horshrinklimit=100%`.

Splitting a cell. If the last rows spanned by a cell doesn’t fit in the fitbox the cell will be split. In case of an image, PDI page or text line cell, the cell contents will be repeated in the next table instance. In case of a Textflow cell, the cell contents will continue in the remaining rows of the cell.

Figure 8.38 shows how the Textflow cell will be split while the Textflow continues in the next row. In Figure 8.39, an image cell is shown which will be repeated in the first row of the next table instance.

table instance 1	1 Giant Wing		Our paper planes are the ideal way of passing the time. We offer revolutionary
	Material	Offset print paper 220g/sqm	
table instance 2	Benefit	It is amazingly robust and can even do aerobatics. But it is best suited to gliding.	new developments of the traditional common paper planes.

Fig. 8.38
Splitting a cell

Splitting a row. If the last body row doesn’t completely fit into the table’s fitbox, it will usually not be split. This behaviour is controlled by the `minrowheight` option of `PDF_fit_table()` with a default value of 100%. With this default setting the row will not be split but will completely be placed in the next table instance.

You can decrease the `minrowheight` value to split the last body row with the given percentage of contents in the first instance, and place the remaining parts of the row in the next instance.

Figure 8.39 illustrates how the Textflow *It’s amazingly robust...* is split and the Textflow is continued in the first body row of the next table instance. The image cell spanning several rows will be split and the image will be repeated. The *Benefit* text line will be repeated as well.

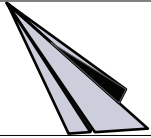

table instance 1	1 Giant Wing		
	Material	Offset print paper 220g/sqm	
	Benefit	It is amazingly robust and can even do aerobatics. But	
table instance 2	Benefit	it is best suited to gliding.	

Fig. 8.39
Splitting a row

8.3.6 Table Formatting Algorithm

This section details the steps performed by the table formatter when placing a table. The description below applies to horizontal text. However, if you swap the terms »row height« and »column width« it also applies to vertical or rotated text.

In the first call to `PDF_fit_table()` the options `colwidth`, `rowheight`, `fittextline`, and `fittextflow` are examined for all cells, and the width and height of the full table is calculated based on column widths, row heights and text contents.

Calculate the height and width of table cells with Textlines. The table formatter initially determines the size of all those table cells with Textlines which span table columns or rows without `colwidth` or `rowheight`. In order to achieve this it calculates the width of the Textline and therefore the table cell according to the `fittextline` option. It assumes twice the text size as height of the table cell (more precisely: twice the `boxheight`, which has the default value `{capheight none}` unless specified otherwise). For vertical text the width of the widest character will be used as cell width. For text orientated to west or east twice the text height will be used as cell width.

The resulting width and height of the table cell is then distributed evenly among all those columns and rows spanned by the cell for which `colwidth` or `rowheight` hasn't been specified.

Calculate a tentative table size. In the next step the formatter calculates a tentative table width and height as the sum of all column widths and row heights, respectively. Column widths and row heights specified as percentages are converted to absolute values based on the width and height of the first fitbox. If there are still columns or rows without `colwidth` or `rowheight` the remaining space is evenly distributed until the tentative table size equals the first fitbox.

The `rowheightdefault` option can be used to completely fill the height of the fitbox (keywords `auto` and `distribute`) or save space (keyword `minimum`). Explicitly specifying the height of a row with the `rowheight` option always overrides the `rowheightdefault` setting.

Enlarge cells which are too small. Now the formatter determines all inner cell boxes (see Figure 8.32). If the combined margins are larger than the cell's width or height, the cell box is suitably enlarged by evenly enlarging all columns and rows which belong to the cell.

Fit Textlines horizontally. The formatter attempts to increase the width of all cells with Textlines so that the Textline fits into the cell without reducing the font size. If this is not possible, the Textline is automatically placed with `fitmethod=auto`. This guarantees that the Textline will not extend beyond the inner cell box. You can prevent the cell width from being increased by setting `fitmethod=auto` in the `fittextline` option.

You can use the `colscalegroup` option to make sure that all columns which belong to the same column scaling group will be scaled to equal widths, i.e. their widths will be unified and adjusted to the widest column in the group (see Figure 8.35).

Avoid forced hyphenation. If the calculated table width is smaller than the fitbox the formatter tries to increase the width of a Textflow cell so that the text fits without forced hyphenation. This can be avoided with the option `checkwordsplitting=false`. The

widths of such cells will be increased until the table width equals the width of the fitbox.

You can query the difference between table width and fitbox width with the *horbox-gap* key of *PDF_info_table()*.

Fit text vertically. The formatter attempts to increase the height of all Textline and Textflow cells so that the Textline or Textflow fits into the inner cell box without reducing the font size. However, the cell height will not be increased if for a Textline or Textflow the suboption *fitmethod=auto* is set, or a Textflow is continued in another cell with the *continuetextflow* option.

This process of increasing the cell height applies only to cells containing a Textline or Textflow, but not for other types of cell contents, i.e. images, PDI pages, path objects, annotations, and fields.

You can use the *rowscalegroup* option to make sure that all rows which belong to the same row scaling group will be scaled to equal heights.

Continue the table in the next fitbox. If the table's resulting total height is larger than the fitbox (i.e. not all table cells fit into the fitbox), the formatter stops placing rows in the fitbox before it encounters the first row which doesn't fit into the fitbox.

If a cell spans multiple lines and not all of those lines fit into the fitbox, this cell will be split. If the cell contains an image, PDI page, path object, annotation, form field, or Textline, the cell contents will be repeated in the next fitbox unless *repeatcontent=false* has been specified. Textflows, however, will be continued in the subsequent rows spanned by the cell (see Figure 8.38).

You can use the *rowjoininggroup* option to make sure that all rows belonging to a row joining group will always appear together in a fitbox. All rows which belong to the header or footer plus one body line automatically form a row joining group. The formatter may therefore stop placing table rows before it encounters the first line which doesn't fit into the fitbox (see Figure 8.37).

You can use the *return* option to make sure that now more rows will be placed in the table instance after placing the affected line.

Split a row. A row may be split if it is very high or if there is only a single body line. If the last body line doesn't fully fit into the table's fitbox, it will completely be moved to the next fitbox. This behavior is controlled by the *minrowheight* option of *PDF_fit_table()*, which has a default value of 100%. If you reduce the *minrowheight* value the specified percentage of the content of the last body line will be placed in the current fitbox and the rest of the line in the next fitbox (see Figure 8.39).

You can check whether a row has been split with the *rowsplit* key of *PDF_info_table()*.

Adjust the calculated table width. The calculated table width may be larger than the fitbox width after one of the determination steps, e.g. after fitting a Textline horizontally. In this case all column widths will be evenly reduced until the table width equals the width of the fitbox. This shrinking process is limited by the *horshrinklimit* option.

You can query the horizontal shrinking factor with the *horshrinking* key of *PDF_info_table()*.

If the *horshrinklimit* threshold is exceeded the following error message appears:

Calculated table width \$1 is too large (> \$2, shrinking \$3)"

Here \$1 designates the calculated table width, \$2 the maximum possible width and \$3 the *horshrinklimit* value.

Adjust the table size to a small fitbox. If the table width which has been calculated for the previous fitbox is too large for the current fitbox, the formatter evenly reduces all columns until the table width equals the width of the current fitbox. The cell contents will not be adjusted, however. In order to calculate the table width anew, call *PDF_fit_table()* with *rewind=1*.

8.4 Matchboxes

Matchboxes provide access to coordinates calculated by PDFlib as a result of placing some content on the page. Matchboxes are not defined with a dedicated function, but with the *matchbox* option in the function call which places the actual element, for example *PDF_fit_textline()* and *PDF_fit_image()*. Matchboxes can be used for various purposes:

- ▶ Matchboxes can be decorated, e.g. filled with color or surrounded by a frame.
- ▶ Matchboxes can be used to automatically create one or more annotations with *PDF_create_annotation()*.
- ▶ Matchboxes define the height of a text line which will be fit into a box with *PDF_fit_textline()* or the height of a text fragment in a Textflow which will be decorated (*boxheight* option).
- ▶ Matchboxes define the clipping for an image.
- ▶ The coordinates of the matchbox and other properties can be queried with *PDF_info_matchbox()* to perform some other task, e.g. insert an image.

For each element PDFlib will calculate the matchbox as a rectangle corresponding to the bounding box which describes the position of the element on the page (as specified by all relevant options). For Textflows and table cells a matchbox may consist of multiple rectangles because of line or row breaking.

The rectangle(s) of a matchbox will be drawn before drawing the element to be placed. As a result, the element may obscure the effect of the matchbox border or filling, but not vice versa. In particular, those parts of the matchbox which overlap the area covered by an image are hidden by the image. If the image is placed with *fitmethod=slice* or *fitmethod=clip* the matchbox borders outside the image fitbox will be clipped as well. To avoid this effect the matchbox rectangle can be drawn using the basic drawing functions, e.g. *PDF_rect()*, after the *PDF_fit_image()* call. The coordinates of the matchbox rectangle can be retrieved using *PDF_info_matchbox()* as far as the matchbox has been provided with a name in the *PDF_fit_image()* call.

In the following sections some examples for using matchboxes are shown. For details about the functions which support the *matchbox* option list, see the *PDFlib API Reference*.

8.4.1 Decorating a Textline

Let's start with a discussion of matchboxes in text lines. In *PDF_fit_textline()* the matchbox is the textbox of the supplied text. The width of the textbox is the text width, and the height is the capheight of the given font size, by default. To illustrate the matchbox size the following code fragment will fill the matchbox with blue background color (see Figure 8.40a).

```
String optlist =  
    "font=" + normalfont + " fontsize=8 position={left top} " +  
    "matchbox={fillcolor={rgb 0.8 0.8 0.87} boxheight={capheight none}}";  
  
p.fit_textline("Giant Wing Paper Plane", 2, 20, optlist);
```

You can omit the *boxheight* option since *boxheight={capheight none}* is the default setting. It will look better if we increase the box height so that it also covers the descenders using the *boxheight* option (see Figure 8.40b).

To increase the box height to match the font size we can use `boxheight={fontsize descender}` (see Figure 8.40c).

In the next step we extend the matchbox by some offsets to the left, right and bottom to make the distance between text and box margins the same. In addition, we draw a rectangle around the matchbox by specifying the border width (see Figure 8.40d).

Cookbook A full code sample can be found in the Cookbook topic `text_output/text_on_color`.

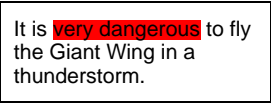
Fig. 8.40 Decorating a text line using a matchbox with various suboptions

Generated output	Suboptions of the matchbox option of PDF_fit_textline()
a) Giant Wing Paper Plane	boxheight={capheight none}
b) Giant Wing Paper Plane	boxheight={ascender descender}
c) Giant Wing Paper Plane	boxheight={fontsize descender}
d) Giant Wing Paper Plane	boxheight={fontsize descender} borderwidth=0.3 offsetleft=-2 offsetright=2 offsetbottom=-2

8.4.2 Using Matchboxes in a Textflow

Decorating parts of a Textflow. In this section we will decorate some text within a Textflow: The words *very dangerous* will be emphasized similar to a marker pen. To accomplish this the words are enclosed in the `matchbox` and `matchbox=end` inline options (see Figure 8.41).

Fig. 8.41 Textflow with matchbox inline option

Generated output	Text and inline options for PDF_create_textflow()
	It is <matchbox={fillcolor={rgb 1 0 0}} boxheight={ascender descender}>very dangerous <matchbox=end> to fly the Giant Wing in a thunderstorm.

Adding a Web link to the Textflow matchbox. Now we will add a Web link to parts of a Textflow. In the first step we create the Textflow with a matchbox called *kraxi* indicating the text part to be linked. Second, we will create the action for opening a URL. Third, we create an annotation of type *Link* with an invisible frame. In its option list we reference the *kraxi* matchbox to be used as the link's rectangle (the rectangle coordinates in `PDF_create_textflow()` will be ignored).

Cookbook A full code sample can be found in the Cookbook topic `text_output/weblink_in_text`.

```
/* create and fit Textflow with matchbox "kraxi" */
String tftext =
    "For more information about the Giant Wing Paper Plane see the Web site of " +
    "<underline=true matchbox={name=kraxi boxheight={fontsize descender}}>" +
    "Kraxi Systems, Inc.<matchbox=end underline=false>";

String optlist = "font=" + normalfont + " fontsize=8 leading=110%";
tflow = p.create_textflow(tftext, optlist);
if (tflow == -1)
    throw new Exception("Error: " + p.get_errmsg());
```

```

result = p.fit_textflow(tflow, 0, 0, 50, 70, "fitmethod=auto");
if (!result.equals("_stop"))
    { /* ... */ }

/* create URI action */
optlist = "url={http://www.kraxi.com}";
act = p.create_action("URI", optlist);

/* create Link annotation on matchbox "kraxi" */
optlist = "action={activate " + act + "} linewidth=0 usematchbox={kraxi}";
p.create_annotation(0, 0, 0, 0, "Link", optlist);

```

Even if the text *Kraxi Systems, Inc.* spans several lines the appropriate number of link annotations will be created automatically with a single call to *PDF_create_annotation()*. The result is shown in Figure 8.42.

For information about
Giant Wing Paper
Planes see the Web
site of Kraxi Systems,
Inc.



<http://www.kraxi.com>

*Fig. 8.42
Add Weblinks to parts of a Textflow*

8.4.3 Matchboxes and Images

Adding a Web link to an image. To add a Web link to the area covered by an image the image matchbox can be used. The code is similar to »Adding a Web link to the Textflow matchbox«, page 238, above. However, instead of placing the Textflow, fit the image using the following option list:

```

String optlist = "boxsize={130 130} fitmethod=meet matchbox={name=kraxi}";
p.fit_image(image, 10, 10, optlist);

```


Cookbook A full code sample can be found in the *Cookbook topic* [interactive/link_annotations](#).

Drawing a frame around an image. In this example we want to use the image matchbox to draw a frame around the image. We completely fit the image into the supplied box while maintaining its proportions using *fitmethod=meet*. We use the *matchbox* option with the *borderwidth* suboption to draw a thick border around the image. The *strokecolor* suboption determines the border color, and the *linecap* and *linejoin* suboptions are used to round the corners.

Cookbook A full code sample can be found in the *Cookbook topic* [images/frame_around_image](#).

The matchbox is always drawn before the image which means it would be partially hidden by the image. To avoid this we use the *offset* suboptions with 50 percent of the border width to enlarge the frame beyond the area covered by the image. Alternatively, we could increase the border width accordingly. Figure 8.43 shows the option list used with *PDF_fit_image()* to draw the frame.

Fig. 8.43 Using the image matchbox to draw a frame around the image

Generated output	Option list for PDF_fit_image()
	<pre>boxsize={60 60} position={center} fitmethod=meet matchbox={name=kraxi borderwidth=4 offsetleft=-2 offsetright=2 offsetbottom=-2 offsettop=2 linecap=round linejoin=round strokecolor {rgb 0.0 0.3 0.3}}</pre>

Align text at an image. The following code fragment shows how to align vertical text at the right margin of an image. The image is proportionally fit into the supplied box with a fit method of *meet*. The actual coordinates of the fitbox are retrieved with *PDF_info_matchbox()* and a vertical text line is placed relative to the lower right (*x2*, *y2*) corner of the fitbox. The border of the matchbox is stroked (see Figure 8.44).

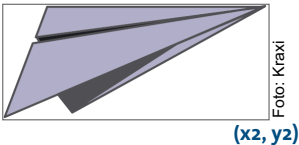
Cookbook A full code sample can be found in the Cookbook topic `images/align_text_at_image`.

```
/* use this option list to load and fit the image */
String  optlist = "boxsize={300 200} position={center} fitmethod=meet " +
    "matchbox={name=giantwing borderwidth=3 strokecolor={rgb 0.85 0.83 0.85}}";

/* load and fit the image */
...

/* retrieve the coordinates of the lower right (second) matchbox corner */
if ((int) p.info_matchbox("giantwing", 1, "exists") == 1)
{
    x1 = p.info_matchbox("giantwing", 1, "x2");
    y1 = p.info_matchbox("giantwing", 1, "y2");
}
/* start the text line at that corner with a small distance of 2 */
p.fit_textline("Foto: Kraxi", x2+2, y2+2, "font=" + font + " fontsize=8 orientate=west");
```

Fig. 8.44 Use the coordinates of the image matchbox to fit a text line

Generated output	Generation steps
	<p>Step 1: Fit image with matchbox</p> <p>Step 2: Retrieve matchbox info for coordinates (<i>x2</i>, <i>y2</i>)</p> <p>Step 3: Fit text line starting at retrieved coordinates (<i>x2</i>, <i>y2</i>) with option <i>orientate=west</i></p>

9 The pCOS Interface

The pCOS (*PDFlib Comprehensive Object Syntax*) interface provides a simple and elegant facility for retrieving arbitrary information from all sections of a PDF document which do not describe page contents, such as page dimensions, metadata, interactive elements, etc. Examples for using the pCOS interface and a description of the pCOS path syntax are contained in the pCOS Path Reference which is available as a separate document. Additional examples can be found in the pCOS Cookbook at www.pdflib.com/pcos-cookbook/





10 PDF Versions and Standards

10.1 Acrobat and PDF Versions

At the user’s option PDFlib generates output according to the following PDF versions:

- ▶ PDF 1.3 (Acrobat 4)
- ▶ PDF 1.4 (Acrobat 5)
- ▶ PDF 1.5 (Acrobat 6)
- ▶ PDF 1.6 (Acrobat 7)
- ▶ PDF 1.7 (Acrobat 8), technically identical to ISO 32000-1
- ▶ PDF 1.7 Adobe extension level 3 (Acrobat 9)
- ▶ PDF 1.7 Adobe extension level 8 (Acrobat X)

The PDF output version can be controlled with the *compatibility* option in *PDF_begin_document()*. In each PDF compatibility mode the PDFlib features for higher levels are not available (see Table 10.1). Trying to use such features will result in an exception.

PDF version of documents imported with PDI. In all compatibility modes only PDF documents with a lower or the same PDF version can be imported with PDI. If you must import a PDF with a newer PDF version you must set the *compatibility* option accordingly (see Section 7.2.3, »Acceptable PDF Documents«, page 184). As an exception to the cannot-import-higher-PDF-version rule, documents according to PDF 1.7 extension level 3 (Acrobat 9) and PDF 1.7 extension level 8 (Acrobat X) can also be imported into PDF 1.7 documents.

Changing the PDF version of a document. If you must create output according to a particular PDF version, but need to import PDFs which use a higher PDF version you must convert the documents to the desired lower PDF version before you can import them with PDI. You can use the menu item *Advanced, PDF Optimizer, Make compatible with* in Acrobat 7/8/9 Professional or *File, Save As..., Optimized PDF...* in Acrobat X to change the PDF version as follows:

- ▶ Acrobat 7: PDF 1.3 - PDF 1.6
- ▶ Acrobat 8: PDF 1.3 - PDF 1.7
- ▶ Acrobat 9: PDF 1.3 - PDF 1.7 extension level 3
- ▶ Acrobat X: PDF 1.3 - PDF 1.7 extension level 8

Table 10.1 PDFlib features which require a specific PDF compatibility mode

Feature	PDFlib API functions and options
Features which require PDF 1.7 extension level 8 (Acrobat X)	
direct use of layers for PDF/X-4:2010 (without layer variants)	<code>PDF_set_layer_dependency()</code> : option <code>createorderlist</code>
Features which require PDF 1.7 extension level 3 (Acrobat 9)	
Geospatial PDF	<code>PDF_begin_document()</code> : option <code>viewports</code> <code>PDF_load_image()</code> : option <code>georeference</code>
PDF portfolios with folders	<code>PDF_add_portfolio_folder()</code>

Table 10.1 PDFlib features which require a specific PDF compatibility mode

Feature	PDFlib API functions and options
AES encryption with 256-bit keys	<code>PDF_begin_document()</code> : AES encryption with 256-bit will automatically be used with <code>compatibility=1.7ext3</code> when the <code>masterpassword</code> , <code>userpassword</code> , <code>attachmentpassword</code> , or <code>permissions</code> option is supplied
Layer variants	<code>PDF_set_layer_dependency()</code> : <code>dependency</code> type Variant (This feature does not require PDF 1.7 ext 3, but works only in Acrobat 9)
Referenced PDF	<code>reference</code> option in <code>PDF_open_pdi_page()</code> and <code>PDF_begin_template_ext()</code> (This feature does not require PDF 1.7 ext 3, but works only in Acrobat 9)
embed 3D models in PRC format	<code>PDF_load_3ddata()</code> : <code>option</code> type=PRC
barcode fields	<code>PDF_create_field()</code> and <code>PDF_create_fieldgroup()</code> : <code>option</code> barcode
Features which require PDF 1.7 (Acrobat 8)	
PDF portfolios	<code>PDF_begin_document()</code> : <code>option</code> portfolio <code>PDF_add_portfolio_file()</code>
Unicode file names for attachments	<code>PDF_begin/end_document()</code> : <code>option</code> attachments, <code>suboption</code> filename
Features which require PDF 1.6 (Acrobat 7)	
user units	<code>PDF_begin/end_document()</code> : <code>option</code> userunit
print scaling	<code>PDF_begin/end_document()</code> : <code>suboption</code> printscaling for viewer preferences <code>option</code>
document open mode	<code>PDF_begin/end_document()</code> : <code>option</code> openmode=attachments
AES encryption with 128-bit keys	<code>PDF_begin_document()</code> : AES encryption will automatically be used with <code>compatibility=1.6</code> or <code>1.7</code> when the <code>masterpassword</code> , <code>userpassword</code> , <code>attachmentpassword</code> , or <code>permissions</code> option is supplied
encrypt file attachments only	<code>PDF_begin/end_document()</code> : <code>option</code> attachmentpassword
attachment description	<code>PDF_begin/end_document()</code> : <code>suboption</code> description for <code>option</code> attachments
embed 3D models in U3D format	<code>PDF_load_3ddata()</code> , <code>PDF_create_3dview()</code> ; <code>PDF_create_annotation()</code> : <code>type=3D</code>
Features which require PDF 1.5 (Acrobat 6)	
various field options	<code>PDF_create_field()</code> and <code>PDF_create_fieldgroup()</code>
page layout	<code>PDF_begin/end_document()</code> : <code>option</code> pagelayout=twopageleft/right
various annotation options	<code>PDF_create_annotation()</code>
extended permission settings	<code>permissions=plainmetadata</code> in <code>PDF_begin_document()</code> , see Table 3.3
various CMaps for CJK fonts	<code>PDF_load_font()</code> , see Table 4.3
Tagged PDF	various options for <code>PDF_begin_item()</code> ; <code>PDF_begin/end_page_ext()</code> : <code>option</code> taborder
Layers	<code>PDF_define_layer()</code> , <code>PDF_begin_layer()</code> , <code>PDF_end_layer()</code> , <code>PDF_layer_dependency()</code>
JPEG2000 images	<code>imagetype=jpeg2000</code> in <code>PDF_load_image()</code>
compressed object streams	compressed object streams will automatically be generated with <code>compatibility=1.5</code> or above unless <code>objectstreams=none</code> has been set in <code>PDF_begin_document()</code>

Table 10.1 PDFlib features which require a specific PDF compatibility mode

Feature	PDFlib API functions and options
Features which require PDF 1.4 (Acrobat 5)	
smooth shadings (color blends)	<code>PDF_shading_pattern()</code> , <code>PDF_shfill()</code> , <code>PDF_shading()</code>
soft masks	<code>PDF_load_image()</code> with the <code>masked</code> option referring to an image with more than 1 bit pixel depth
JBIG2 images	<code>imagetype=jbig2</code> in <code>PDF_load_image()</code>
128-bit encryption	<code>PDF_begin_document()</code> with the <code>userpassword</code> , <code>masterpassword</code> , <code>permissions</code> options
extended permission settings	<code>PDF_begin_document()</code> with <code>permissions</code> option, see Table 3.3
various CMaps for CJK fonts	<code>PDF_load_font()</code> , see Table 4.3
transparency and other graphics state options	<code>PDF_create_gstate()</code> with options <code>alphaissshape</code> , <code>blendmode</code> , <code>opacityfill</code> , <code>opacitystroke</code> , <code>textknockout</code>
various options for actions	<code>PDF_create_action()</code>
various options for annotations	<code>PDF_create_annotation()</code>
various field options	<code>PDF_create_field()</code> and <code>PDF_create_fieldgroup()</code>
Tagged PDF	<code>tagged</code> option in <code>PDF_begin_document()</code>
Referenced PDF	<code>reference</code> option in <code>PDF_open_pdi_page()</code> and <code>PDF_begin_template_ext()</code> (however, note that this feature requires Acrobat 9 for proper display/printing)

10.2 ISO 32 000

PDF 1.7 has been standardized as ISO 32000-1. The technical contents of this international standard are identical to Adobe's PDF 1.7, the file format of Acrobat 8. PDFlib adheres to Adobe's PDF Reference and therefore ISO 32000-1. To the best of our knowledge the PDF output created with PDFlib fully conforms to ISO 32000-1. However, currently no validation software is available for checking ISO 32000-1 conformance.

At the time of writing the next version of the ISO standard is being prepared under the title ISO 32000-2. This standard will incorporate Acrobat 9 features which are currently supported in PDFlib with the *compatibility=pdf1.7ext3* setting, e.g. georeferenced PDF, hierarchical Portfolios, and AES-256 encryption (see Table 10.1 for a detailed list).

10.3 PDF/X for Print Production

10.3.1 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. PDFlib can generate output and process input conforming to the PDF/X flavors described below.

PDF/X-1a:2001 as defined in ISO 15930-1. This standard for »blind exchange« (exchange of print data without the requirement for any prior technical discussions) is based on PDF 1.3 and supports CMYK and spot color data. RGB and device-independent colors (ICC-based, Lab) are explicitly prohibited. PDF/X-1a:2001 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.

Note PANTONE® colors are not supported in PDF/X-1a mode.

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.

PDF/X-4 as defined in ISO 15930-7. This standard can be regarded as the successor of PDF/X-1a and PDF/X-3. It is based on PDF 1.6 and consists of the following flavors:

- ▶ In PDF/X-4 transparency and layers are allowed (with some restrictions), but some other PDF 1.6 features are still prohibited.
- ▶ PDF/X-4p allows output intent ICC profiles to be kept external from the PDF document to save space.

PDFlib implements the 15930-7:2010 version of the PDF/X-4 standard. The 2010 version introduces some changes regarding treatment of layers. The 2010 functionality can be addressed with the *createorderlist* option of *PDF_set_layer_dependency()*. This option should not be used for PDF/X-4:2008.

PDF/X-5 as defined in ISO 15930-8. This standard is targeted at »partial exchange« which requires prior discussion between supplier and receiver of a file. It can be regarded as an extension of PDF/X-4 and PDF/X-4p (i.e. based on PDF 1.6), and consists of the following flavors:

- ▶ PDF/X-5g allows graphical content external from the PDF document; this requires some communication between sender and receiver of the documents.
- ▶ PDF/X-5pg allows external graphical content and external output intent ICC profiles;

- ▶ PDF/X-5n supports external output intent ICC profiles for n-colorant print characterizations. This flavor is not supported in PDFlib.

If none of the specific features of PDF/X-5 is required the document should instead be prepared according to PDF/X-4 or PDF/X-4p since this is the more general standard. ISO 15930-8:2008 contains several errors related to XMP identification entries for externally referenced graphics. The revised version ISO 15930-8:2010 of the standard replaces the 2008 version. PDFlib implements PDF/X-5:2010 including corrigendum 1 which has been published in 2011.

Note PDF/X-5g validation fails in Acrobat 9 Preflight if external pages are referenced. This problem has been fixed in Acrobat X.

10.3.2 Generating PDF/X-conforming Output

Cookbook Code samples for generating PDF/X can be found in the and pdfx category of the PDFlib Cookbook.

- 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 options as detailed in Table 10.2.
 - ▶ The PDFlib client must refrain from using certain function calls and options as detailed in Table 10.3.
 - ▶ Additional rules apply when importing pages from existing PDF/X-conforming documents (see Section 10.3.4, »Importing PDF/X Documents with PDI«, page 252).

Required operations. Table 10.2 lists all operations required to generate PDF/X-conforming 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 10.2 Operations which must be applied for PDF/X compatibility

item	PDFlib function and option requirements for PDF/X compatibility
conformance level	The pdfx option in PDF_begin_document() must be set to the desired PDF/X conformance level.
output condition (output intent)	PDF_load_iccprofile() with usage=outputintent or PDF_process_pdi() with action=copy-outputintent (but not both methods) must be called immediately after PDF_begin_document(). If HKS or Pantone spot colors, ICC-based colors, or Lab colors are used, an output device ICC profile must be embedded; using a standard output condition is not allowed in this case. PDF/X-1a: the output device must be a monochrome or CMYK device; PDF/X-3/4/5: the output device must be a monochrome, RGB, or CMYK device.
font embedding	Set the embedding option of PDF_load_font() (and other functions which accept this option) to true to enable font embedding. Note that embedding is also required for the PDF core fonts.
page boxes	The page boxes, which are settable via the cropbox, bleedbox, trimbox, and artbox options, must satisfy all of the following requirements: <ul style="list-style-type: none"> ▶ The TrimBox or ArtBox must be set, but not both of these box entries. If both TrimBox and ArtBox are missing PDFlib will take the CropBox (if present) as the TrimBox, and the MediaBox if the CropBox is also missing. ▶ The BleedBox, if present, must fully contain the ArtBox and TrimBox. ▶ The CropBox, if present, must fully contain the ArtBox and TrimBox.

Table 10.2 Operations which must be applied for PDF/X compatibility

item	PDFlib function and option requirements for PDF/X compatibility
grayscale color	PDF/X-3/4/5: Grayscale images and PDF_setcolor() with a gray color space can only be used if the output condition is a grayscale or CMYK device, or if the defaultgray option in PDF_begin_page_ext() has been set.
RGB color	PDF/X-3/4/5: RGB images and PDF_setcolor() with an RGB color space can only be used if the output condition is an RGB device, or the defaultrgb option in PDF_begin_page_ext() has been set.
CMYK color	PDF/X-3/4/5: CMYK images and PDF_setcolor() with a CMYK color space can only be used if the output condition is a CMYK device, or the defaultcmky option in PDF_begin_page_ext() has been set.
document info keys	The Creator and Title info keys must be set to a non-empty value with PDF_set_info() or (in PDF/X-4 and PDF/X-5) with the xmp:CreatorTool and dc:title XMP properties in the metadata option of PDF_begin/end_document()

Prohibited operations. Table 10.3 lists all operations which are prohibited when generating PDF/X-conforming 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. Similarly, if an imported PDF page doesn't match the current PDF/X conformance level, the corresponding PDI call will fail.

Table 10.3 Operations which must be avoided or are restricted to achieve PDF/X compatibility

item	Prohibited or restricted PDFlib functions and options for PDF/X compatibility
grayscale color	PDF/X-1a: the defaultgray option in PDF_begin_page_ext() must be avoided.
RGB color	PDF/X-1a: RGB images and the defaultrgb option in PDF_begin_page_ext() must be avoided.
CMYK color	PDF/X-1a: the defaultcmky option in PDF_begin_page_ext() must be avoided.
ICC-based color	PDF/X-1a: the iccbasedgray/rgb/cmyk color space in PDF_setcolor() and the setcolor:icc-profilegray/rgb/cmyk parameters must be avoided.
Lab color	PDF/X-1a: the Lab color space in PDF_setcolor() must be avoided.
annotations and form fields	Annotations inside the BleedBox (or TrimBox/ArtBox if no BleedBox is present) must be avoided: PDF_create_annotation(), PDF_create_field().
file attachments	PDF/X-1a/3: PDF_begin/end_document(): option attachments must be avoided; PDF_create_annotation() with type=FileAttachment must be avoided
actions and JavaScript	All actions including JavaScript must be avoided: PDF_create_action()
images	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. PDF/X-1 and PDF/X-3: JBIG2 images must be avoided. The OPI-1.3 and OPI-2.0 options in PDF_load_image() must be avoided.
transparent images and graphics	PDF/X-1 and PDF/X-3: Soft masks for images must be avoided: the masked option for PDF_load_image() must be avoided unless the mask refers to a 1-bit image. Images with implicit transparency (alpha channel) are not allowed; they must be loaded with the ignoremask option of PDF_load_image(). The opacityfill and opacitystroke options for PDF_create_gstate() must be avoided unless they have a value of 1; if blendmode is used it must be Normal. Transparent images and graphics are allowed in PDF/X-4 and PDF/X-5.

Table 10.3 Operations which must be avoided or are restricted to achieve PDF/X compatibility

item	Prohibited or restricted PDFlib functions and options for PDF/X compatibility
transparency groups	<p>The transparencygroup option of PDF_begin/end_page_ext(), PDF_begin_template_ext(), and PDF_open_pdi_page() is not allowed in PDF/X-1 and PDF/X-3, but only in PDF/X-4 and PDF/X-5. If transparencygroup is used, the value of the colorspace suboption is subject to the following requirements:</p> <ul style="list-style-type: none">▶ DeviceGray: the PDF/X output condition must be a grayscale or CMYK device. For the generated page (but not for templates and imported pages) the defaultgray option in PDF_begin_page_ext() can be set as an alternative.▶ DeviceRGB: the PDF/X output condition must be an RGB device. For the generated page (but not for templates and imported pages) the defaultrgb option in PDF_begin_page_ext() can be set as an alternative.▶ DeviceCMYK: the PDF/X output condition must be a CMYK device. For the generated page (but not for templates and imported pages) the defaultcmyk option in PDF_begin_page_ext() can be set as an alternative.
viewer preferences / view and print areas	<p>When the viewarea, viewclip, printarea, and printclip suboptions for the viewer-preferences option in PDF_begin/end_document() are used values other than media or bleed are not allowed.</p>
document info keys	<p>Values other than True or False for the Trapped info key or the corresponding XMP property pdf:Trapped PDF_set_info() must be avoided.</p>
security	<p>The userpassword, masterpassword, and permissions options in PDF_begin_document() must be avoided.</p>
PDF version / compatibility	<p>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.</p> <p>PDF/X-1a:2003 and PDF/X-3:2003 are based on PDF 1.4. Operations that require PDF 1.5 or above must be avoided.</p> <p>PDF/X-4 and PDF/X-5 are based on PDF 1.6. Operations that require PDF 1.7 or above must be avoided.</p>
PDF import (PDI)	<p>Imported documents must conform to a compatible PDF/X level according to Table 10.5, and must have been prepared according to the same output intent.</p>
external graphical content (references)	<p>PDF/X-1/3/4: The reference option in PDF_begin_template_ext() and PDF_open_pdi_page() must be avoided.</p> <p>PDF/X-5g and PDF/X-5pg: the target provided in the reference option in PDF_begin_template_ext() and PDF_open_pdi_page() must conform to one of the following standards: PDF/X-1a:2003, PDF/X-3:2003, PDF/X-4, PDF/X-4p, PDF/X-5g, or PDF/X-5pg, and must have been prepared for the same output intent. Since certain XMP metadata entries are required in the target, not all PDF/X documents are acceptable as target. PDF/X documents generated with PDFlib 8 can be used as target.</p> <p>See Section 3.2.5, »Referenced Pages from an external PDF Document«, page 70, for more details on the reference option and the required Acrobat configuration.</p>
layers	<p>PDF/X-1 and PDF/X-3: layers require PDF 1.5 and can therefore not be used.</p> <p>PDF/X-4 and PDF/X-5: layers can be used but certain rules must be obeyed:</p> <ul style="list-style-type: none">▶ Some options of PDF_define_layer() and PDF_set_layer_dependency() must be avoided.▶ PDF/X-4:2010: the createorderlist option of PDF_set_layer_dependency() is allowed. It is required to display the list of layers in Acrobat X.
file size	<p>PDF/X-4 and PDF/X-5: The file size of the generated PDF document must not exceed 2 GB, and the number of PDF objects must be smaller than 8.388.607. See Section 3.1.5, »Large PDF Documents«, page 62, for more details about these limits.</p>

10.3.3 Output Intent and Standard Output Conditions

The output intent (also called output condition) defines the intended target device, which is mainly useful for reliable proofing. The output intent can be specified as a name (called standard output intent) or with an ICC color profile. The details vary among the PDF/X flavors:

- ▶ PDF/X-1a/3/4 and PDF/X-5g: by embedding an ICC profile for the output intent.
- ▶ PDF/X-1a and PDF/X-3: by supplying the name of a standard output intent. The standard output intents are known internally to PDFLib; see PDFLib API Reference for a complete list of the standard output intent names and a description of the corresponding printing conditions. ICC profiles for these output intents are not required to be available locally. Additional standard output intents can be defined using the *StandardOutputIntent* resource category (see Section 3.1.3, »Resource Configuration and File Search«, page 56). It is the user's responsibility to add only those names as standard output intents which will be recognized by PDF/X-processing software. Standard output intents can be referenced as follows:

```
if (p.load_iccprofile("CGATS TR 001", "usage=outputintent") == -1)
{
    /* Error */
}
```

When creating PDF/X-3 output and using any of HKS, PANTONE, ICC-based, or Lab colors referencing the name of standard output intents is not sufficient, but an ICC profile of the output device must be embedded.

- ▶ PDF/X-4p and PDF/X-5pg: by referencing an external ICC profile for the output intent (the *p* in the name of the standard means that an external profile is referenced). Unlike standard output intents, the output intent ICC profile is not only referenced by name, but a strong reference is created which requires the ICC profile to be locally available when the document is generated. Although the ICC profile will not be embedded in the PDF output, it must nevertheless be available at PDF creation time to create a strong reference. The *urls* option must be provided with one or more valid URLs where the ICC profile can be found:

```
if (p.load_iccprofile("CGATS TR 001",
    "usage=outputintent urls={http://www.color.org}") == -1)
{
    /* Error */
}
```

A special rule applies to PDF/X-4/5: a CMYK output intent profile (i.e. loaded with *usage=outputintent*) can not be used for an ICCBased color space (i.e. loaded with *usage=iccbased*) in the same document. This requirement is mandated by the PDF/X standard, and applies only to CMYK profiles, but not to grayscale or RGB profiles. A similar condition applies to imported PDF/X-1/3 documents: if an imported page uses the same CMYK ICC profile as the generated document's output intent, it is rejected by *PDF_open_pdi_page()*.

Choosing a suitable PDF/X output intent. The PDF/X output intent is usually selected as a result of discussions between you and your print service provider who will take care of print production. If your printer cannot provide any information regarding the choice of output intent, you can use the standard output intents listed in Table 10.4 as a starting point (taken from the PDF/X FAQ).

Table 10.4 Suitable PDF/X output intents for common printing situations

	Europe	North America
Magazine ads	FOGRA28	CGATS TR 001
Newsprint ads	IFRA26	IFRA30
Sheet-fed offset	Dependent on paper stock: Types 1 & 2 (coated): FOGRA39 Type 3 (LWC): FOGRA45 Type 4 (uncoated): FOGRA47	Dependent on paper stock: Grades 1 and 2 (premium coated): FOGRA39 Grade 5: CGATS TR 001 Uncoated: FOGRA47
Web-fed offset	Dependent on paper stock: Type 1 & 2 (coated): FOGRA45 Type 4 (uncoated, white): FOGRA47 Type 5 (uncoated, yellowish): FOGRA30	Dependent on paper stock: Grade 5: CGATS TR 001 Uncoated (white): FOGRA47 Uncoated (yellowish): FOGRA30

10.3.4 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 7.2, »Importing PDF Pages with PDI«, page 182, for details). All imported documents must conform to a compatible PDF/X conformance level according to Table 10.5. 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 also acceptable. For all allowed combinations with PDF/X-4/5 output the following additional rule must be observed: if an imported page uses the same CMYK ICC profile as the generated document's output intent, it is rejected by `PDF_open_pdi_page()` since this would violate the PDF/X-4/5 standard.

If a particular PDF/X conformance level is configured in PDFlib and the imported documents adhere to one of the compatible levels, the generated output is guaranteed to conform to the selected PDF/X conformance level. Imported documents which do not adhere to one of the acceptable PDF/X levels will be rejected. If multiple PDF/X documents are imported, they must all have been prepared for the same output condition. For example, only documents with a CMYK output intent can be imported into a document which uses the same CMYK output intent.

While PDFlib can correct certain items, it is not intended to work as a full PDF/X validator or to enforce full PDF/X compatibility for imported documents. For example, PDFlib will not embed fonts which are missing from imported PDF pages, and does not apply any color correction to imported pages.

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

```
pdfxlevel = p.pcos_get_string(doc, "pdfx");
```

Table 10.5 Compatible PDF/X input levels for various PDF/X output levels

PDF/X output level	PDF/X level of the imported document							
	PDF/X-1a:2001	PDF/X-1a:2003	PDF/X-3:2002	PDF/X-3:2003	PDF/X-4	PDF/X-4p	PDF/X-5g	PDF/X-5pg
PDF/X-1a:2001	allowed							
PDF/X-1a:2003	allowed	allowed						
PDF/X-3:2002	allowed		allowed					
PDF/X-3:2003	allowed	allowed	allowed	allowed				
PDF/X-4	allowed	allowed	allowed	allowed	allowed	allowed		
PDF/X-4p	allowed	allowed	allowed	allowed	allowed	allowed ¹		
PDF/X-5g	allowed	allowed	allowed	allowed	allowed	allowed	allowed ²	allowed ²
PDF/X-5pg	allowed	allowed	allowed	allowed	allowed	allowed ¹	allowed ²	allowed ^{1,2}

1. `PDF_process_pdi()` with `action=copyoutputintent` will copy the reference to the external output intent ICC profile.
2. If the imported page contains referenced XObjects, `PDF_open_pdi_page()` will copy both proxy and reference to the target.

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()`.

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

```
ret = p.process_pdi(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. Copying the output intent works for imported PDF/A and PDF/X documents.

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 *usage=outputintent*.

10.4 PDF/A for Archiving

10.4.1 The PDF/A Standards

The PDF/A formats specified in the ISO 19005 standard strive to provide a consistent and robust subset of PDF which can safely be archived over a long period of time, or used for reliable data exchange in enterprise and government environments.

PDF/A Competence Center. PDFlib GmbH is a founding member of the PDF Association which hosts the PDF/A Competence Center as one of its activities. The aim of this organization is to promote the exchange of information and experience in the area of long-term archiving in accordance with ISO 19005. The members of the PDF/A Competence Center actively exchange information related to the PDF/A standard and its implementations, and conduct seminars and conferences on the subject. For more information refer to the PDF/A Competence Center section on the PDF Association's Web site at www.pdfa.org.



PDF/A-1a:2005 and PDF/A-1b:2005 as defined in ISO 19005-1. PDF/A is targeted at reliable long-time preservation of digital documents. The standard is based on PDF 1.4, and imposes some restrictions regarding the use of color, fonts, annotations, and other elements. There are two flavors of PDF/A-1, both of which can be created and processed with PDFlib:

- ▶ ISO 19005-1 Level B conformance (PDF/A-1b) ensures that the visual appearance of a document is preservable over the long term. Simply put, PDF/A-1b ensures that the document will look the same when it is processed some time in the future.
- ▶ ISO 19005-1 Level A conformance (PDF/A-1a) is based on level B, but adds properties which are known from the »Tagged PDF« flavor: it adds structure information and reliable text semantics in order to preserve the document's logical structure and natural reading order. Simply put, PDF/A-1a not only ensures that the document will look the same when it is processed some time in the future, but also that its contents (semantics) can be reliably interpreted and will be accessible to physically impaired users. PDFlib's support for PDF/A-1a is based on the features for producing Tagged PDF (see Section 10.5, »Tagged PDF«, page 262).

When PDF/A-1 (without any conformance level) is mentioned below, both conformance levels are meant.

Implementation basis. The following standards and documents form the basis for PDFlib's implementation of PDF/A-1:

- ▶ The PDF/A standard (ISO 19005-1:2005)
- ▶ Technical Corrigendum 1 (ISO 19005-1:2005/Cor 1:2007)
- ▶ Technical Corrigendum 2 (ISO 19005-1:2005/Cor.2:2011)
- ▶ All relevant TechNotes published by the PDF/A Competence Center.

10.4.2 Generating PDF/A-conforming Output

Cookbook Code samples for generating PDF/A can be found in the pdfa category of the PDFlib Cookbook.

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

- ▶ PDFlib will automatically take care of several formal settings for PDF/A, such as PDF version number and PDF/A conformance keys.
- ▶ The PDFlib client program must explicitly use certain function calls and options as detailed in Table 10.6.
- ▶ The PDFlib client program must refrain from using certain function calls and option settings as detailed in Table 10.7.
- ▶ Additional rules apply when importing pages from existing PDF/A-conforming documents (see Section 10.4.3, »Importing PDF/A Documents with PDI«, page 258).

If the PDFlib client program obeys to these rules, valid PDF/A output is guaranteed. If PDFlib detects a violation of the PDF/A creation rules it will throw an exception which must be handled by the application. No PDF output will be created in case of an error.

Required operations for PDF/A-1b. Table 10.6 lists all operations required to generate PDF/A-conforming output. The items apply to both PDF/A conformance levels unless otherwise noted. Not calling one of the required functions while in PDF/A mode will trigger an exception.

Table 10.6 Operations which must be applied for PDF/A-1 level A and B conformance

item	PDFlib function and option requirements for PDF/A conformance
conformance level	The pdfa option in PDF_begin_document() must be set to the required PDF/A conformance level, i.e. one of PDF/A-1a:2005 or PDF/A-1b:2005.
output condition (output intent)	PDF_load_iccprofile() with usage=outputintent or PDF_process_pdi() with action=copy-outputintent (but not both methods) must be called immediately after PDF_begin_document() if any of the device-dependent colors spaces Gray, RGB, or CMYK is used in the document. If an output intent is used, an ICC profile must be embedded (unlike PDF/X, unembedded standard output conditions are not sufficient in PDF/A). Use the embedprofile option of PDF_load_iccprofile() to embed a profile for a standard output condition.
fonts	The embedding option of PDF_load_font() (and other functions which accept this option) must be true. Note that embedding is also required for the PDF core fonts. The only exception to the embedding requirements applies to fonts which are exclusively used for invisible text (mainly useful for OCR results. This can be controlled with the optimizeinvisible option.
grayscale color	Grayscale images and PDF_setcolor() with a gray color space can only be used if the output condition is a grayscale, RGB, or CMYK device, or if the defaultgray option in PDF_begin_page_ext() has been set.
RGB color	RGB images and PDF_setcolor() with an RGB color space can only be used if the output condition is an RGB device, or the defaultrgb option in PDF_begin_page_ext() has been set.
CMYK color	CMYK images and PDF_setcolor() with a CMYK color space can only be used if the output condition is a CMYK device, or the defaultcmky option in PDF_begin_page_ext() has been set.

Prohibited and restricted operations. Table 10.7 lists all operations which are prohibited when generating PDF/A-conforming output. The items apply to both PDF/A conformance levels unless otherwise noted. Calling one of the prohibited functions while in PDF/A mode will trigger an exception. Similarly, if an imported PDF document does not conform to the current PDF/A output level, the corresponding PDI call will fail.

Table 10.7 Operations which must be avoided or are restricted to achieve PDF/A conformance

item	Prohibited or restricted PDFlib functions and options for PDF/A conformance
annotations	<i>PDF_create_annotation()</i> : annotations with type=FileAttachment and Movie must be avoided; for text annotations the zoom and rotate options must not be set to true. The annotcolor and interiorcolor options must only be used if an RGB output intent has been specified. The fillcolor option must only be used if an RGB or CMYK output intent has been specified, and a corresponding rgb or cmyk color space must be used. The opacity option must not be used.
attachments	<i>PDF_begin/end_document()</i> : the attachments option must be avoided.
form fields	<i>PDF_create_field()</i> and <i>PDF_create_fieldgroup()</i> for creating form fields must be avoided.
actions and JavaScript	<i>PDF_create_action()</i> : actions with type=Hide, Launch, Movie, ResetForm, ImportData, JavaScript must be avoided; for type=name only NextPage, PrevPage, FirstPage, and LastPage are allowed.
images	The OPI-1.3 and OPI-2.0 options and interpolate=true option in <i>PDF_load_image()</i> must be avoided.
ICC profiles	ICC profiles loaded explicitly with <i>PDF_load_iccprofile()</i> or implicitly with <i>PDF_load_image()</i> and ICC-tagged images must comply to ICC specification ICC.1:1998-09 and its addendum ICC.1A:1999-04 (internal profile version 2.x).
page sizes	There are no strict page size limits in PDF/A. However, it is recommended to keep the page size (width and height, and all box entries) in the range 3...14400 points (508 cm) to avoid problems with Acrobat.
templates	The OPI-1.3 and OPI-2.0 options in <i>PDF_begin_template_ext()</i> must be avoided.
transparency	Soft masks for images must be avoided: the masked option for <i>PDF_load_image()</i> must be avoided unless the mask refers to a 1-bit image. Images with implicit transparency (alpha channel) are not allowed; they must be loaded with the ignoremask option of <i>PDF_load_image()</i> . The opacityfill and opacitystroke options for <i>PDF_create_gstate()</i> must be avoided unless they have a value of 1; if blendmode is used it must be Normal. The opacity option in <i>PDF_create_annotation()</i> must be avoided.
transparency groups	The transparencygroup option of <i>PDF_begin/end_page_ext()</i> , <i>PDF_begin_template_ext()</i> , and <i>PDF_open_pdi_page()</i> is not allowed.
security	The userpassword, masterpassword, and permissions options in <i>PDF_begin_document()</i> must be avoided.
PDF version / compatibility	PDF/A is based on PDF 1.4. Operations that require PDF 1.5 or above (such as layers) must be avoided.
PDF import (PDI)	Imported documents must conform to a PDF/A level which is compatible to the output document, and must have been prepared according to a compatible output intent (see Table 10.10).
metadata	All predefined XMP schemas (see PDFlib API Reference) can be used. In order to use other schemas (extension schemas) the corresponding description must be embedded using the PDF/A extension schema container schema.
external content	The reference option in <i>PDF_begin_template_ext()</i> and <i>PDF_open_pdi_page()</i> must be avoided.
file size	The file size of the generated PDF document must not exceed 2 GB, and the number of PDF objects must be smaller than 8.388.607. See Section 3.1.5, »Large PDF Documents«, page 62, for more details about these limits.

Additional requirements and restrictions for PDF/A-1a. When creating PDF/A-1a, all requirements for creating Tagged PDF output as discussed in Section 10.5, »Tagged PDF«, page 262, must be met. In addition, some operations are not allowed or restricted as detailed in Table 10.8.

The user is responsible for creating suitable structure information; PDFlib does neither check nor enforce any semantic restrictions. A document which contains all of its text in a single structure element is technically correct PDF/A-1a, but violates the goal of faithful semantic reproduction, and therefore the spirit of PDF/A-1a.

Table 10.8 Additional requirements for PDF/A-1a conformance

item	PDFlib function and option requirements for PDF/A-1a conformance
Tagged PDF	<p>All requirements for Tagged PDF must be met (see Section 10.5, »Tagged PDF«, page 262). The following are strongly recommended:</p> <ul style="list-style-type: none">▶ The Lang option should be supplied in PDF_begin/end_document() to specify the default document language.▶ The Lang option should be specified properly in PDF_begin_item() for all content items which differ from the default document language.▶ Non-textual content items, e.g. images, should supply an alternate text description using the Alt option of PDF_begin_item().▶ Non-Unicode text, e.g. logos and symbols should have appropriate replacement text specified in the ActualText option of PDF_begin_item() for the enclosing content item.▶ Abbreviations and acronyms should have appropriate expansion text specified in the E option of PDF_begin_item() for the enclosing content item.
annotations	PDF_create_annotation(): the contents option is recommended.

Table 10.9 Additional operations which must be avoided or are restricted for PDF/A-1a conformance

item	Prohibited or restricted PDFlib functions and options or PDF/A-1a conformance
fonts	The monospace option, unicodemap=false, and autocidfont=false in PDF_load_font() (and other functions which accept these options) must be avoided.
PDF import (PDI)	Imported documents must conform to a PDF/A level which is compatible to the output document (see Table 10.10), and must have been prepared according to the same output intent.

Output intents. The output condition defines the intended target device, which is important for consistent color rendering. Unlike PDF/X, which strictly requires an output intent, PDF/A allows the specification of an output intent, but does not require it. An output intent is only required if device-dependent colors are used in the document. The output intent can be specified with an ICC profile. Output intents can be specified as follows:

```
icc = p.load_iccprofile("sRGB", "usage=outputintent");
```

As an alternative to loading an ICC profile, the output intent can also be copied from an imported PDF/A document using PDF_process_pdi() with the option action=copyoutput-intent.

Creating PDF/A and PDF/X at the same time. A PDF/A-1 document can at the same time conform to PDF/X-1a:2003, PDF/X-3:2003, or PDF/X-4 (but not to PDF/X-4p or PDF/X-5). In order to create such a combo file supply appropriate values for the *pdfa* and *pdfx* options of *PDF_begin_document()*, e.g.:

```
ret = p.begin_document("combo.pdf", "pdfx=PDF/X-4 pdfa=PDF/A-1b:2005");
```

The output intent must be the same for PDF/A and PDF/X, and must be specified as an output device ICC profile. PDF/X standard output conditions can only be used in combination with the *embedprofile* option.

10.4.3 Importing PDF/A Documents with PDI

Special rules apply when pages from an existing PDF document will be imported into a PDF/A-conforming output document (see Section 7.2, »Importing PDF Pages with PDI«, page 182, for details on PDI). All imported documents must conform to a PDF/A conformance level which is compatible to the current PDF/A mode according to Table 10.10.

Note *PDFlib* does not validate PDF input documents for PDF/A compliance, nor can it create valid PDF/A from arbitrary input PDFs.

If a certain PDF/A conformance level is configured in *PDFlib* and the imported documents adhere to a compatible level, the generated output is guaranteed to comply with the selected PDF/A conformance level. Documents which are incompatible to the current PDF/A level will be rejected in *PDF_open_pdi_document()*.

Table 10.10 Compatible PDF/A input levels for various PDF/A output levels

PDF/A output level	PDF/A level of the imported document	
	PDF/A-1a:2005	PDF/A-1b:2005
PDF/A-1a:2005	–	–
PDF/A-1b:2005	allowed	allowed

Cookbook A full code sample can be found in the Cookbook topic *pdfa/import_pdfa*.

If one or more PDF/A documents are imported, they must all have been prepared for a compatible output condition according to Table 10.11. The output intents in all imported documents must be identical or compatible; it is the user’s responsibility to make sure that this condition is met.

Table 10.11 Output intent compatibility when importing PDF/A documents

output intent of generated document	output intent of imported document			
	none	Grayscale	RGB	CMYK
none	yes	–	–	–
Grayscale ICC profile	yes	yes ¹	–	–
RGB ICC profile	yes	–	yes ¹	–
CMYK ICC profile	yes	–	–	yes ¹

1. Output intent of the imported document and output intent of the generated document must be identical

While PDFlib can correct certain items, it is not intended to work as a full PDF/A validator or to enforce full PDF/A conformance for imported documents. For example, PDFlib will not embed fonts which are missing from imported PDF pages.

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

```
pdfalevel = p.pcos_get_string(doc, "pdfa");
```

This statement will retrieve a string designating the PDF/A conformance level of the imported document if it conforms to a PDF/A level, or *none* otherwise. The returned string can be used to set the PDF/A conformance level of the output document appropriately, using the *pdfa* option in *PDF_begin_document()*.

Copying the PDF/A output intent from an imported document. In addition to querying the PDF/A conformance level you can also copy the PDF/A output intent from an imported document. Since PDF/A documents do not necessarily contain any output intent (unlike PDF/X which requires an output intent) you must first use pCOS to check for the existence of an output intent before attempting to copy it.

Cookbook A full code sample can be found in the Cookbook topic *pdfa/import_pdfa*.

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. Copying the output intent works for imported PDF/A and PDF/X documents.

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*. The output intent should be set immediately after *PDF_begin_document()*.

10.4.4 Color Strategies for creating PDF/A

The PDF/A requirements related to color handling may be confusing. The summary of color strategies in Table 10.12 can be helpful for planning PDF/A applications. The easiest approach which will work in many situations is to use the *sRGB* output intent profile, since it supports most common color spaces except CMYK. In addition, *sRGB* is known to PDFlib internally and thus doesn't require any external profile data or configuration. Color spaces may come from the following sources:

- ▶ Images loaded with *PDF_load_image()*
- ▶ Explicit color specifications using *PDF_setcolor()*
- ▶ Color specifications through option lists, e.g. in Textflows
- ▶ Interactive elements may specify border colors

In order to create black text output without the need for any output intent profile the CIELab color space can be used. The Lab color value (0, 0, 0) specifies pure black in a device-independent manner, and is PDF/A-conforming without any output intent profile (unlike DeviceGray, which requires an output intent profile). PDFlib will automatically initialize the current color to black at the beginning of each page. Depending on whether or not an ICC output intent has been specified, it will use the DeviceGray or Lab color space for selecting black. Use the following call to manually set Lab black color:

```
p.setcolor("fillstroke", "lab", 0, 0, 0, 0);
```

Table 10.12 PDF/A color strategies

output intent	color spaces which can be used in the document				
	CIELab ¹	ICCBased	Grayscale ²	RGB ²	CMYK ²
none	yes	yes	–	–	–
Grayscale ICC profile	yes	yes	yes	–	–
RGB ICC profile, e.g. sRGB	yes	yes	yes	yes	–
CMYK ICC profile	yes	yes	yes	–	yes

1. LZW-compressed TIFF images with CIELab color will be converted to RGB.
2. Device color space without any ICC profile

In addition to the color spaces listed in Table 10.12, spot colors can be used subject to the corresponding alternate color space. Since PDFlib uses CIELab as the alternate color space for the builtin HKS and PANTONE spot colors, these can always be used with PDF/A. For custom spot colors the alternate color space must be chosen so that it is compatible with the PDF/A output intent.

Note More information on PDF/A and color spaces can be found in Technical Note 0002 of the PDF/A Competence Center at www.pdfa.org.

10.4.5 XMP Document Metadata for PDF/A

PDF/A-1 heavily relies on the XMP format for embedding metadata in PDF documents. ISO 19005-1 refers to the XMP 2004 specification¹; older or newer versions of the XMP specification are not supported. PDF/A-1 supports two kinds of document-level metadata: a set of well-known metadata schemas called predefined schemas, and custom extension schemas. PDFlib will automatically create the required PDF/A conformance entries in the XMP as well as several common entries (e.g. *CreationDate*).

User-generated document metadata can be supplied with the *metadata* option of *PDF_begin/end_document()*. In PDF/A mode PDFlib verifies whether user-supplied XMP document metadata conforms to the PDF/A requirements. There are no PDF/A requirements for component-level metadata (e.g. page or image).

XMP metadata from imported PDF documents can be fetched from the input PDF via the pCOS path */Root/Metadata*.

Cookbook A full code sample can be found in the Cookbook topic *interchange/import_xmp_from_pdf*.

Predefined XMP schemas. PDF/A-1 supports all schemas in XMP 2004. These are called predefined schemas, and are listed in Table 10.13 along with their namespace URI and the preferred namespace prefix. Only those properties of predefined schemas must be used which are listed in XMP 2004. A full list of all properties in the predefined XMP schemas for PDF/A-1 is available from the PDF/A Competence Center.

XMP extension schemas. If your metadata requirements are not covered by the predefined schemas you can define an XMP extension schema. PDF/A-1 describes an extension mechanism which must be used when custom schemas are to be embedded in a PDF/A document. Table 10.14 summarizes the schemas which must be used for describing one or more extension schemas, along with their namespace URI and the required

1. See www.aiim.org/documents/standards/xmpspecification.pdf

Table 10.13 Predefined XMP schemas for PDF/A-1

Schema name and description (see XMP 2004 for details)	namespace URI	preferred namespace prefix
Adobe PDF schema	http://ns.adobe.com/pdf/1.3/	pdf
Dublin Core schema	http://purl.org/dc/elements/1.1/	dc
EXIF schema for EXIF-specific properties	http://ns.adobe.com/exif/1.0/	exif
EXIF schema for TIFF properties	http://ns.adobe.com/tiff/1.0/	tiff
Photoshop schema	http://ns.adobe.com/photoshop/1.0/	photoshop
XMP Basic Job Ticket schema	http://ns.adobe.com/xap/1.0/bj	xmpBJ
XMP Basic schema	http://ns.adobe.com/xap/1.0/	xmp
XMP Media Management schema	http://ns.adobe.com/xap/1.0/mm/	xmpMM
XMP Paged-Text schema	http://ns.adobe.com/xap/1.0/t/pg/	xmpTPg
XMP Rights Management schema	http://ns.adobe.com/xap/1.0/rights/	xmpRights

namespace prefix. Note that the namespace prefixes are required (unlike the preferred namespace prefixes for predefined schemas).

The details of constructing an XMP extension schema for PDF/A-1 are beyond the scope of this manual. Detailed instructions are available from the PDF/A Competence Center.

XMP document metadata packages can be supplied to the *metadata* options of *PDF_begin_document()*, *PDF_end_document()*, or both.

Cookbook Full code and XMP samples can be found in the *Cookbook* topics *pdfa/pdfa_extension_schema* and *pdfa/pdfa_extension_schema_with_type*.

Table 10.14 PDF/A-1 extension schema container schema and auxiliary schemas

Schema name and description	namespace URI ¹	required namespace prefix
PDF/A extension schema container schema: container for all embedded extension schema descriptions	http://www.aiim.org/pdfa/ns/extension/	pdfaExtension
PDF/A schema value type: describes a single extension schema with an arbitrary number of properties	http://www.aiim.org/pdfa/ns/schema#	pdfaSchema
PDF/A property value type: describes a single property	http://www.aiim.org/pdfa/ns/property#	pdfaProperty
PDF/A ValueType value type: describes a custom value type used in extension schema properties; only required if types beyond the XMP 2004 list of types are used.	http://www.aiim.org/pdfa/ns/type#	pdfaType
PDF/A field type schema: describes a field in a structured type	http://www.aiim.org/pdfa/ns/field#	pdfaField

1. Note that the namespace URIs are incorrectly listed in ISO 19005-1, and have been corrected in Technical Corrigendum 1.

10.5 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. PDFlib supports standard tag names (a list of standard tags can be found in the PDFlib API Reference) as well as custom tags. Custom tags require a role map which maps each custom tag to one of the standard tag names.

Cookbook Code samples regarding Tagged PDF issues can be found in the document_interchange category of the PDFlib Cookbook.

10.5.1 Generating Tagged PDF with PDFlib

Cookbook A full code sample can be found in the Cookbook topic document_interchange/starter_tagged.

Required operations. Table 10.15 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 10.15 Operations which must be applied for generating Tagged PDF

item	PDFlib function and option 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() should be set to specify the natural language of the document. It should 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 mappings. All text contents in Tagged PDF should have proper Unicode mappings to make sure that the document is accessible (e.g. can be read aloud by Software) and the text can be searched and extracted. Since PDFlib internally creates Unicode mappings for almost all font/encoding combinations, the PDF output will technically have Unicode mappings. However, the PUA values created for some symbols will not result in reusable text. In order to improve the searchability of text it is recommended to provide alternate text for the content via the *ActualText* or *Alt* options in *PDF_begin_item()*. In non-Tagged PDF mode this can be achieved with the *ActualText* option of *PDF_begin_mc()*.

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 the *PDFlib API Reference* 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 10.16 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 ASpan 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.

Recommended operations. Table 10.17 lists all operations which are optional, but recommended 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 10.17 Operations which are recommended for generating Tagged PDF

item	Recommended PDFlib functions and options for Tagged PDF compatibility
Unicode mappings	Provide alternate text for symbols via the ActualText or Alt options of PDF_begin_item().
hyphenation	Word breaks (separating words in two parts at the end of a line) should be presented using a soft hyphen character (U+00AD) 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.
Type 3 font properties	The familyname, stretch, and weight options of PDF_begin_font() should be supplied with reasonable values for all Type 3 fonts used in a Tagged PDF document.
interactive elements	Interactive elements, e.g. links, should be included in the document structure and made accessible if required, e.g. by supplying alternate text. The tab order for interactive elements can be specified with the taborder option of PDF_begin/end_document() (this is not necessary if the interactive elements are properly included in the document structure).

10.5.2 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 (p.begin_document("hello-tagged.pdf", "tagged=true") == -1)
    throw new Exception("Error: " + p.get_errmsg());

/* automatically create spaces between chunks of text */
p.set_parameter("autospace", "true");

/* open the first structure element as a child of the document structure root (=0) */
id = p.begin_item("P", "Title={Simple Paragraph}");

p.begin_page_ext(0, 0, "width=a4.width height=a4.height");
font = p.load_font("Helvetica-Bold", "unicode", "");

p.setfont(font, 24);
p.show_xy("Hello, Tagged PDF!", 50, 700);
p.continue_text("This PDF has a very simple");
p.continue_text("document structure.");

p.end_page_ext("");
p.end_item(id);
p.end_document("");
```


Generating Tagged PDF with Textflow. The Textflow feature (see Section 8.2, »Multi-Line Textflows«, page 201) 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 the complete contents of a single Textflow fitbox can 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).
- ▶ If you use the matchbox feature for creating links or other annotations in a Textflow it is difficult to maintain control over the annotation's position in the structure tree.

10.5.3 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 10.1. 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.

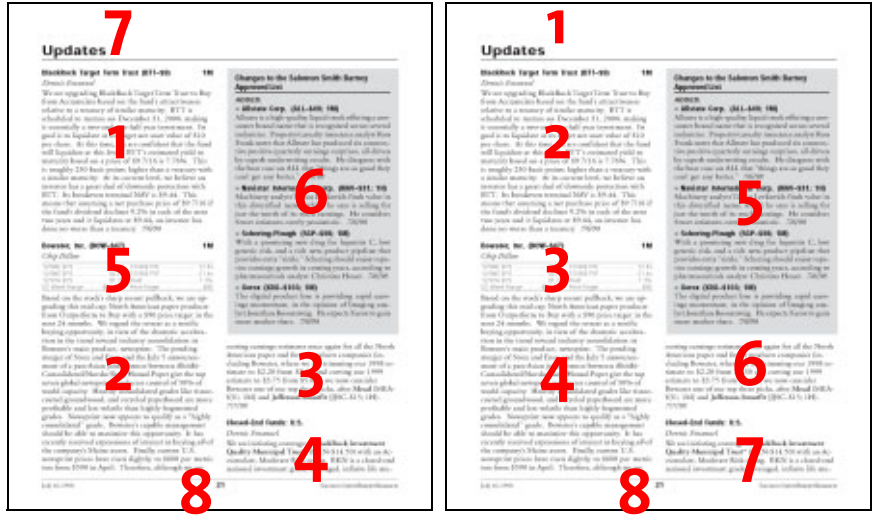
Generating page contents in logical order. From the logical structure point of view the page content should be created in the following order: left column, right column (on the lower right part of the page), table, insert, header and footer. The following pseudo code implements this ordering:

```
/* create page layout in logical structure order */

id_art = p.begin_item("Art", "Title=Article");

id_sect1 = p.begin_item("Sect", "Title={First Section}");
    /* 1 create top part of left column */
    p.set_text_pos(x1_left, y1_left_top);
    ...
    /* 2 create bottom part of left column */
    p.set_text_pos(x1_left, y1_left_bottom);
    ...
    /* 3 create top part of right column */
    p.set_text_pos(x1_right, y1_right_top);
```

Fig. 10.1
Creating a complex
page layout in logical
structure order (left)
and in visual order
(right). The right variant
uses item activation
for the first section
before continuing
fragments 4 and 6.



```

...
p.end_item(id_sect1);

id_sect2 = p.begin_item("Sect", "Title={Second Section}");
/* 4 create bottom part of right column */
p.set_text_pos(x2_right, y2_right);
...
/* second section may be continued on next page(s) */
p.end_item(id_sect2);

String optlist = "Title=Table parent=" + id_art;
id_table = p.begin_item("Table", optlist);
/* 5 create table structure and content */
p.set_text_pos(x_start_table, y_start_table);
...
p.end_item(id_table);

optlist = "Title=Insert parent=" + id_art;
id_insert = p.begin_item("P", optlist);
/* 6 create insert structure and content */
p.set_text_pos(x_start_table, y_start_table);
...
p.end_item(id_insert);

id_artifact = p.begin_item("Artifact", "");
/* 7+8 create header and footer */
p.set_text_pos(x_header, y_header);
...
p.set_text_pos(x_footer, y_footer);
...
p.end_item(id_artifact);

/* article may be continued on next page(s) */
...
p.end_item(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 = p.begin_item("Artifact", "");
/* 1 create header */
p.set_text_pos(x_header, y_header);
...
p.end_item(id_header);

id_art = p.begin_item("Art", "Title=Article");

id_sect1 = p.begin_item("Sect", "Title = {First Section}");
/* 2 create top part of left column */
p.set_text_pos(x1_left, y1_left_top);
...

String optlist = "Title=Table parent=" + id_art;
id_table = p.begin_item("Table", optlist);
/* 3 create table structure and content */
p.set_text_pos(x_start_table, y_start_table);
...
p.end_item(id_table);

/* continue with first section */
p.activate_item(id_sect1);
/* 4 create bottom part of left column */
p.set_text_pos(x1_left, y1_left_bottom);
...

optlist = "Title=Insert parent=" + id_art;
id_insert = p.begin_item("P", optlist);
/* 5 create insert structure and content */
p.set_text_pos(x_start_table, y_start_table);
...
p.end_item(id_insert);

/* still more contents for the first section */
p.activate_item(id_sect1);
/* 6 create top part of right column */
p.set_text_pos(x1_right, y1_right_top);
...
p.end_item(id_sect1);

id_sect2 = p.begin_item("Sect", "Title={Second Section}");
/* 7 create bottom part of right column */
p.set_text_pos(x2_right, y2_right);
...
/* second section may be continued on next page(s) */
p.end_item(id_sect2);

id_footer = p.begin_item("Artifact", "");
/* 8 create footer */
p.set_text_pos(x_footer, y_footer);
```

```

...
p.end_item(id_footer);

/* article may be continued on next page(s) */
...
p.end_item(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.

10.5.4 Using Tagged PDF in Acrobat

This section mentions observations which we made while testing Tagged PDF output in Adobe Acrobat 8/9/X. The observations below are mostly related to bugs or inconsistent behavior in Acrobat. Unless mentioned otherwise the observations relate to Acrobat 8, 9, and X. 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 Reflow in Acrobat 8 to crash, and can disable Reflow in Acrobat 9. For this reason it is recommended to put the text in a *Figure* element. This problem is fixed in Acrobat X.
- ▶ 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.
Structure items with mixed types of children (i.e., both page content sequences and non-inline structure elements) should be avoided since otherwise Reflow may fail.
- ▶ 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.
- ▶ Acrobat 8 and 9: 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.
- ▶ Reflow seems to have problems with PDF documents generated with the *topdown* option.

- ▶ If an activated item contains only content, but no structure children, Reflow may 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 cannot reflow pages with form fields (including digital signature fields), and will display a warning in this case.
- ▶ Acrobat 8: every reflow problem disables the Reflow feature and disables its menu item.

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. Some hints:

- ▶ Most importantly, all page content should be tagged. Content outside the tag structure will not be accessible, and will therefore be flagged by Acrobat's accessibility checker.
- ▶ In order to make form fields accessible, use the *tooltip* option of *PDF_create_field()* and *PDF_create_fieldgroup()*.
- ▶ If a page contains annotations, Acrobat reports that »tab order may be inconsistent with the structure order«.
- ▶ The *Alt* tag is ignored for *Figure* tags.

Export to other formats with Acrobat. Tagged PDF can significantly improve the result of saving PDF documents in formats such as XML or RTF with Acrobat.

- ▶ Acrobat 8/9: 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.
- ▶ Acrobat X extracts the content «as is»: the *Alt* and *ActualText* options are ignored, as well as *Private* and *NonStruct* tags.
- ▶ Acrobat 8/9 only: the contents 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 period 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.



11 PPS and the PDFlib Block Plugin

The PDFlib Personalization Server (PPS) supports a template-driven PDF workflow for variable data processing. Using the Block concept, imported pages can be populated with variable amounts of single- or multi-line text, images, or PDF graphics. 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

You can create and edit Blocks interactively with the PDFlib Block Plugin, convert existing PDF form fields to PDFlib Blocks with the form field conversion Plugin. Blocks can be filled with PPS. The results of Block filling with PPS can be previewed in Acrobat since the Block Plugin contains an integrated version of PPS.

Note Block processing requires the PDFlib Personalization Server (PPS). Although PPS is contained in all 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 interactively.

Cookbook Code samples regarding variable data and Blocks can be found in the blocks category of the PDFlib Cookbook.

11.1 Installing the PDFlib Block Plugin

The Block Plugin works with the following Acrobat versions:

- ▶ Acrobat 8/9/X/XI Standard, Professional, and Pro Extended on Windows
- ▶ Acrobat 8/9/X/XI Professional on the Mac.

The Plugin doesn't work with Acrobat Elements or any version of Adobe Reader.

Installing the PDFlib Block Plugin for Acrobat 8/9/X/XI on Windows. To install the PDFlib Block Plugin and the PDF form field conversion plugin in Acrobat, the plugin files must be placed in a subdirectory of 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*. A typical location of the plugin folder looks as follows:

C:\Program Files\Adobe\Acrobat 11.0\Acrobat\plug_ins\PDFlib Block Plugin

Installing the PDFlib Block Plugin for Acrobat 8/9/X/XI on the Mac. With Acrobat on the Mac the plugin folder is not directly 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 to a folder by double-clicking the disk image.
- ▶ Locate the *Adobe Acrobat* application icon in the Finder. It is usually located in a folder which has a name similar to the following:

- ▶ Single-click on the Acrobat application icon, open the context icon, and select *Show Package Contents*.
- ▶ In the Finder window that pops up navigate to the *Contents/Plug-ins* folder and copy the *PDFlib Block Plugin* folder which has been created in the first step into this folder.

Multi-lingual Interface. The PDFlib Block Plugin supports multiple languages in the user interface. Depending on the application language of Acrobat, the Block Plugin will choose its interface language automatically. Currently English, German and Japanese interfaces are available. If Acrobat runs in any other language mode, the Block Plugin will use the English interface.

Troubleshooting. If the PDFlib Block Plugin doesn't seem to work check the following:

- ▶ Make sure that in *Edit, Preferences, [General...], General* the box *Use only certified plug-ins* is unchecked. The plugins will not be loaded if Acrobat is running in Certified Mode.
- ▶ Some PDF forms created with Adobe Designer may prevent the Block Plugin as well as other Acrobat plugins from working properly since they interfere with Acrobat's internal security model. For this reason we suggest to avoid Designer's static PDF forms, and only use dynamic PDF forms as input for the Block Plugin.

11.2 Overview of the Block Concept

11.2.1 Separation of Document Design and Program Code

PDFlib Blocks make it easy to place variable text, images, or graphics on imported pages. In contrast to simple PDF pages, pages with Blocks intrinsically carry information about the required processing which will be performed later on the server side. The Block concept separates the following tasks:

- ▶ The designer creates the page layout and specifies the location of variable page 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.
- ▶ The 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. PPS 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 can move the Block with name of the addressee in a mailing to a different location on the page, or change the font size. 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.

As an intermediate step Block filling can be previewed in Acrobat to accelerate the development and test cycles. Block previews are based on default data (e.g. a string or an image file name) which is specified in the Block definitions.

11.2.2 Block Properties

The behavior of Blocks can be controlled with Block properties. Properties are assigned to a Block with the Block Plugin.

Standard Block properties. 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 by PPS. The name is an arbitrary string which identifies the Block, such as *firstname*, *lastname*, or *zipcode*. PPS supports different kinds of Blocks:

- ▶ *Textline Blocks* hold a single line of textual data which will be processed with the Textline output method in PPS.
- ▶ *Textflow Blocks* hold one or more lines of textual data. Multi-line text will be formatted with the Textflow formatter in PPS. Textflow Blocks can be linked so that one Block holds the overflow text of the previous Block (see »Linking Textflow Blocks«, page 291).
- ▶ *Image Blocks* hold a raster image. This is similar to placing a TIFF or JPEG file in a DTP application.
- ▶ *PDF Blocks* hold arbitrary PDF graphics imported from a page in another PDF document. This is similar to placing a PDF page in a DTP application.

Blocks can carry a number of standard properties depending on their type. Properties can be created and modified with the Block Plugin (see Section 11.3.2, »Editing Block Properties«, page 280). A full list of standard Block properties can be found in Section 11.6, »Block Properties«, page 294. For example, a text Block can specify the font and size of the text, an image or PDF Block can specify the scaling factor or rotation PPS offers dedicated functions for processing the Block types, e.g. *PDF_fill_textblock()*. These functions search a placed PDF page for a Block by its name, analyze its properties, and place client-supplied data (single- or multi-line text, raster image, or PDF page) on the new page according to the specified Block properties. The programmer can override Block properties by specifying the corresponding options to the Block filling functions.

Properties for default contents. Special Block properties can be defined which hold the default contents of a Block, i.e. the text, image or PDF contents which will be placed in the Block if no variable data has been supplied to the Block filling functions, or in situations where the Block contents are currently constant, but may change in the next print run.

Default properties are also used by the Preview feature of the Block Plugin (see Section 11.4, »Previewing Blocks in Acrobat«, page 286).

Custom Block properties. Standard Block properties make it possible to quickly implement variable data processing applications, but they are restricted to the set of properties which are internally known to PPS and can automatically be processed. In order to provide more flexibility, the designer can also assign custom properties to a Block. These can be used to extend the Block concept in order to match the requirements of more advanced variable data processing applications.

There are no rules for custom properties since PPS will not process custom properties in any way, except making them available to the client. The client code can retrieve the value of custom properties and process it as appropriate. Based on a custom property of a Block the application 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.

11.2.3 Why not use PDF Form Fields?

Experienced Acrobat users may ask why we implemented a new Block concept instead of relying on the existing form field mechanism 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 combine PDF forms and PDFlib Blocks with the form field conversion plugin (see Section 11.3.4, »Converting PDF Form Fields to PDFlib Blocks«, page 282).

Although there are many parallels between both concepts, PDFlib Blocks offer several advantages over PDF form fields as detailed in Table 11.1.


Table 11.1 Comparison of PDF form fields and PDFlib Blocks

feature	PDF form fields	PDFlib Blocks
<i>design objective</i>	<i>for interactive use</i>	<i>for automated filling</i>
<i>typographic features (beyond choice of font and font size)</i>	–	<i> Kerning, word and character spacing, underline/overline/strikeout</i>
<i>OpenType layout features</i>	–	<i>dozens of OpenType layout features, e.g. ligatures, swash characters, oldstyle figures</i>
<i>complex script support</i>	<i>limited</i>	<i>shaping and bidirectional formatting, e.g. for Arabic and Devanagari</i>
<i>font control</i>	<i>font embedding</i>	<i>font embedding and subsetting, encoding</i>
<i>text formatting controls</i>	<i>left-, center-, right-aligned</i>	<i>left-, center-, right-aligned, justified; various formatting algorithms and controls; inline options can be used to control the appearance of text</i>
<i>change font or other text attributes within text</i>	–	<i>yes</i>
<i>merged result is integral part of PDF page description</i>	–	<i>yes</i>
<i>users can edit merged field contents</i>	<i>yes</i>	<i>no</i>
<i>extensible set of properties</i>	–	<i>yes (custom Block properties)</i>
<i>use image files for filling</i>	–	<i>BMP, CCITT, GIF, PNG, JPEG, JBIG2, JPEG 2000, TIFF</i>
<i>color support</i>	<i>RGB</i>	<i>grayscale, RGB, CMYK, Lab, spot color (HKS and Pantone spot colors integrated in the Block Plugin)</i>
<i>PDF/X and PDF/A</i>	<i>PDF/X: no PDF/A: restricted</i>	<i>yes (both Block container and merged results)</i>
<i>graphics and text properties can be overridden upon filling</i>	–	<i>yes</i>
<i>transparent contents</i>	–	<i>yes</i>
<i>Text Blocks can be linked</i>	–	<i>yes</i>

11.3 Editing Blocks with the Block Plugin

11.3.1 Creating Blocks

Activating the Block tool. The 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 the following ways:

- ▶ By clicking the Block icon  in the *Tools, Advanced Editing* pane (Acrobat X/XI) or the *Advanced Editing* toolbar (Acrobat 9). If Acrobat does not display this toolbar you can enable it via *View, Tools, Plug-In Advanced Editing* (Acrobat X) or *View, Toolbars, Advanced Editing* (Acrobat 9).
- ▶ Via the menu item *PDFlib Blocks, PDFlib Block Tool*.

Creating and modifying Blocks. When the Block tool is active you can drag the cross-hair pointer to create a Block at the desired position on the page and with the desired size. Blocks are always rectangular with edges parallel to the page edges (use the *rotate* property for Block contents which are not parallel to the page edges). After dragging a Block rectangle the Block properties dialog appears where you can edit the properties of the Block (see Section 11.3.2, »Editing Block Properties«, page 280). The Block tool automatically creates a synthetic Block name which can be changed in the properties dialog. Block names must be unique on a page, but can be repeated on another page.

You can change the Block type in the top area to one of *Textline*, *Textflow*, *Image*, or *PDF*. Different colors are used for representing the Block types (see Figure 11.1). The *Block Properties* dialog hierarchically organizes the properties in groups and subgroups depending on the 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, moving, deleting, or editing Properties, work with one or more selected Blocks. You can select Blocks with the Block tool as follows:

- ▶ To select a single Block simply click on it.
- ▶ To select multiple Blocks hold down the Shift key while clicking on the second and subsequent Block.
- ▶ 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.

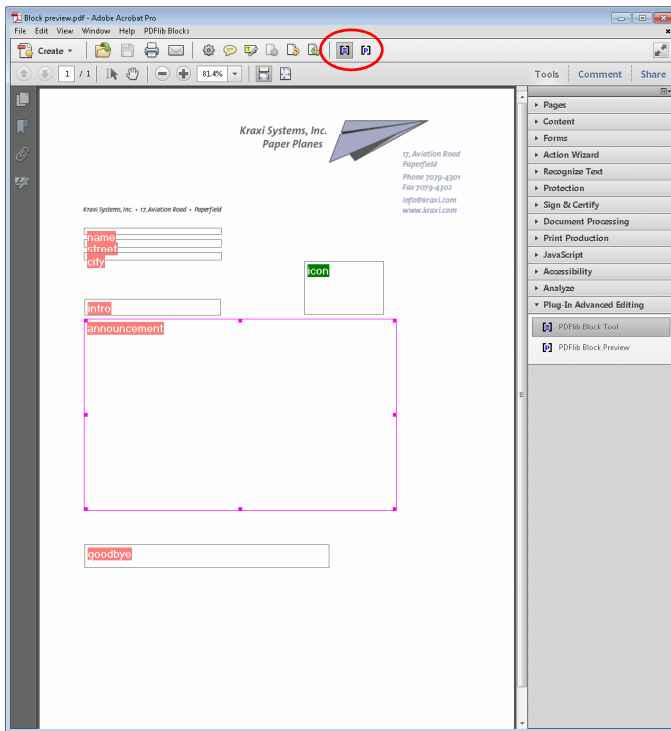


Fig. 11.1
Visualization of Blocks

If you right-click (or Ctrl-click on the Mac) an area on the page where no Block is located the context menu contains entries for creating a Block Preview and for configuring the Preview feature.

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 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 in small increments by using the arrow keys.

Alternatively, you can enter numerical Block coordinates in the properties dialog. The origin of the coordinate system is in the upper left corner of the page. The coordinates will be displayed in the unit which is currently selected in Acrobat:

- ▶ To change the display units in Acrobat 9/X/XI proceed as follows: go to *Edit, Preferences, [General...], Units & Guides, Page & Ruler Units* and choose one of Points, Inches, Millimeters, Picas, Centimeters.
- ▶ To display cursor coordinates use *View, Show/Hide, Cursor Coordinates* (Acrobat X/XI) or *View, Cursor Coordinates* (Acrobat 9).

Note that the selected unit will only affect the *Rect* property, but not any other numerical properties (e.g. *fontSize*).

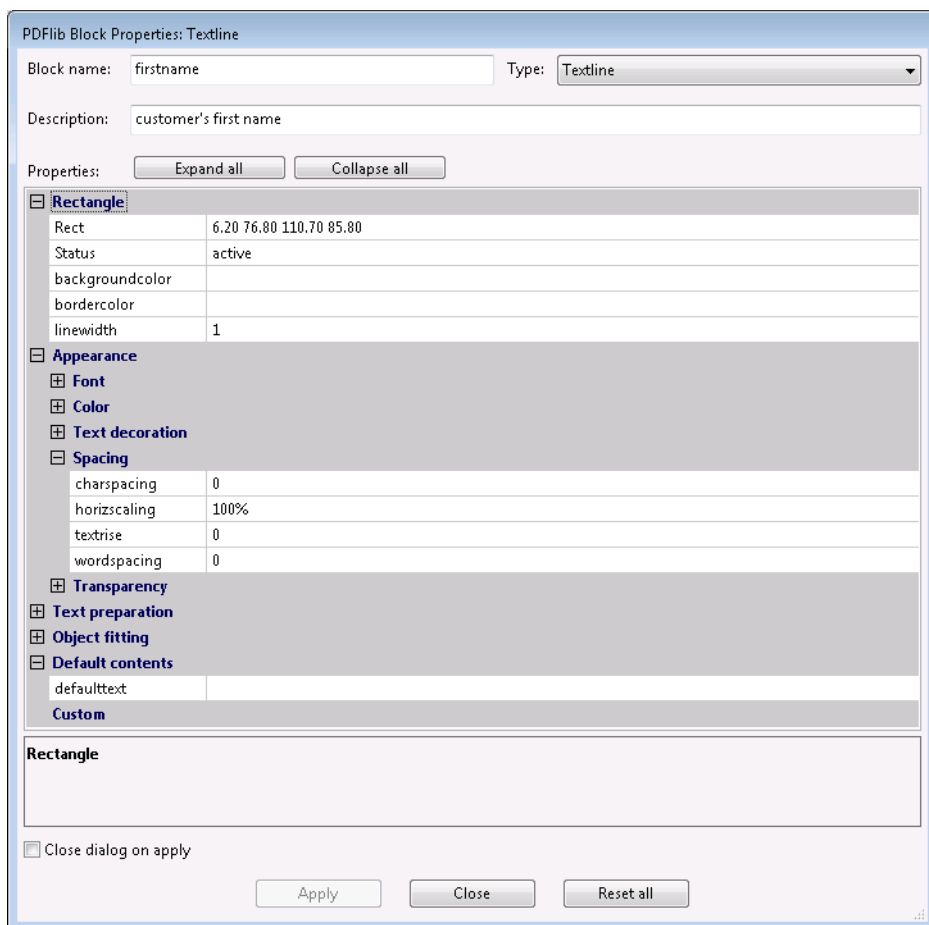


Fig. 11.2
The Block properties dialog

Using a grid to position Blocks. You can take advantage of Acrobat's grid feature for precisely positioning and resizing Blocks:

- ▶ Display the grid: *View, Show/Hide, Rulers & Grids, Grid* (Acrobat X/XI) or *View, Grid* (Acrobat 9);
- ▶ Enable grid snapping: *View, Show/Hide, Rulers & Grids, Snap to Grid* (Acrobat X/XI) or *View, Snap to Grid* (Acrobat 9);
- ▶ Change the grid (see Figure 11.3): go to *Edit, Preferences, [General...], Units & Guides*. Here you can change the spacing and position of the grid as well as the color of the grid lines.

If *Snap to Grid* is enabled the size and position of Blocks will be aligned with the configured grid. *Snap to Grid* affects newly generated Blocks as well as existing Blocks which are moved or resized with the Block tool.

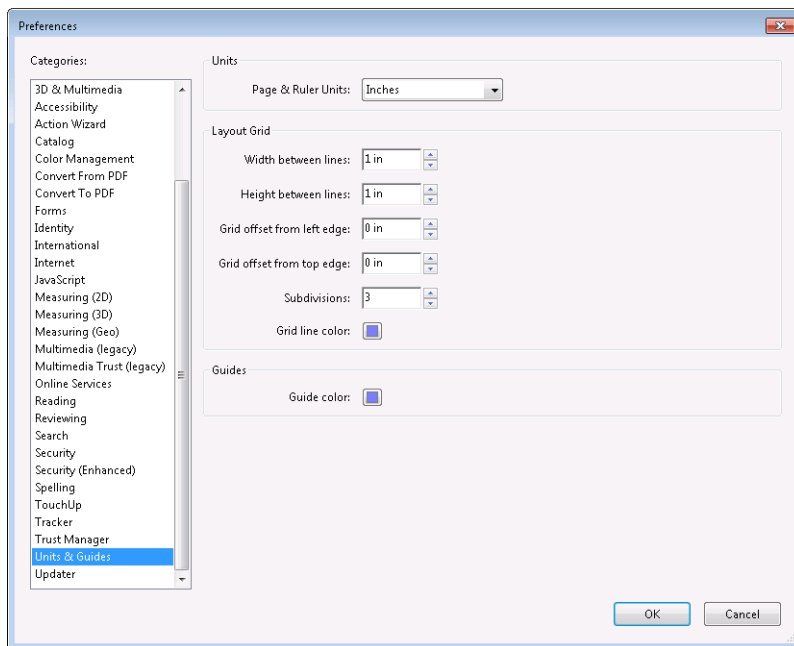


Fig. 11.3
Grid preferences
in Acrobat

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 same size and location as 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 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, not only individual lines.

Automatically detect font properties. The Block Plugin can analyze the underlying font which is present at the location where a Textline or Textflow 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 if 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, resizing, or deleting. 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 edit its properties.

11.3.2 Editing Block Properties

When you create a new Block, double-click an existing one, or choose *Properties* from a Block's context menu, the properties dialog will appear where you can edit all settings related to the selected Block (see Figure 11.2). As detailed in Section 11.6, »Block Properties«, page 294, there are several groups of properties available, subject to the Block type.

The *Apply* button will only be enabled if you changed one or more properties in the dialog. The *Apply* button will be inactive for locked Blocks.

Note Some properties may be inactive depending on the Block type and certain property settings. For example, the property subgroup *Ruler tabs* for *hortabmethod=ruler* where you can edit *tabulator settings* is enabled only if the *hortabmethod* property in the *Textflow* group is set to *ruler*.

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 a drop-down list (e.g. *fitmethod*, *orientate*), or select a font, color value or file name by clicking the »...« button at the right-hand side of the dialog (e.g. *backgroundcolor*, *defaultimage*). For the *fontname* property you can either choose from the list of fonts installed on the system 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 with PPS.

Modified properties will be displayed in bold face in the Block Properties dialog. If any of the properties in a Block has been modified, the suffix (*) will be appended to the displayed Block name. When you are done editing properties click the *Apply* button to update the Block. 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 will always select the topmost Block. In this 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 this way, the next action within its area will not affect other Blocks, but only the selected one. For example, press *Enter* to edit the selected Block's properties. This way Block properties can easily be edited even for Blocks which are partially or completely covered by other Blocks.

Using and restoring repeated values of Block properties. In order to save some amount of typing and clicking, the Block tool remembers 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 resets most Block properties to their respective default values. The following items remain unmodified:

- ▶ the *Name*, *Type*, *Rect*, and *Description* properties;
- ▶ all custom properties.

Note Do not confuse the default values of standard Block properties with the defaulttext, default-image, and defaultpdf properties which hold placeholder data for generating previews (see »Default Block contents«, page 286).

Editing multiple Blocks at once. Editing the properties of multiple Blocks at once is a big time saver. You can select multiple Blocks as follows:

- ▶ Activate the Block tool via the menu item *PDFlib Blocks, PDFlib Block Tool*.
- ▶ Click on the first Block to select it. The first selected Block will be the master Block. Shift-click other Blocks to add them to the set of selected Blocks. Alternatively, click *Edit, Select All* to select all Blocks on the current page.
- ▶ Double-click on any of the Blocks to open the Block Properties dialog. The Block where you double-click will be the new master Block.
- ▶ Alternatively, you can click on a single Block to designate it as master Block, and then press the *Enter* key to open the Block Properties dialog.

The Properties dialog displays only the subset of properties which apply to all selected Blocks. The dialog will be populated with property values taken from the master Block. Closing the dialog with *Apply* copies its current contents to all selected Blocks, i.e. the values of the master Block with possible manual changes applied in the dialog. This behavior can be used to copy Block properties from a particular Block to one or more other Blocks.

The following standard properties can not be shared, i.e. they can not be edited for multiple Blocks at once:

Name, Description, Subtype, Type, Rect, Status

Custom properties also cannot be shared among Blocks.

11.3.3 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 between 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 using standard copy/paste operations
- ▶ export Blocks to a new file (with empty pages) or to an existing document (apply the Blocks to existing pages)
- ▶ import Blocks from another document

In order to update the page contents while maintaining Block definitions you can replace the underlying page(s) while keeping the Blocks. Use *Document, Replace Pages...* in Acrobat for this purpose.

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 this key is pressed. A copied Block has the same properties as the original Block, with the exception of its name and position which will automatically be adjusted in the new Block.

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.
- ▶ Navigate to the target page (if necessary).
- ▶ Use Ctrl-V (on Windows) or Cmd-V (on the Mac) or *Edit, Paste* to paste the Blocks from the clipboard to the current page.

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 Blocks on this Page*) and the range of target pages to which you want to duplicate the selected 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 of the new PDF with 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 *Repeat 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.

11.3.4 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 complex PDF forms which you want to fill automatically with PPS 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 one or more 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 *Textline* or *Textflow*. 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 11.3.

Multiple form fields with the same name. Multiple form fields on the same page are allowed to have the same name, while Block names must be unique on a page. When converting form fields to Blocks a numerical suffix will therefore be added to the name of generated Blocks in order to create unique Block names (see also »Associating form fields with corresponding Blocks«, page 283).

Note that due to a problem in Acrobat the field attributes of form fields with the same names are not reported correctly. If multiple fields have the same name, but different attributes these differences will not be reflected in the generated Blocks. The Conversion process will issue a warning in this case and provide the names of affected form fields. In this case you should carefully check the properties of the generated Blocks.

Associating form fields with corresponding Blocks. Since the form field names will be modified when converting multiple fields with the same name (e.g. radio buttons) it is difficult to reliably identify the Block which corresponds to a particular form field. This is especially important when using an FDF or XFDF file as the source for filling Blocks such that the final result resembles the filled form.

In order to solve this problem the AcroFormConversion plugin records details about the original form field as custom properties when creating the corresponding Block. Table 11.2 lists the custom properties which can be used to reliably identify the Blocks; all properties have type *string*.

Table 11.2 Custom properties for identifying the original form field corresponding to the Block

custom property	meaning
PDFlib:field:name	Fully qualified name of the form field
PDFlib:field:pagenumber	Page number (as a string) in the original document where the form field was located
PDFlib:field:type	Type of the form field; one of pushbutton, checkbox, radiobutton, listbox, combobox, textfield, signature
PDFlib:field:value	(Only for type=checkbox) Export value of the form field

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 plugin 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.

Table 11.3 Conversion of PDF form fields to PDFlib Blocks

PDF form field attribute...	...will be converted to the PDFlib Block property
all fields	
Position	Rect
Name	Name
Tooltip	Description
Appearance, Text, Font	fontname
Appearance, Text, Font Size	fontsize; auto font size will be converted to a fixed font size of 2/3 of the Block height, and 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	strokecolor and fillcolor
Appearance, Border, Border Color	bordercolor
Appearance, Border, Fill Color	backgroundcolor
Appearance, Border, Line Thickness	linewidth: Thin=1, Medium=2, Thick=3
General, Common Properties, Form Field	Status: Visible=active Hidden=ignore Visible but doesn't print=ignore Hidden but printable=active
General, Common Properties, Orientation	orientate: 0=north, 90=west, 180=south, 270=east
text fields	
Options, Default Value	defaulttext
Options, Alignment	position: Left={left center} Center={center center} Right={right center}
Options, Multi-line	checked creates Textflow Block unchecked creates a Textline Block
radio buttons and check boxes	
If »Check box/Button is checked by default« is selected: Options, Check Box Style or Options, Button Style	defaulttext: Check=4 Circle=l Cross=8 Diamond=u Square=n Star=H (these characters represent the respective symbols in the ZapfDingbats font)
list boxes and combo boxes	
Options, Selected (default) item	defaulttext
buttons	
Options, Icon and Label, Label	defaulttext

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 until the conversion is finished.

11.3.5 Customizing the Block Plugin User Interface with XML

The following aspects of the Block Plugin user interface can be controlled via the XML configuration file. The XML file must be located in the Block Plugin directory. The default configuration file *default.PPSoptions* is loaded at startup. Please refer to the default configuration file which is installed with the PDFlib Block Plugin:

- ▶ The element */Block_Plugin/MainDialog/CloseOnApply* controls the initial status of the *Close dialog on apply* checkbox in the Block properties dialog. This checkbox determines whether the Block Properties dialog will be kept open after creating a Block or modifying Block properties.
- ▶ The element */Block_Plugin/FontDialog/ShowBaseFonts* controls whether the base 14 fonts will be displayed in the font list of the Block Properties dialog (property group *Appearance*, property *fontname*) in addition to the fonts installed on the system.
- ▶ The element */Block_Plugin/Command/ControlByClick* controls the initial status of the menu item *PDFlib Blocks, Click object to define Block*.
- ▶ The element */Block_Plugin/Command/DetectFonts* controls the initial status of the menu item *PDFlib Blocks, Detect underlying font and color*.
- ▶ (Unsupported) The element */Block_Plugin/Command/KeyAccelerator* with the possible values *control* (which designates the Ctrl key on Windows and the Cmd key on the Mac), *control+shift* or *none* controls the accelerator key for the following keyboard shortcuts:

C (copy), I (Block Properties dialog), V (paste), X (cut)

This element has an effect only in the default configuration file *default.PPSoptions* since keyboard shortcuts cannot be changed at runtime. If this entry is absent, no accelerators will be available. The default is *control*.

11.4 Previewing Blocks in Acrobat

Note You can try the Preview feature with the `block_template.pdf` document in the PDFlib distribution. The required resources (e.g. font and image) are also included in the distribution.

PDFlib Blocks will be processed by PPS where the Block filling process can be customized regarding the data sources (e.g. text from a database, image files on disk) as well as visual and interactive aspects of the generated documents. This process is detailed in Section 11.5, »Filling Blocks with PPS«, page 290.

However, the Block Plugin contains an integrated version of PPS which can be used to generate Preview versions of the filled Blocks interactively in Acrobat without any programming. Although this Preview feature cannot offer the same flexibility as custom programming, it provides a quick overview of Block filling results. The Block Preview can be used for improving the position and size of Blocks as well as for checking the Block properties (e.g. font name and size). You can change the Blocks and create a new Preview until you are satisfied with the results shown in the Preview. Previews can be generated for the current page or the whole document.


The Preview will always be shown in a new PDF document. The original document (which contains the Blocks) will not be modified by generating a Preview. You can save or discard the generated Preview documents according to your requirements. The original Block container document is not affected by the Preview.

Default Block contents. Since the server-side data sources (e.g. a database) for the text, image, or PDF contents of a Block is not available in the plugin, the Preview feature will always use the Block's default contents, i.e. the data specified in the *defaulttext*, *defaultimage*, or *defaultpdf* properties. Usually, a sample data set will be used as default data which is representative for the real Block contents used with PPS. Blocks without any default contents will be ignored when generating the Preview, as well as Blocks with *Status=ignoredefault*.

The default properties are empty for new Blocks. Before using the Preview feature you must fill the *defaulttext*, *defaultimage*, or *defaultpdf* properties (depending on the Block type) in the *Default contents* property group.

Note Entering default text for symbolic fonts can be a bit tricky; see »Using symbolic fonts for default text«, page 289, for details.

Generating Block Previews. You can create Block Previews with one of the following methods:

- ▶ Via the menu item *PDFlib Blocks, Preview, Generate Preview*.
- ▶ By clicking the PDFlib Block Preview icon  in the *Tools, Advanced Editing* pane (Acrobat X/XI) or the *Advanced Editing* toolbar (Acrobat 9). If Acrobat does not display this toolbar you can enable it via *View, Tools, Advanced Editing* (Acrobat X/XI) or *View, Toolbars, Advanced Editing* (Acrobat 9).
- ▶ If the Block tool is active you can right-click outside of any Block to bring up a context menu with the entries *Generate Preview* and *Preview Configuration*.

The Previews will be created based on the PDF file on disk. Any changes that you may have applied in Acrobat will only be reflected in the Preview if the Block PDF has been saved to disk using *File, Save* or *File, Save As...*. You can identify modified Blocks by the as-

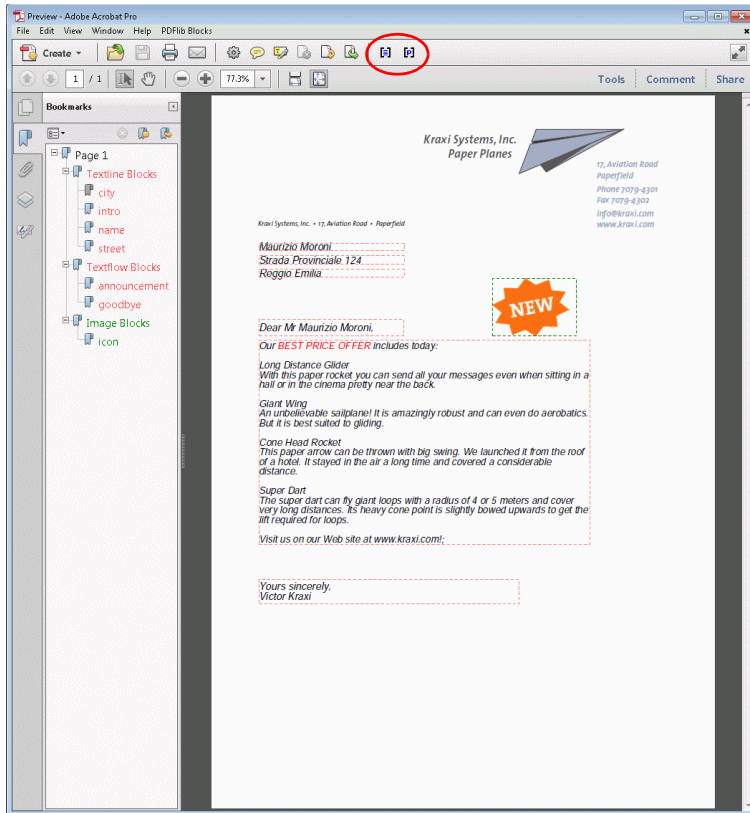


Fig. 11.4
Preview PDF for the container
document shown in Figure 11.1.
It contains Block info layers and
annotations

terisk after the Block name. The Preview feature can be configured to save the Block PDF automatically before creating a Preview. This way you can make sure that interactive changes will immediately be reflected in the preview.

Configuring the Preview. Several aspects of Block Preview creation and the underlying PPS operation can be configured via *PDFlib Blocks, Preview, Preview Configuration...*:

- ▶ Preview for the current page or the full document;
- ▶ Output directory for the generated Preview documents;
- ▶ Automatically save the Block PDF before creating the Preview;
- ▶ Add Block info layers and annotations;
- ▶ Clone PDF/A-1b or PDF/X status; since these standards restrict the use of layers and annotations the *Block info layers and annotations* option is mutually exclusive with this option.
- ▶ The *Advanced PPS options* dialog can be used to specify additional option lists for PPS functions according to the PPS API. For example, the *searchpath* option for *PDF_set_option()* can be used to specify a directory where fonts or images for Block filling are located. It is recommended to specify advanced options in cooperation with the programmer who writes the PPS code.

The Preview configuration can be saved to a disk file and later be reloaded.

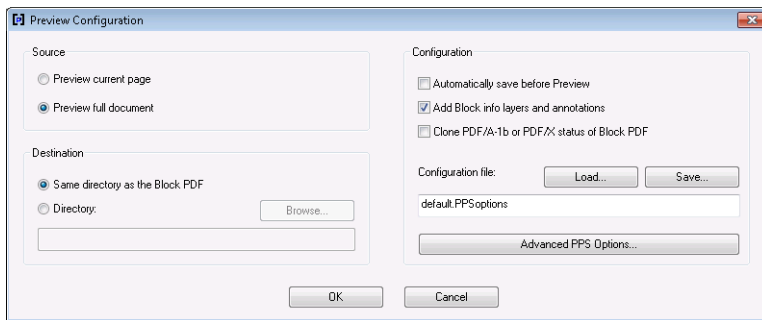


Fig. 11.5
Block Preview
configuration

Information provided with the Preview. The generated Preview documents contain the original page contents (the background), the filled Blocks, and optionally various other pieces of information. This information can be useful for checking and improving Blocks and PPS configuration. The following items will be created for each active Block with default contents:

- ▶ **Error markers:** Blocks which could not be filled successfully are visualized by a crossed-out rectangle so that they can easily be identified. Error markers will always be created if a Block couldn't be processed.
- ▶ **Bookmarks:** The processed Blocks will be summarized in bookmarks which are structured according to the page number, the Block type, and possible errors. Bookmarks can be displayed via *View, Navigation Panels, Bookmarks*. Bookmarks will always be created.
- ▶ **Annotations:** For each processed Block an annotation will be created on the page in addition to the actual Block contents. The annotation rectangle visualizes the original Block boundary (depending on the default contents and filling mode this may differ from the boundary of the Block contents). The annotation contains the name of the Block and an error message if the Block couldn't be filled. Annotations are generated by default, but can be disabled in the Preview configuration. Since the use of annotations is restricted in the PDF/A-1 and PDF/X standards, annotations are not created if the *Clone PDF/A-1b or PDF/X status of Block PDF* option is enabled.
- ▶ **Layers:** The page contents will be placed on layers to facilitate analysis and debugging. A separate layer will be created for the page background (i.e. the contents of the original page), each Block type, error Blocks which couldn't be filled, and the annotations with Block information. If a layer remains empty (e.g. no errors occurred) it will not be created. The layer list can be displayed via *View, Navigation Panels, Layers*. By default, all layers on the page will be displayed. In order to hide the contents of a layer click on the eye symbol to the left of the layer name. Layer creation can be disabled in the Preview configuration. Since the use of layers is restricted in the PDF/A-1 and PDF/X standards, layers are not created if the *Clone PDF/A-1b or PDF/X status of Block PDF* option is enabled.

Cloning the PDF/A or PDF/X status. The *Clone PDF/A-1b or PDF/X status of Block PDF* configuration is useful when PDF output according to one of the PDF/A or PDF/X standards must be created. Clone mode can be enabled if the input conforms to one of the following standards:

PDF/A-1b:2005
PDF/X-1a:2001, PDF/X-1a:2003

PDF/X-3:2002, PDF/X-3:2003
PDF/X-4, PDF/X-4p
PDF/X-5g, PDF/X-5pg

When Previews are created in clone mode, PPS will duplicate the following aspects of the Block PDF in the generated Preview:

- ▶ the PDF standard identification;
- ▶ output intent condition;
- ▶ page sizes including all page boxes;
- ▶ XMP document metadata.

When cloning standard-conforming PDF documents all Block filling operations must conform to the respective standard. For example, if no output intent is present RGB images without ICC profile can not be used. Similarly, all used fonts must be embedded. The full list of requirements can be found in Section 10.3, »PDF/X for Print Production«, page 247, and Section 10.4, »PDF/A for Archiving«, page 254. If a Block filling operation in PDF/A or PDF/X cloning mode would violate the selected standard (e.g. because a default image uses RGB color space, but the document does not contain a suitable output intent) an error message pops up and no Preview will be generated. This way users can catch potential standard violations very early in the workflow.

Using symbolic fonts for default text. Two methods are available to supply default text for Blocks with symbolic fonts:

- ▶ Working with 8-bit legacy codes, e.g. as shown in the Windows character map application: supply the 8-bit codes for the *defaulttext* either by entering the corresponding 8-bit character literally (e.g. by copy/pasting from the Windows character map) or as a numerical escape sequence. In this case you must keep the default value of the *charref* property in the *Text preparation* property group as *false* and can not work with character references. For example, the following default text will produce the »smiley« glyph from the symbolic Wingdings font if *charref=false*:

```
J  
\x4A  
\112
```

- ▶ Working with the Unicode values or glyph names used in the font: set the *charref* property in the *Text preparation* property group to *true* and supply character references or glyph name references for the symbols (see Section 4.5.2, »Character References«, page 108). For example, the following default text will produce the »smiley« glyph from the symbolic Wingdings font if *charref=true*:

```
&#xF04A;  
&.smileface;
```

Keep in mind that with both methods an alternate representation will be visible instead of the actual symbolic glyphs in the Block properties dialog.

11.5 Filling Blocks with PPS

In order to fill Blocks with PPS you must first place the page containing the Blocks on the output page with the `PDF_fit_pdi_page()` function. After placing the page its Blocks can be filled with the `PDF_fill_*block()` functions.

Simple example: add variable text to a template. Adding dynamic text to a PDF template is a very common task. The following code fragment opens a page in an input PDF document (the template or Block container), places it on the output page, and fills some variable text into a text Block called *firstname*:

```
doc = p.open_pdi_document(filename, "");
if (doc == -1)
    throw new Exception("Error: " + p.get_errmsg());

page = p.open_pdi_page(doc, pageno, "");
if (page == -1)
    throw new Exception("Error: " + p.get_errmsg());

p.begin_page_ext(width, height, "");
/* Place the imported page */
p.fit_pdi_page(page, 0.0, 0.0, "");

/* Fill a single Block on the placed page */
p.fill_textblock(page, "firstname", "Serge", "encoding=winansi");

p.close_pdi_page(page);
p.end_page_ext("");
p.close_pdi_document(doc);
```

Cookbook A full code sample can be found in the *Cookbook* topic `blocks/starter_block`.

Overriding Block properties. In certain situations the programmer wants 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:

- ▶ Business logic may decide to enforce certain overrides.
- ▶ The scaling factor for an image or PDF page will be calculated by the application 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 customer requirements in a print shop application.

Property overrides can be achieved by supplying property names and the corresponding values in the option list of the `PDF_fill_*block()` functions, e.g.

```
p.fill_textblock(page, "firstname", "Serge", "fontsize=12");
```

This will override the Block's internal *fontsize* property with the supplied value 12. Almost all property names can be used as options.

Property overrides apply only to the respective function calls; they will not be stored in the Block definition.

Placing the imported page on top of the filled Blocks. The imported page must have been placed on the output page before using any of the Block filling functions. This means that the original page will usually be placed below the Block contents. However, in some situations it may be desirable to place the original page on top of the filled Blocks. This can be achieved by placing the page once with the *blind* option of *PDF_fit_pdi_page()* in order to make its Blocks and their position known to PPS, and place it again after filling the Blocks in order to actually show the page contents:

```
/* Place the page in blind mode to prepare the Blocks, without the page being visible */
p.fit_pdi_page(page, 0.0, 0.0, "blind");

/* Fill the Blocks */
p.fill_textblock(page, "firstname", "Serge", "encoding=winansi");
/* ... fill more Blocks ... */

/* Place the page again, this time visible */
p.fit_pdi_page(page, 0.0, 0.0, "");
```

Cookbook A full code sample can be found in the *Cookbook topic* `blocks/block_below_contents`.

Ignoring the container page when filling Blocks. Imported Blocks can also be useful as placeholders without any reference to the underlying contents of the Block's page. You can import a container page with Blocks in blind mode on one or more pages, i.e. with the *blind* option of *PDF_fit_pdi_page()*, and subsequently fill its Blocks. This way you can take advantage of the Block and its properties without placing the container page on the output page, and can duplicate Blocks on multiple pages (or even on the same output page).

Cookbook A full code sample can be found in the *Cookbook topic* `blocks/duplicate_block`.

Linking Textflow Blocks. Textflow Blocks can be linked so that one Block holds the overflow text of a previous Block. For example, if you have long variable text which may need to be continued on another page you can link two Blocks and fill the remaining text of the first Block into the second Block.

PPS internally creates a Textflow from the text provided to *PDF_fill_textblock()* and the Block properties. For unlinked Blocks this Textflow will be placed in the Block and the corresponding Textflow handle will be deleted at the end of the call; overflow text will be lost in this case.

With linked Textflow Blocks the overflow text of the first Block can be filled into the next Block. The remainder of the first Textflow will be used as Block contents instead of creating a new Textflow. Linking Textflow Blocks works as follows:

- ▶ In the first call to *PDF_fill_textblock()* within a chain of linked Textflow Blocks the value -1 (in PHP: 0) must be supplied for the *textflowhandle* option. The Textflow handle created internally will be returned by *PDF_fill_textblock()*, and must be stored by the application.
- ▶ In the next call to *PDF_fill_textblock()* the Textflow handle returned in the previous step can be supplied to the *textflowhandle* option (the text supplied in the *text* parameter will be ignored in this case, and should be empty). The Block will be filled with the remainder of the Textflow.
- ▶ This process can be repeated with more Textflow Blocks.

- ▶ The returned Textflow handle can be supplied to `PDF_info_textflow()` in order to determine the results of Block filling, e.g. the end condition or the end position of the text.

Note that the *fitmethod* property should be set to *clip* (this is the default anyway if *textflowhandle* is supplied). The basic code fragment for linking Textflow Blocks looks as follows:

```
p.fit_pdi_page(page, 0.0, 0.0, "");
tf = -1;

for (i = 0; i < blockcount; i++)
{
    String optlist = "encoding=winansi textflowhandle=" + tf;
    tf = p.fill_textblock(page, blocknames[i], text, optlist);
    text = null;

    if (tf == -1)
        break;

    /* check result of most recent call to fit_textflow() */
    reason = (int) p.info_textflow(tf, "returnreason");
    result = p.get_parameter("string", (float) reason);

    /* end loop if all text was placed */
    if (result.equals("_stop"))
    {
        p.delete_textflow(tf);
        break;
    }
}
```

Cookbook A full code sample can be found in the *Cookbook* topic blocks/linked_textblocks.

Block filling order. The Block functions `PDF_fill_*block()` process properties and Block contents in the following order:

- ▶ **Background:** if the *backgroundcolor* property is present and contains a color space keyword different from *None*, the Block area will be filled with the specified color.
- ▶ **Border:** if the *bordercolor* property is present and contains a color space keyword different from *None*, the Block border will be stroked with the specified color and linewidth.
- ▶ **Contents:** the supplied Block contents and all other properties except *bordercolor* and *linewidth* will be processed.
- ▶ **Textline and Textflow Blocks:** if neither text nor default text has been supplied, there won't be any output at all, not even background color or Block border.

Nested Blocks. Before Blocks can be filled the page containing the Blocks must have been placed on the output page before (since otherwise PPS wouldn't know the location of the Blocks after scaling, rotating, and translating the page). If the page only serves as a Block container without bringing static content to the new page you can place the imported page with the *blind* option.

For successful Block filling it doesn't matter how the imported page was placed on the output page:

- ▶ The page can be placed directly with `PDF_fit_pdi_page()`.

- ▶ The page can be placed indirectly in a table cell with *PDF_fit_table()*.
- ▶ The page can be placed as contents of a another PDF Block with *PDF_fill_pdfblock()*.

The third method, i.e. filling a PDF Block with another page containing Blocks, allows nested Block containers. This allows simple implementations of interesting use cases. For example, you can implement both imposition and personalization with a two-step Block filling process:

- ▶ The first-level Block container page contains several large PDF Blocks which indicate the major areas on the paper to be printed on. The arrangement of PDF Blocks reflects the intended postprocessing of the paper (e.g. folding or cutting).
- ▶ Each of the first-level PDF Blocks is then filled with a second-level container PDF page which contains Text, Image, or PDF Blocks to be filled with variable text for personalization.

With this method Block containers can be nested. Although Block nesting works to an arbitrary level, a nesting level of three or more will only rarely be required.

The second-level Block containers may be identical or different for each imposed page. If they are identical all second-level Blocks must be filled before filling the next first-level Block since the information about placing the previous set of second-level Blocks on the page would be no longer available once the next instance of the second-level container page is placed.

Cookbook A full code sample can be found in the *Cookbook topic* `blocks/nested_blocks`.

Block coordinates. The Rectangle coordinates of a Block refer to the PDF default coordinate system. When the page containing the Block is placed on the output page with PPS, several positioning and scaling options can be supplied to *PDF_fit_pdi_page()*. These options 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.

The Block Plugin displays 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. The coordinate display in the Properties dialog is also subject to the units specified in Acrobat (see »Block size and position«, page 277).

Spot colors in Block properties. To use a separation (spot) color in a Block property you can click the »...« button which will present a list of all HKS and PANTONE spot colors. These color names are built into PPS (see Section 3.5.2, »Pantone, HKS, and custom Spot Colors«, page 77) and can be used without further preparations. For custom spot colors an alternate color can be defined in the Block Plugin. If no alternate color is specified in the Block properties, the custom spot color must have been defined earlier in the PPS application using *PDF_makespotcolor()*. Otherwise Block filling will fail.

11.6 Block Properties

PPS and the Block Plugin support general properties which can be assigned to any type of Block. In addition there are properties which are specific to the Block types *Textline*, *Textflow*, *Image*, and *PDF*.

Properties support the same data types as option lists except handles and action lists. The names of Block properties are generally identical to options for *PDF_fit_image()* (e.g., *fitmethod*, *charspacing*). In these cases the behavior is exactly the same as the one documented for the respective option.

11.6.1 Administrative Properties

Administrative properties apply to all Block types. Required entries will automatically be generated by the Block Plugin. Table 11.4 lists the administrative Block properties.

Table 11.4 Administrative properties

keyword	possible values and explanation
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 by PPS.
Locked	(Boolean) If true, the Block and its properties can not be edited with the Block Plugin. This property will be ignored by PPS. Default: false
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. Block names are restricted to a maximum of 125 characters.
Subtype	(Keyword; required) Depending on the Block type, one of Text, Image, or PDF. Note that Textline and Textflow Blocks both have Subtype Text, but are distinguished by the textflow property.
textflow	(Boolean) Controls single- or multiline processing. This property is not available explicitly in the user interface of the Block Plugin, but will be mapped to Textline or Textflow Blocks, respectively (Default: false): false Textline Block: text spans a single line and will be processed with PDF_fit_textline(). true Textflow Block: text can span multiple lines and will be processed with PDF_fit_textflow(). In addition to the standard text properties Textflow-related properties can be specified (see Table 11.9).
Type	(Keyword; required) Always Block

11.6.2 Rectangle Properties

Rectangle properties apply to all Block types. They describe the appearance of the Block rectangle itself. Required entries will automatically be generated by the Block Plugin. Table 11.5 lists the rectangle properties.

Table 11.5 Rectangle properties

keyword	possible values and explanation
background-color	(Color) 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) 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; must be greater than 0) Stroke width of the line used to draw the Block rectangle; only used if bordercolor is set. Default: 1
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 displays the coordinates in Acrobat's notation, i.e., with the origin in the upper left corner of the page. The coordinates will be displayed in the unit which is currently selected in Acrobat, but will always be stored in points in the PDF file.
Status	(Keyword) Describes how the Block will be processed by PPS and the Preview feature (default: active): <div><div>active</div><div>The Block will be fully processed according to its properties.</div><div>ignore</div><div>The Block will be ignored.</div><div>ignoredefault</div><div>Like active, except that the defaulttext/image/pdf properties will be ignored, i.e. the Block remains empty if no variable contents are available (especially in the Preview). This may be useful to make sure that the Block's default contents will not be used for filling Blocks on the server side although the Block may contain default contents for generating Previews. It can also be used to disable the default contents for previewing a Block without removing the default contents from the Block properties.</div><div>static</div><div>No variable contents will be placed; instead, the Block's default text, image, or PDF contents will be used if available.</div></div>

11.6.3 Appearance Properties








Appearance properties specify formatting details:

- Table 11.6 lists transparency appearance properties for all Block types.
- Table 11.7 lists text appearance properties for Textline and Textflow Blocks.

Table 11.6 Transparency appearance properties for all Block types

keyword	possible values and explanation
blendmode	(Keyword list; PDF 1.4; if used in PDF/A mode it must have the value Normal) Name of the blend mode: None, Color, ColorDodge, ColorBurn, Darken, Difference, Exclusion, HardLight, Hue, Lighten, Luminosity, Multiply, None, Normal, Overlay, Saturation, Screen, SoftLight. Default: None
opacityfill	(Float; PDF 1.4; if used in PDF/A mode it must have the value 1) Opacity for fill operations in the range 0..1. The value 0 means fully transparent; 1 means fully opaque.
opacitystroke	(Float; PDF 1.4; if used in PDF/A mode it must have the value 1) Opacity for stroke operations in the range 0..1. The value 0 means fully transparent; 1 means fully opaque.

Table 11.7 Text appearance properties for Textline and Textflow Blocks

keyword	possible values and explanation
charspacing	(Float or percentage) Character spacing. Percentages are based on fontsize. Default: 0
decoration-above	(Boolean) If true, the text decoration enabled with the underline, strikeout, and overline options will be drawn above the text, otherwise below the text. Changing the drawing order affects visibility of the decoration lines. Default: false
fillcolor	(Color) Fill color of the text. Default: gray 0 (=black)
fontname ¹	(String) Name of the font as required by PDF_load_font(). The Block Plugin will present a list of fonts available in the system. However, these font names may not be portable between Mac, Windows, and Unix systems. If fontname starts with an '@' character the font will be applied in vertical writing mode. 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.
fontsize ²	(Float) Size of the font in points
fontstyle	(Keyword) Font style, must be one of normal, bold, italic, or bolditalic
horizscaling	(Float or percentage) Horizontal text scaling. Default: 100%
italicangle	(Float) Italic angle of text in degrees. Default: 0
kerning	(Boolean) Kerning behavior. Default: false
monospace	(Integer: 1...2048) Forces the same width for all characters in the font. Default: absent (metrics from the font will be used)
overline	(Boolean) Overline mode. Default: false
strikeout	(Boolean) Strikeout mode. Default: false
strokecolor	(Color) Stroke color of the text. Default: gray 0 (=black)
strokewidth	(Float, percentage, or keyword; only effective if textrendering is set to outline text) Line width for outline text (in user coordinates or as a percentage of the fontsize). The keyword auto or the value 0 uses a built-in default. Default: auto
textrendering	(Integer) Text rendering mode. Only the value 3 has an effect on Type 3 fonts (default: 0): <div><div><div>0  fill text</div><div>1  stroke text (outline)</div><div>2  fill and stroke text</div><div>3 invisible text</div></div><div><div>4  fill text and add it to the clipping path</div><div>5  stroke text and add it to the clipping path</div><div>6  fill and stroke text and add it to the clipping path</div><div>7  add text to the clipping path (not for Blocks)</div></div></div>
textrise	(Float or percentage) Text rise parameter. Percentages are based on fontsize. Default: 0
underline	(Boolean) Underline mode. Default: false
underline-position	(Float, percentage, or keyword) Position of the stroked line for underlined text relative to the baseline. Percentages are based on fontsize. Default: auto
underline-width	(Float, percentage, or keyword) Line width for underlined text. Percentages are based on fontsize. Default: auto
wordspacing	(Float or percentage) Word spacing. Percentages are based on fontsize. Default: 0

1. This property is required in Textline and Textflow Blocks; it will be enforced by the Block Plugin.

11.6.4 Text Preparation Properties

Text preparation properties specify preprocessing steps for Textline and Textflow Blocks. Table 11.8 lists text preparation properties for Textline and Textflow Blocks.

Table 11.8 Text preparation properties for Textline and Textflow Blocks

keyword	possible values and explanation
charref	(Boolean) If true, enable substitution of numeric and character entity references and glyph name references. Default: the global charref parameter
escape-sequence	(Boolean) If true, enable substitution of escape sequences in content strings, hypertext strings, and name strings. Default: the global escapesequences parameter
features	<p>(List of keywords) Specifies which typographic features of an OpenType font will be applied to the text, subject to the script and language options. Keywords for features which are not present in the font will silently be ignored. The following keywords can be supplied:</p> <p>_none Apply none of the features in the font. As an exception, the vert feature must explicitly be disabled with the novert keyword.</p> <p><name> Enable a feature by supplying its four-character OpenType tag name. Some common feature names are liga, ital, tnum, smcp, swsh, zero. The full list with the names and descriptions of all supported features can be found in Section 6.3.1, »Supported OpenType Layout Features«, page 152.</p> <p>no<name> The prefix no in front of a feature name (e.g. noliga) disables this feature.</p> <p>Default: _none for horizontal writing mode. In vertical writing mode vert will automatically be applied. The readfeatures option in PDF_load_font() is required for OpenType feature support.</p>
language	(Keyword; only relevant if script is supplied) The text will be processed according to the specified language, which is relevant for the features and shaping options. A full list of keywords can be found in Section 6.4.2, »Script and Language«, page 160, e.g. ARA (Arabic), JAN (Japanese), HIN (Hindi). Default: _none (undefined language)
script	(Keyword; required if shaping=true) The text will be processed according to the specified script, which is relevant for the features, shaping, and advancedlinebreaking options. The most common keywords for scripts are the following: _none (undefined script), latn, grek, cyrl, armn, hebr, arab, deva, beng, guru, gujr, orya, taml, thai, lao, tib, hang, kana, han. The keyword _auto selects the script to which the majority of characters in the text belong, where _latn and _none are ignored. A full list of keywords can be found in Section 6.4.2, »Script and Language«, page 160. Default: _none
shaping	(Boolean) If true, the text will be formatted (shaped) according to the script and language options. The script option must have a value different from _none and the required shaping tables must be available in the font. Default: false

11.6.5 Text Formatting Properties

Table 11.9 lists properties which can only be used for Textflow Blocks, with the exception of the *stamp* property which can also be used for Textline Blocks. They 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 for Textflows 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()*, or in the Block's *defaulttext* property.

Table 11.9 Text formatting properties (mostly for Textflow Blocks)

keyword	possible values and explanation
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 <i>minspacing</i> and <i>maxspacing</i> options (default: <i>auto</i>): auto The following methods are applied in order: <i>shrink</i> , <i>spread</i> , <i>nofit</i> , <i>split</i> . clip Same as <i>nofit</i> , except that the long part at the right edge of the fit box (taking into account the <i>rightindent</i> 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 <i>nofitlimit</i> 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 <i>shrinklimit</i> . If it still doesn't fit the <i>nofit</i> 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 <i>spreadlimit</i> . If justification still cannot be achieved the <i>nofit</i> method will be applied.
advanced-linebreak	(Boolean) Enable the advanced line breaking algorithm which is required for complex scripts. This is required for linebreaking in scripts which do not use space characters for designating word boundaries, e.g. Thai. The options <i>locale</i> and <i>script</i> will be honored. Default: <i>false</i>
alignment	(Keyword) Specifies formatting for lines in a paragraph. Default: <i>left</i> . left left-aligned, starting at <i>leftindent</i> center centered between <i>leftindent</i> and <i>rightindent</i> right right-aligned, ending at <i>rightindent</i> justify left- and right-aligned
avoid-emptybegin	(Boolean) If <i>true</i> , empty lines at the beginning of a fitbox will be deleted. Default: <i>false</i>
fixedleading	(Boolean) If <i>true</i> , 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: <i>false</i>
hortab-method	(Keyword) Treatment of horizontal tabs in the text. If the calculated position is to the left of the current text position, the tab will be ignored (default: <i>relative</i>): relative The position will be advanced by the amount specified in <i>hortabsize</i> . typewriter The position will be advanced to the next multiple of <i>hortabsize</i> . ruler The position will be advanced to the <i>n</i> -th tab value in the <i>ruler</i> option, where <i>n</i> is the number of tabs found in the line so far. If <i>n</i> is larger than the number of tab positions the <i>relative</i> method will be applied.
hortabsize	(Float or percentage) Width of a horizontal tab ¹ . The interpretation depends on the <i>hortabmethod</i> option. Default: 7.5%

Table 11.9 Text formatting properties (mostly for Textflow Blocks)

keyword	possible values and explanation
lastalignment	(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.
leading	(Float or percentage) Distance between adjacent text baselines in user coordinates, or as a percentage of the font size. Default: 100%
locale	(Keyword) The locale which will be used for localized linebreaking methods if advancedlinebreak=true. The keywords consists of one or more components, where the optional components are separated by an underscore character '_' (the syntax slightly differs from NLS/POSIX locale IDs): <ul style="list-style-type: none">▶ A required two- or three-letter lowercase language code according to ISO 639-2 (see www.loc.gov/standards/iso639-2), e.g. en, (English), de (German), ja (Japanese). This differs from the language option.▶ An optional four-letter script code according to ISO 15924 (see www.unicode.org/iso15924/iso15924-codes.html), e.g. Hira (Hiragana), Hebr (Hebrew), Arab (Arabic), Thai (Thai).▶ An optional two-letter uppercase country code according to ISO 3166 (see www.iso.org/iso/country_codes/iso_3166_code_lists), e.g. DE (Germany), CH (Switzerland), GB (United Kingdom) Specifying a locale is not required for advanced line breaking: the keyword _none specifies that no locale-specific processing will be done. Default: _none Examples: de_DE, en_US, en_GB
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%
minlinecount	(Integer) Minimum number of lines in the last paragraph of the fitbox. If there are fewer lines they will be placed in the next fitbox. The value 2 can be used to prevent single lines of a paragraph at the end of a fitbox («orphans»). Default: 1
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%
parindent	(Float or percentage) Left indent of the first line of a paragraph ¹ . The amount will be added to leftindent. Specifying this option within a line will act like a tab. Default: 0
rightindent leftindent	(Float or percentage) Right or left indent of all text lines ¹ . If leftindent 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
ruler²	(List of floats or percentages) List of absolute tab positions for hortabmethod=ruler ¹ . The list may contain up to 32 non-negative entries in ascending order. Default: integer multiples of hortabsize
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 horizscaling 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
stamp	(Keyword; Textline and Textflow Blocks) This option can be used to create a diagonal stamp within the Block rectangle. The text comprising the stamp will be as large as possible. The options position, fitmethod, and orientate (only north and south) will be honored when placing the stamp text in the box. Default: none. ll2ur The stamp will run diagonally from the lower left corner to the upper right corner. ul2lr The stamp will run diagonally from the upper left corner to the lower right corner. none No stamp will be created.

Table 11.9 Text formatting properties (mostly for Textflow Blocks)

keyword	possible values and explanation
tabalignchar	(Unichar) Unicode value of the character at which decimal tabs will be aligned. Default: the period character '.' (U+002E)
tabalignment²	(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 tabalignchar will be left-aligned at the tab position. If no tabalignchar 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.

1. In user coordinates, or as a percentage of the width of the fit box
2. Tab settings can be edited in the property subgroup Ruler Tabs for hortabmethod=ruler in the Block properties dialog.

11.6.6 Object Fitting Properties

Fitting properties are available for all Block types, although some properties are specific to a certain Block type. They manage how the contents will be placed in the Block:

- ▶ Table 11.10 lists fitting properties for Textline, Image, and PDF Blocks
- ▶ Table 11.11 lists fitting properties for Textflow Blocks (mostly related to aspects of vertical fitting).

The object fitting algorithm uses the Block rectangle as fitbox. Except for *fitmethod=clip* there will be no clipping; if you want to make sure that the Block contents do not exceed the Block rectangle avoid *fitmethod=nofit*.

Table 11.10 Fitting properties for Textline, Image, and PDF Blocks

keyword	possible values and explanation
alignchar	(Unichar or keyword; only for Textline Blocks) If the specified character is found in the text, its lower left corner will be aligned at the lower left corner of the Block rectangle. For horizontal text with orientate=north or south the first value supplied in the position option defines the position. For horizontal text with orientate=west or east the second value supplied in the position option defines the position. This option will be ignored if the specified alignment character is not present in the text. The value o and the keyword none suppress alignment characters. The specified fitmethod will be applied, although the text cannot be placed within the Block rectangle because of the forced positioning of alignchar. Default: none
dpi	(Float list; only for image Blocks) One or two values specifying the desired image resolution in pixels per inch in horizontal and vertical direction. With the value o 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: o
fitmethod	(Keyword) Strategy to use if the supplied content doesn’t fit into the Block rectangle. Possible values are auto, nofit, clip, meet, slice, and entire. For Textline Blocks, image, and PDF Blocks this property will be interpreted according to the standard interpretation. 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.
margin	(Float list; only for Textline Blocks) One or two float values describing additional horizontal and vertical reduction of the Block rectangle. Default: o
orientate	(Keyword) Specifies the desired orientation of the content when it is placed. Possible values are north, east, south, west. Default: north
position	(Float list) One or two values specifying the position of the reference point within the content. The position is specified as a percentage within the Block. Only for Textline Blocks: the keyword auto can be used for the first value in the list. It indicates right if the writing direction of the text is from right to left (e.g. for Arabic and Hebrew text), and left otherwise (e.g. for Latin text). Default: {o o}, i.e. the lower left corner
rotate	(Float) 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: o
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
shrinklimit	(Float or percentage; only for Textline Blocks) The lower limit of the shrinkage factor which will be applied to fit text with fitmethod=auto. Default: 0.75

Table 11.11 Fitting properties for Textflow Blocks

keyword	possible values and explanation
firstlinedist	<p>(Float, percentage, or keyword) The distance between the top of the Block rectangle and the baseline for the first line of text, specified in user coordinates, as a percentage of the relevant font size (the first font size in the line if <code>fixedleading=true</code>, and the maximum of all font sizes in the line otherwise), or as a keyword. Default: <code>leading</code>.</p> <p>leading The leading value determined for the first line; typical diacritical characters such as À will touch the top of the fitbox.</p> <p>ascender The ascender value determined for the first line; typical characters with larger ascenders, such as d and h will touch the top of the fitbox.</p> <p>capheight The capheight value determined for the first line; typical capital uppercase characters such as H will touch the top of the fitbox.</p> <p>xheight The xheight value determined for the first line; typical lowercase characters such as x will touch the top of the fitbox.</p> <p>If <code>fixedleading=false</code> the maximum of all <code>leading</code>, <code>ascender</code>, <code>xheight</code>, or <code>capheight</code> values found in the first line will be used.</p>
fitmethod	<p>(Keyword) Strategy to use if the supplied content doesn't fit into the box. Possible values are <code>auto</code>, <code>nofit</code>, <code>clip</code>. Default: <code>auto</code>. For Textflow Blocks where the Block is too small for the text the interpretation is as follows:</p> <p>auto fontsize and leading will be decreased until the text fits.</p> <p>nofit Text will run beyond the bottom margin of the Block.</p> <p>clip Text will be clipped at the Block margin.</p>
lastlinedist	<p>(Float, percentage, or keyword) Will be ignored for <code>fitmethod=nofit</code>) The minimum distance between the baseline for the last line of text and the bottom of the fitbox, specified in user coordinates, as a percentage of the font size (the first font size in the line if <code>fixedleading= true</code>, and the maximum of all font sizes in the line otherwise), or as a keyword. Default: <code>o</code>, i.e. the bottom of the fitbox will be used as baseline, and typical descenders will extend below the Block rectangle.</p> <p>descender The descender value determined for the last line; typical characters with descenders, such as g and j will touch the bottom of the fitbox.</p> <p>If <code>fixedleading=false</code> the maximum of all descender values found in the last line will be used.</p>
linespread-limit	<p>(Float or percentage; only for <code>verticalalign=justify</code>) Maximum amount in user coordinates or as percentage of the leading for increasing the leading for vertical justification. Default: <code>200%</code></p>
maxlines	<p>(Integer or keyword) The maximum number of lines in the fitbox, or the keyword <code>auto</code> which means that as many lines as possible will be placed in the fitbox. When the maximum number of lines has been placed <code>PDF_fit_textflow()</code> will return the string <code>_boxfull</code>.</p>
minfontsize	<p>(Float or percentage) Minimum allowed font size when text is scaled down to fit into the Block rectangle with <code>fitmethod=auto</code> when <code>shrinklimit</code> is exceeded. The limit is specified in user coordinates or as a percentage of the height of the Block. If the limit is reached the text will be created with the specified <code>minfontsize</code> as <code>fontsize</code>. Default: <code>0.1%</code></p>
orientate	<p>(Keyword) Specifies the desired orientation of the text when it is placed. Possible values are <code>north</code>, <code>east</code>, <code>south</code>, <code>west</code>. Default: <code>north</code></p>
rotate	<p>(Float) Rotate the coordinate system, using the lower left corner of the fitbox 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: <code>0</code></p>

Table 11.11 Fitting properties for Textflow Blocks

keyword	possible values and explanation
verticalalign	(Keyword) Vertical alignment of the text in the fitbox. Default: top.
top	Formatting will start at the first line, and continue downwards. If the text doesn't fill the fitbox there may be whitespace below the text.
center	The text will be vertically centered in the fitbox. If the text doesn't fill the fitbox there may be whitespace both above and below the text.
bottom	Formatting will start at the last line, and continue upwards. If the text doesn't fill the fitbox there may be whitespace above the text.
justify	The text will be aligned with top and bottom of the fitbox. In order to achieve this the leading will be increased up to the limit specified by linespreadlimit. The height of the first line will only be increased if firstlinedist=leading.

11.6.7 Properties for default Contents

Properties for default contents specify how to fill the Block if no specific contents are provided. They are especially useful for the Preview feature since it will fill the Blocks with their default contents. Table 11.12 lists properties for default contents.

Table 11.12 Properties for default contents

keyword	possible values and explanation
defaultimage	(String; only for image Blocks) Path name of an image which will be used if no image is supplied by the client application. It is recommended to use file names without absolute paths, and use the SearchPath feature in the PPS client application. This makes Block processing independent from platform and file system details.
defaultpdf	(String; only for PDF Blocks) Path name of a PDF document which will be used if no substitution PDF is supplied by the client application. It is recommended to use file names without absolute paths, and use the SearchPath feature in the PPS client application. This makes Block processing independent from platform and file system details.
default-pdfpage	(Integer; only for PDF Blocks) Page number of the page in the default PDF document. Default: 1
defaulttext	(String; only for Textline and Textflow Blocks) Text which will be used if no variable text is supplied by the client application ¹

1. The text will be interpreted in winansi encoding or Unicode.

11.6.8 Custom Properties

Custom properties apply to Blocks of any type of Block, and will be ignored by PPS and the Preview feature. Table 11.13 lists the naming rules for custom properties.

Table 11.13 Custom Block properties for all Block types

keyword	possible values and explanation
any name not containing the three characters [] /	(String, name, float, or float list) The interpretation of the values of custom properties is completely up to the client application; they will be ignored by PPS.

11.7 Querying Block Names and Properties with pCOS

In addition to automatic Block processing with PPS, the integrated pCOS facility can be used to enumerate Block names and query standard or custom properties.

Cookbook A full code sample for querying the properties of Blocks contained in an imported PDF can be found in the Cookbook topic `blocks/query_block_properties`.

Finding the number and names of Blocks. The client code must not even know the names or number of Blocks on an imported page since these can also be queried. The following statement returns the number of Blocks on page with number *pagenum*:

```
blockcount = (int) p.pcos_get_number(doc, "length:pages[" + pagenum + "]/blocks");
```

The following statement returns the name of Block number *blocknum* on page *pagenum* (Block and page counting start at 0):

```
blockname = p.pcos_get_string(doc,
    "pages[" + pagenum + "]/blocks[" + blocknum + "]/Name");
```

The returned Block name can subsequently be used to query the Block's properties or fill the Block with text, image, or PDF content. If the specified Block doesn't exist an exception will be thrown. You can avoid this by using the *length* prefix to determine the number of Blocks and therefore the maximum index in the *blocks* array (keep in mind that the Block count will be one higher than the highest possible index since array indexing starts at 0).

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>*:

```
pages[...]/blocks[<number>]
pages[...]/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 = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/Rect[0]");
lly = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/Rect[1]");
urx = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/Rect[2]");
ury = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/Rect[3]");
```

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. Since the *Rect* option for overriding Block coordinates does not take into account any modifications applied by the CropBox entry, the coordinates queried from the original Block cannot be directly used as new coordinates if a CropBox is present. As a workaround you can use the *refpoint* and *boxsize* options.

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* on page *pagenum*:

```
zip = p.pcos_get_string(doc, "pages[" + pagenum + "]/blocks/b1/Custom/zipcode");
```

If you don't know which custom properties are actually present in a Block, you can determine the names at runtime. In order to find the name of the first custom property in a Block named *b1* use the following:

```
propname = p.pcos_get_string(doc, "pages[" + pagenum + "]/blocks/b1/Custom[0].key");
```

Use increasing indexes instead of 0 in order to determine the names of all custom properties. Use the *length* prefix to determine the number of custom properties.

Non-existing Block properties and default values. Use the *type* prefix to determine whether a Block or property is actually present. If the type for a path is 0 or *null* the respective object is not present in the PDF document. Note that for standard properties this means that the default value of the property will be used.

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 PPS property names (as defined in Section 11.6, »Block Properties«, page 294) will never conflict with custom property names.

11.8 PDFlib Block Specification

The Block syntax fully conforms to 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 Blocks by other means than the Block Plugin. Plugin users can safely skip this section.

11.8.1 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 11.14.

Table 11.14 Entries in a PDFlib application data dictionary

key	value
LastModified	(Data string; required) The date and time when the Blocks on the page were created or most recently modified.
Private	(Dictionary; required) A Block list (see Table 11.15)

A Block list is a dictionary containing general information about Block processing, plus a list of all Blocks on the page. Table 11.15 lists the keys in a Block list dictionary.

Table 11.15 Entries in a Block list dictionary

key	value
Version	(Number; required) The version number of the Block specification to which the file complies. This document describes version 9 of the Block specification.
Blocks	(Dictionary; required) Each key is a name object containing the name of a Block; the corresponding value is the Block dictionary for this Block (see Table 11.17). The value of the Name key in the Block dictionary must be identical to the Block's name in this dictionary.
PluginVersion	(String; required unless the pdfmark key is present ¹) A string containing a version identification of the Block Plugin which has been used to create the Blocks.
pdfmark	(Boolean; required unless the PluginVersion key is present ¹) Must be true if the Block list has been generated by use of pdfmarks.

1. Exactly one of the PluginVersion and pdfmark keys must be present.

Data types for Block properties. Properties support the same data types as option lists except handles and specialized lists such as action lists. Table 11.16 details how these types are mapped to PDF data types.

Table 11.16 Data types for Block properties

Data type	PDF type and remarks
boolean	(Boolean)
string	(String)
keyword (name)	(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.
percentage	(Array with two elements) The first element in the array is the number, the second element is a string containing a percent character.
list	(Array)
color	<p>(Array with two or three elements) The first element in the array specifies a color space, and the second element specifies a color value as follows. The following entries are supported for the first element in the array:</p> <p>/DeviceGray The second element is a single gray value.</p> <p>/DeviceRGB The second element is an array of three RGB values.</p> <p>/DeviceCMYK The second element is an array of four CMYK values.</p> <p>[/Separation/spotname] The first element is an array containing the keyword /Separation and a spot color name. The second element is a tint value. The optional third element in the array specifies an alternate color for the spot color, which is itself a color array in one of the /DeviceGray, /DeviceRGB, /DeviceCMYK, or /Lab color spaces. If the alternate color is missing, the spot color name must either refer to a color which is known internally to PPS, or which has been defined by the application at runtime.</p> <p>[/Lab] The first element is an array containing the keyword /Lab. 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>
unichar	(Text string) Unicode string in utf16be format, starting with the BOM U+FEFF

11.8.2 Block Dictionary Keys

Block dictionaries may contain the keys in Table 11.17.

Table 11.17 Entries in a Block dictionary

<i>property group</i>	<i>values</i>
<i>administrative properties</i>	<i>(Some keys are required) Administrative properties according to Table 11.4</i>
<i>rectangle properties</i>	<i>(Some keys are required) Rectangle properties according to Table 11.5</i>
<i>appearance properties</i>	<i>(Some keys are required) Appearance properties for all Block types according to Table 11.6 and text appearance properties according to Table 11.7 for Textline and Textline Blocks</i>
<i>text preparation properties</i>	<i>(Optional) Text preparation properties for Textline and Textflow Blocks according to Table 11.8</i>
<i>text formatting properties</i>	<i>(Optional) Text formatting properties for Textline and Textflow Blocks according to Table 11.9</i>
<i>object fitting properties</i>	<i>(Optional) Object fitting properties for Textline, Image, and PDF Blocks according to Table 11.10, and fitting properties for Textflow Blocks according to Table 11.11</i>
<i>properties for default contents</i>	<i>(Optional) Properties for default contents according to Table 11.12</i>
<i>Custom</i>	<i>(Dictionary; optional) A dictionary containing key/value pairs for custom properties according to Table 11.13.</i>

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 9
    /pdfmark true
  >>
  /LastModified (D:20120813200730)
>>
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
>>

```

11.8.3 Generating PDFlib Blocks with pdfmarks

As an alternative to creating 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:20120813200730)
            /Private <<
                /Version 9
                /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
```


A Revision History

Date	Changes
October 23, 2012	► Various updates and corrections for PDFlib 8.0.5
December 23, 2011	► Various updates and corrections for PDFlib 8.0.4
July 11, 2011	► Various updates and corrections for PDFlib 8.0.3
December 09, 2010	► Various updates and corrections for PDFlib 8.0.2
September 22, 2010	► Various updates and corrections for PDFlib 8.0.1p7
April 13, 2010	► Various updates and corrections for PDFlib 8.0.1
December 07, 2009	► Updates for PDFlib 8.0.0
April 20, 2010	► Minor corrections for PDFlib 7.0.5
March 13, 2009	► Various updates and corrections for PDFlib 7.0.4
February 13, 2008	► Various updates and corrections for PDFlib 7.0.3
August 08, 2007	► Various updates and corrections for PDFlib 7.0.2
February 19, 2007	► Various updates and corrections for PDFlib 7.0.1
October 03, 2006	► Updates and restructuring for PDFlib 7.0.0
February 15, 2007	► Various updates and corrections for PDFlib 6.0.4
February 21, 2006	► Various updates and corrections for PDFlib 6.0.3; added Ruby section
August 09, 2005	► Various updates and corrections for PDFlib 6.0.2
November 17, 2004	► Minor updates and corrections for PDFlib 6.0.1 ► introduced new format for language-specific function prototypes in chapter 8 ► added hypertext examples in chapter 3
June 18, 2004	► Major changes for PDFlib 6
January 21, 2004	► Minor additions and corrections for PDFlib 5.0.3
September 15, 2003	► Minor additions and corrections for PDFlib 5.0.2; added block specification
May 26, 2003	► Minor updates and corrections for PDFlib 5.0.1
March 26, 2003	► Major changes and rewrite for PDFlib 5.0.0
June 14, 2002	► Minor changes for PDFlib 4.0.3 and extensions for the .NET binding
January 26, 2002	► Minor changes for PDFlib 4.0.2 and extensions for the IBM eServer edition
May 17, 2001	► Minor changes for PDFlib 4.0.1
April 1, 2001	► Documents PDI and other features of PDFlib 4.0.0
February 5, 2001	► Documents the template and CMYK features in PDFlib 3.5.0
December 22, 2000	► ColdFusion documentation and additions for PDFlib 3.0.3; separate COM edition of the manual
August 8, 2000	► Delphi documentation and minor additions for PDFlib 3.0.2
July 1, 2000	► Additions and clarifications for PDFlib 3.0.1
Feb. 20, 2000	► Changes for PDFlib 3.0

<i>Date</i>	<i>Changes</i>
<i>Aug. 2, 1999</i>	▶ <i>Minor changes and additions for PDFlib 2.01</i>
<i>June 29, 1999</i>	▶ <i>Separate sections for the individual language bindings</i> ▶ <i>Extensions for PDFlib 2.0</i>
<i>Feb. 1, 1999</i>	▶ <i>Minor changes for PDFlib 1.0 (not publicly released)</i>
<i>Aug. 10, 1998</i>	▶ <i>Extensions for PDFlib 0.7 (only for a single customer)</i>
<i>July 8, 1998</i>	▶ <i>First attempt at describing PDFlib scripting support in PDFlib 0.6</i>
<i>Feb. 25, 1998</i>	▶ <i>Slightly expanded the manual to cover PDFlib 0.5</i>
<i>Sept. 22, 1997</i>	▶ <i>First public release of PDFlib 0.4 and this manual</i>

Index

A

- Acrobat plugin for creating Blocks 271
- Adobe Font Metrics (AFM) 112
- advanced linebreaking 216
- AES (Advanced Encryption Standard) 73
- AFM (Adobe Font Metrics) 112
- ArtBox 67
- artificial font styles 149
- AS/400 62
- ascender 147
- asciifile parameter 63
- auto: see *hypertextformat*
- autocidfont parameter 138
- autosubsetting parameter 138

B

- backslash substitution 107
- baseline compression 175
- Basic Multilingual Plane 93
- Big Five 101
- bindings 27
- BleedBox 67
- Blocks 271
 - plugin 271
 - properties 273
- BMP 93, 177
- Byte Order Mark (BOM) 94, 105
- bytes: see *hypertextformat*
- byteserving 75

C

- C binding 30
- C++ binding 33
- capheight 147
- categories of resources 57
- CCITT 177
- CCSID 96
- character metrics 147
- character references 107, 108
- characters and glyphs 93
- characters per inch 148
- Chinese 100, 101, 166
- CIE L*a*b* color space 80
- CJK (Chinese, Japanese, Korean)
 - configuration 99
 - custom fonts 168
 - standard fonts 99
 - Windows code pages 101
- clip 67

- clone page boxes 191
- CMaps 99, 100
- Cobol binding 28
- code page: Microsoft Windows 1250-1258 95
- COM (Component Object Model) binding 29
- commercial license 12
- content strings 102
- content strings in non-Unicode capable languages 104
- coordinate system 64
 - metric 64
 - top-down 66
- copyoutputintent option 253
- core fonts 129
- CPI (characters per inch) 148
- CropBox 67
- current point 68
- currentx and currenty parameter 147
- custom encoding 97

D

- default coordinate system 64
- defaultgray/rgb/cmyk color space 82
- descender 147
- downsampling 173
- dpi calculations 173

E

- EBCDIC 62
- ebcdic encoding 95
- ebcdicutf8: see *hypertextformat*
- embedding fonts 137
- encoding
 - CJK 99
 - custom 97
 - fetching from the system 96
- encryption 72
- environment variable PDFLIBRESOURCE 60
- error handling 53
- errorpolicy parameter 184
- escape sequences 107
- EUDC (end-user defined characters) 111, 169
- evaluation version 9
- exceptions 53
- EXIF JPEG images 176
- explicit transparency 179

F

- features of PDFlib 21, 24

- file search* 58
- fill* 67
- font metrics* 147
- font style names for Windows* 132
- font styles* 149
- fonts*
 - AFM files* 112
 - embedding* 137
 - legal aspects of embedding* 138
 - monospaced* 148
 - OpenType* 111
 - PDF core set* 129
 - PFA files* 112
 - PFB files* 112
 - PFM files* 112
 - PostScript Type 1* 112
 - resource configuration* 56
 - subsetting* 138
 - TrueType* 111
 - Type 3 (user-defined) fonts* 113
- form fields: converting to blocks* 282
- form XObjects* 69

G

- gaiji characters* 113
- GBK* 101
- GIF* 177
- glyph availability* 142
- glyph id addressing* 115
- glyph replacement* 119
- glyphlets* 112
- glyphs* 93
- gradients* 76
- grid.pdf* 65
- Groovy* 38

H

- HKS colors* 79
- horizontal writing mode* 167
- host encoding* 96
- host fonts* 131
- HTML character references* 107
- hypertext strings* 102
 - in non-Unicode capable languages* 104
- hypertextformat parameter* 104

I

- IBM zSeries and iSeries* 62
- ignoremask* 179
- image data, re-using* 173
- image file formats* 175
- image mask* 178, 180
- image scaling* 173
- image:iccprofile parameter* 81
- inch* 64
- in-core PDF generation* 61

- inline images* 174
- invisible text* 297
- iSeries* 62
- ISO 10646* 121
- ISO 15930* 247
- ISO 19005* 254
- ISO 32000* 246
- ISO 8859-2 to -15* 95

J

- Japanese* 100, 101, 166
- Java binding* 36
- Javadoc* 38
- JBIG2* 176
- JFIF* 175
- Johab* 101
- JPEG* 175
- JPEG 2000* 176
- JPEG images in EXIF format* 176

K

- Kerning* 148
- Korean* 100, 101, 166

L

- language bindings: see bindings*
- layers and PDI* 184
- leading* 147
- line spacing* 147
- linearized PDF* 75
- LWFN (LaserWriter Font)* 112

M

- macroman encoding* 95
- makespres utility* 56
- masked* 179
- masking images* 178
- masterpassword* 73
- MediaBox* 67
- memory, generating PDF documents in* 61
- metric coordinates* 64
- metrics* 147
- millimeters* 64
- monospaced fonts* 148
- multi-page image files* 174

N

- name strings* 102
 - in non-Unicode capable languages* 104
- nesting exceptions* 31
- .NET binding* 39

O

- Objective-C binding* 40

- OpenType fonts 111
- optimized PDF 75
- outline text 297
- output intent
 - for PDF/A 255
 - for PDF/X 248
- output intents
 - for PDF/A 257
 - for PDF/X 251
- overline parameter 150

P

- page 174
- page descriptions 64
- page formats 66
- page size
 - limitations in Acrobat 67
- page-at-a-time download 75
- PANTONE colors 77
- passwords 72
- path 67
- path objects 68
- patterns 76
- pCOS 241
- pCOS interface 241
- PDF import library (PDI) 182
- PDF/A 254
- PDF/X 247
- PDF_EXIT_TRY() 32
- PDF_get_buffer() 61
- PDFlib Blocks 271
- PDFlib features 21, 24
- PDFlib Personalization Server (PPS) 271
- pdflib.upr 60
- PDFLIBRESOURCE environment variable 60
- PDI (PDF Import) 182
- pdiusebox 184
- Perl binding 42
- permissions 72, 73
- PFA (Printer Font ASCII) 112
- PFB (Printer Font Binary) 112
- PFM (Printer Font Metrics) 112
- PHP binding 44
- plugin for creating Blocks 271
- PNG 175, 179
- PostScript Type 1 fonts 112
- PPS (PDFlib Personalization Server) 271
- Printer Font ASCII (PFA) 112
- Printer Font Binary (PFB) 112
- Printer Font Metrics (PFM) 112
- Private Use Area 93
- PUA 93
- Python binding 46

R

- raw image data 177
- REALbasic binding 47

- rendering intents 80
- renderingintent option 80
- resource category 57
- resourcefile parameter 60
- rotating objects 65
- RPG binding 48
- Ruby binding 50

S

- S/390 62
- scaling images 173
- script-specific linebreaking 216
- SearchPath parameter 58
- security 72
- setcolor:iccprofilegray/rgb/cmyk parameters 82
- shadings 76
- Shift-JIS 101
- SING fonts 112
- smooth blends 76
- spot color (separation color space) 77
- sRGB color space 81
- standard output conditions for PDF/X 251
- strikeout parameter 150
- strings in option lists 105
- stroke 67
- style names for Windows 132
- subpath 67
- subscript 148
- subsetminsize parameter 138
- subsetting 138
- superscript 148
- system encoding support 96

T

- Tcl binding 52
- templates 69
- temporary disk space requirements 75
- text metrics 147
- text position 147
- text variations 147
- textformat parameter 104
- textrendering parameter 150
- textx and texty parameter 147
- TIFF 177
- top-down coordinates 66
- transparency 178
- TrimBox 67
- TrueType fonts 111
- TTC (TrueType Collection) 111, 168
- Type 1 fonts 112
- Type 3 (user-defined) fonts 113

U

- UHC 101
- underline parameter 150
- units 64

UPR (Unix PostScript Resource) 56
usehypertextencoding parameter 104
user space 64
usercoordinates parameter 64
user-defined (Type 3) fonts 113
userpassword 73
UTF formats 94
utf16: see hypertextformat
utf16be: see hypertextformat
utf16le: see hypertextformat
utf8: see hypertextformat

V

vertical writing mode 167

W

web-optimized PDF 75
winansi encoding 95
writing modes 167

X

xheight 147
XObjects 69

Z

zSeries 62

PDFlib GmbH

Franziska-Bilek-Weg 9
80339 München, Germany
www.pdflib.com
phone +49 • 89 • 452 33 84-0
fax +49 • 89 • 452 33 84-99

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

Licensing contact

sales@pdflib.com

Support

support@pdflib.com (*please include your license number*)

