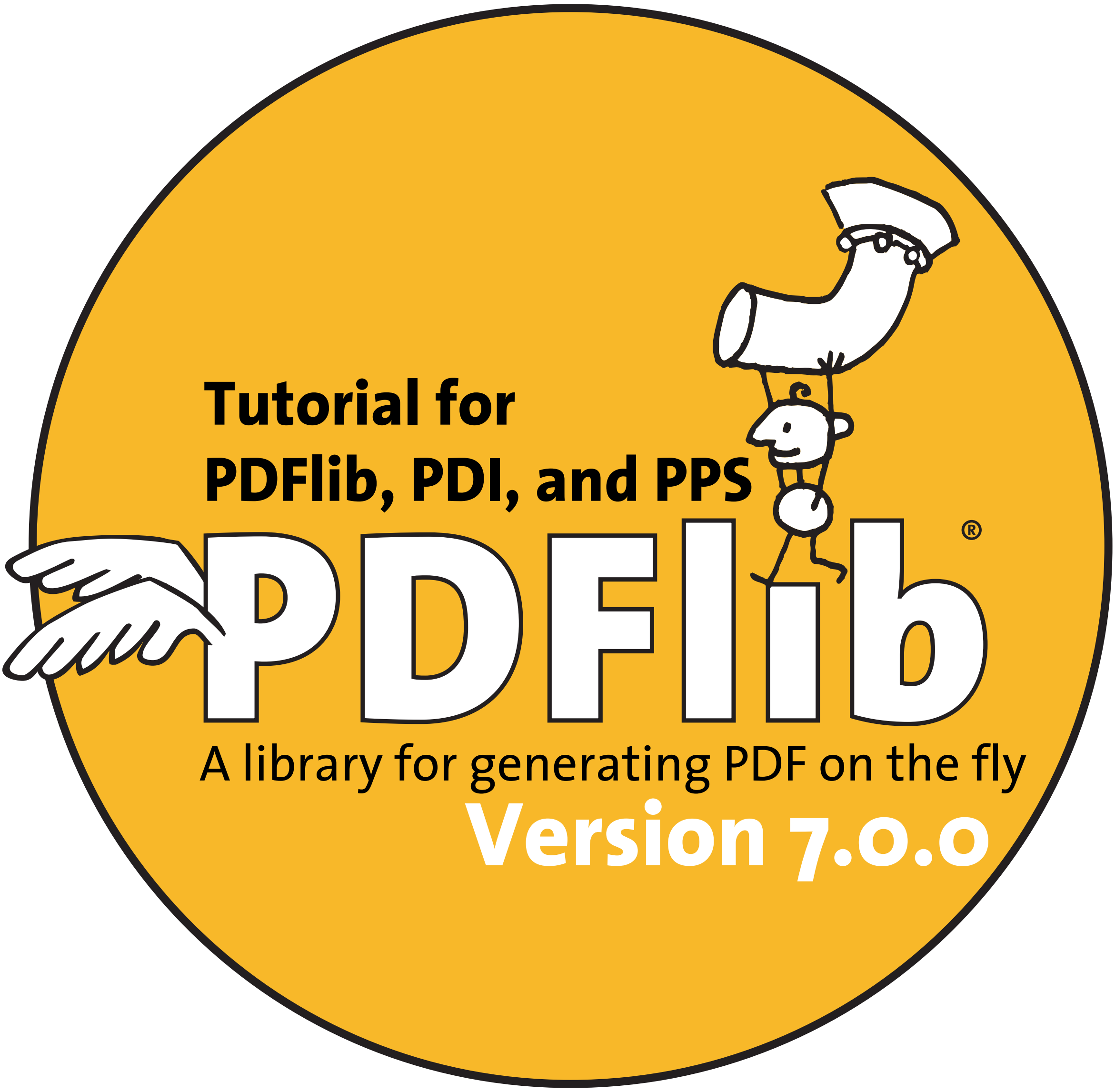
PDFlib GmbH München, Germany

www.pdflib.com



Tutorial for PDFlib, PDI, and PPS

PDFlib®

A library for generating PDF on the fly

Version 7.0.0

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

PDFlib GmbH
Tal 40, 80331 München, Germany
www.pdflib.com
phone +49 • 89 • 29 16 46 87
fax +49 • 89 • 29 16 46 86

If you have questions check the PDFlib mailing list and archive at 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.

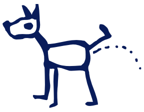
PDFlib and the PDFlib logo are registered trademarks of PDFlib GmbH. PDFlib licensees are granted the right to use the PDFlib name and logo in their product documentation. However, this is not required.

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 (eay@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

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



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

Contents

0 Applying the PDFlib License Key

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

Once you purchased a license key you must apply it in order to get rid of the demo stamp. You can apply the license key with a PDFlib call at runtime, by preparing a license file, or (on Windows) using a registry key. If you are working with the Windows installer you can enter a license key when you install the product.

Applying a license key 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_parameter(p, "license", "...your license key...")
```

- In C++, Java, Ruby, and PHP 5 with the object-oriented interface:

```
p.set_parameter("license", "...your license key...")
```

- In Perl, PHP 4 and PHP 5 with the function-based interface:

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

- In RPG:

```
d licensekey      s             20
d licenseval      s             50
c                   eval      licenseopt='license'+x'00'
c                   eval      licenseval='...your license key...'+x'00'
c                   callp     PDF_set_parameter(p:licenseopt:licenseval)
```

- In Tcl:

```
PDF_set_parameter $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):

```
PDFlib license file 1.0
# Licensing information for PDFlib GmbH products
PDFlib      7.0.0   ...your license key...
```

The license file may contain license keys for multiple PDFlib GmbH products on separate lines. Next, you must inform PDFlib about the license file, either by setting the *licensefile* parameter immediately after instantiating the PDFlib object (i.e., after *PDF_new()* or equivalent call) with a function call similar to the following:

- In C and Python:

```
PDF_set_parameter(p, "licensefile", "/path/to/licensekeys.txt")
```

- In C++, Java, and PHP 5 with the object-oriented interface:

```
p.set_parameter("licensefile", "/path/to/licensekeys.txt");
```

- In Perl, PHP 4 and PHP 5 with the function-based interface:

```
PDF_set_parameter($p, "licensefile", "/path/to/licensekeys.txt");
```

- In Tcl:

```
PDF_set_parameter $p, "licensefile", "/path/to/licensekeys.txt"
```

Alternatively, you can set the environment variable *PDFLIBLICENSEFILE* to point to your license file. On Windows use the system control panel; on Unix apply a command similar to the following:

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

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

```
HKLM\Software\PDFlib\PDFLIBLICENSEFILE
```

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

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 7.0.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 "0" (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
Tal 40, 80331 München, Germany
http://www.pdflib.com
phone +49 • 89 • 29 16 46 87, fax +49 • 89 • 29 16 46 86
Licensing contact: *sales@pdflib.com*
Support for PDFlib licensees: *support@pdflib.com*

1 Introduction

1.1 Roadmap to Documentation and Samples

We provide the following material to assist you in using PDFlib products successfully:

- The *mini samples* (hello, image, pdfclock, etc.) are available in all packages and for all supported 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.
- 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, Textflow and table formatting, PDF/A and PDF/X creation and other topics. The starter samples demonstrate the basic techniques for achieving a particular goal with PDFlib products. It is strongly recommended to take a look at the starter samples.
- The *PDFlib Tutorial* (this manual), which is contained in all packages as a single PDF document, explains important programming concepts in more detail, using explanation, tables, and small or large 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.
- The *PDFlib 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 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 versions of the PDFlib reference 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 Reference* when working with PDFlib.
- The *PDFlib Cookbook* is a growing 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 collection of sample programs. It is available on the Web at the following URL:

  ```
  www.pdflib.com/developer/cookbook
  ```

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 23, 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 supported language bindings.*

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 eServer iSeries and zSeries. PDFlib itself is written in the C language, but it can be also accessed from several other languages and programming environments which are called language bindings. These language bindings cover all current Web and stand-alone application environments. The Application Programming Interface (API) is easy to learn, and is identical for all bindings. Currently the following bindings are supported:

- COM for use with Visual Basic, Active Server Pages with VBScript or JScript, Borland Delphi, Windows Script Host, and other environments
- ANSI C
- ANSI C++
- Cobol (IBM eServer zSeries)
- Java, including servlets
- .NET for use with C#, VB.NET, ASP.NET, and other environments
- PHP hypertext processor
- Perl
- Python
- REALbasic
- RPG (IBM eServer iSeries)
- 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 7?

The following list discusses the most important new or improved features in PDFlib 7.

Table formatting. PDFlib includes a new table formatter which automatically places rows and columns according to user preferences, and splits tables across multiple pages. Table cells can hold single- or multi-line text, images, or PDF pages, and can be formatted according to a variety of options (e.g. border color, background, cell ruling). The size of table rows and columns is calculated automatically subject to a variety of user preferences.

Textflow enhancements. The Textflow engine for formatting text has been improved:

- Links and other interactive elements can automatically be created from text fragments in a Textflow.
- Text can run around images.
- The Textflow formatter supports leaders,e.g. repeated dots between an entry in a table of contents and the corresponding page number.
- Text contents and formatting options can be supplied separately to the Textflow engine in an arbitrary number of steps. This eliminates the need for creating a buffer containing the full text along with inline formatting options.
- Character classes for formatting decisions can now be redefined, e.g. specify whether the »/« character will be treated as a letter or punctuation in formatting decisions.
- Formatting results can be queried programmatically.

Other formatting features.

- Decimal tabs and leader characters are supported in single-line text (in addition to multi-line Textflows)
- A new stamp function calculates optimized size and position for text stamps across a rectangle.
- Improved query functions for text geometry.

Font handling and Unicode. PDFlib's font engine has been improved as follows:

- Users can query detailed properties of a loaded font, including typographic entries in TrueType/OpenType fonts, number of available glyphs, etc.
- Font subsets can be created for Type 3 fonts.
- Unicode encoding is supported for all font types, including Type 3.
- Text can be supplied in the UTF-32 format, and surrogate pairs can be used for Unicode values beyond the Basic Multilingual Plane (i.e. beyond U+FFFF).
- PDFlib can create artificial font styles if a bold, italic, or bold-italic variant is not available in a font family.
- Glyph replacement can be controlled by the user, e.g. if the *Ohm* glyph is not available in a font the greek letter *Omega* will be used instead. If this glyph still is not available, a user-selected replacement glyph will be used. PDFlib can issue a warning if a required glyph is not available in the font.
- In addition to numerical references, glyphs in a font can be addressed by name, e.g. ligatures or stylistic variations.
- C- and Java-style backslash sequences are recognized in the text. This facilitates handling of control characters or non-ASCII characters regardless of programming language requirements and restrictions.

Improved handling of Chinese, Japanese, and Korean text. PDFlib 7 lifts a number of restrictions related to CJK text handling in earlier versions:

- PDFlib fully supports glyph metrics for all standard CMaps; e.g. Shift-JIS text can be formatted with Textflow.
- Vertical writing mode is supported for all TrueType and OpenType fonts.
- Chinese, Japanese, and Korean codepages (e.g. code page 932 for Shift-JIS) are now supported on all platforms (previously only available on Windows).
- CJK CMaps are now also supported for interactive features such as bookmarks (formerly only for page content).
- Acrobat's predefined CJK fonts can now be used with Unicode encoding.
- Font embedding is no longer forced for OpenType CID fonts loaded with one of the predefined CMaps, resulting in smaller file size.

Matchboxes. The matchbox concept, which is supported in various text and image functions, provides easy reference to the coordinates of formatted text or image objects. This can be used to automatically create annotations and decoration by simple markup (instead of doing coordinate calculations), e.g. create links in Textflow-formatted text, add borders to individual portions of text, highlight text within a formatted paragraph, etc.

pCOS interface integrated in PDI. PDFlib includes the pCOS 2.0 interface which can be used to query arbitrary properties of an existing PDF document via a simple path syntax. This can be used to list fonts, images, and color spaces; query page- or document-related properties, PDF/A or PDF/X status, document info fields or XMP metadata, and many more. Many features have been added to the set of core pCOS features as released in the pCOS 1.0 product in 2005, e.g. image and color space properties, page labels, resources, and others.

PDF import (PDI). PDI implements new workarounds for damaged PDF input (repair mode). A new optimization step can remove redundant objects which may result from importing a number of PDF documents. For example, if several imported PDF documents contain the same sets of fonts, the redundant fonts will no longer be included in the output document but will be removed.

PDF/A for archiving. PDFlib can generate output according to the PDF/A-1a and PDF/A-1b standards, formally known as ISO 19005-1. PDF/A specifies a standardized subset of PDF for long-term preservation and archiving of PDF documents. Existing PDF/A documents can be imported and combined or split; images (any color space) can be converted to PDF/A. While PDF/A-1b preserves the visual appearance of PDF documents, PDFlib users can even create the advanced variant PDF/A-1a which in addition preserves the semantics of the documents.

PDFlib Personalization Server and Block Plugin. Multiple Textflow blocks can be linked so that one block holds the overflow text of a previous block. This allows for more flexible layouts for variable data processing. The new pCOS interface can be used for flexible retrieval of all kinds of block-related information from a PDF.

Interactive elements and 3D animations. Annotations (Web links) can be placed on a specific layer so that they are visible only when the corresponding layer is visible. Layers can now be locked. Stamp and FreeText annotations can be rotated within the rectangle. Links can be created so that the edges of the rectangle don't have to be aligned with the page edges.

3D animations in the U3D format can be embedded in the PDF output, controlled by a variety of options. Actions can be defined to interact with 3D animations.

AES encryption. PDFlib supports 128-bit encryption with the AES algorithm (Advanced Encryption Standard) as supported by Acrobat 7. AES encryption is considered much more secure than earlier crypto schemes.

Other PDF 1.6 (Acrobat 7) features. *UserUnits* allow better document scaling and a wider range of possible page sizes. New document open modes are supported (e.g. attachment pane visible) as well as setting a default print scaling for the document.

Spot colors. The set of supported Pantone spot colors has been updated to the latest 2006 editions provided by Pantone, Inc., including the new Pantone color bridge and new colors in the *metallic* and *pastel* color libraries. Pantone color names are now integrated in the PDFlib Block plugin, and can directly be selected in the user interface for block properties.

Image handling. The clipping path in TIFF and JPEG images will be honored, so that placed images automatically retain the separation of foreground and background without any additional clipping or transparency operations.

XMP metadata. PDFlib automatically creates XMP metadata from document info fields. Users can supply prebuilt XMP metadata streams for the document or other objects, such as page, font, image, imported PDF page, template, or ICC profile. Custom XMP schemas are supported to allow for client-specific metadata.

Tagged PDF. PDFlib's existing support for creating Tagged PDF has been extended: links and other interactive elements can now be included in the document structure tree. This is important for creating fully accessible documents where not only the actual page contents conform to accessibility requirements, but also interactive elements such as links and form fields.

Language bindings. Various improvements in the language bindings, most notably support for newer versions (e.g. Python 2.5) and Unicode support in the Python wrapper.

Error handling. Handling of exceptions and other errors has been streamlined for all language bindings.

Documentation. The documentation has been restructured into two separate main manuals, with an associated PDFlib cookbook which presents code samples along with explanation. Improved coding samples are available in each product package.

1.4 Features in PDFlib/PDFlib+PDI/PPS 7

Table 1.1 lists the major PDFlib features for generating and importing PDF. New or improved features in PDFlib 7 are marked.

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

topic	***features***
PDF output	*PDF documents of arbitrary length, directly in memory (for Web servers) or on disk file*
	Suspend/resume and insert page features to create pages out of order
PDF flavors	*PDF 1.3 – 1.7 (Acrobat 4 – 8), Tagged PDF, PDF/A, PDF/X*
	Linearized (web-optimized) PDF for byteserving over the Web
PDF input	*Import pages from existing PDF documents (only PDFlib+PDI and PPS)*
	pCOS interface for querying details about imported PDF documents[1]
	Delete redundant objects (e.g. identical fonts) across multiple imported PDF documents[1]
	Workarounds for malformed PDF input[1]
Blocks	*PDF personalization with PDFlib blocks for text, image, and PDF data (only PPS)*
	PDFlib Block plugin for creating PDFlib blocks interactively in Adobe Acrobat
	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]
Graphics	*Common vector graphics primitives: lines, curves, arcs, rectangles, etc.*
	Smooth shadings (color blends), pattern fills and strokes
	Transparency (opacity) and blend modes
	Layers: optional page content which can selectively be displayed; annotations can be placed on layers[1]*; layers can be locked*[1]
Fonts	*TrueType (TTF and TTC) and PostScript Type 1 fonts (PFB and PFA, plus LWFN on the Mac)*
	OpenType fonts (TTF, OTF) with PostScript or TrueType outlines
	AFM and PFM PostScript font metrics files
	Directly use fonts which are installed on the Windows or Mac host system
	Font embedding for all font types; subsetting for Type 3[1]*, TrueType and OpenType fonts*
	User-defined (Type 3) fonts for bitmap fonts or custom logos
Text output	*Text output in different fonts; underlined, overlined, and strikeout text*
	Glyphs in a font can be addressed by numerical value, Unicode value, or glyph name[1]
	Kerning for improved character spacing
	Artificial bold and italic font styles[1]
	Proportional widths for standard CJK fonts
	Direct glyph selection for advanced typography[1]
	Configurable replacement of missing glyphs[1]
Internationalization	*Unicode strings for page content, interactive elements, and file names; UTF-8, UTF-16, and UTF-32*[1] *formats, little- and big-endian*
	Support for a variety of 8-bit and legacy CJK encodings (e.g. SJIS; Big5)[1]
	Fetch code pages from the system (Windows, IBM eServer iSeries and zSeries)
	Standard CJK fonts and CMaps for Chinese, Japanese, and Korean text[1]
	Custom CJK fonts in the TrueType and OpenType formats[1]
	Embed Unicode information in PDF for correct text extraction in Acrobat
Images	*Embed BMP, GIF, PNG, TIFF, JPEG, JPEG2000, and CCITT raster images*
	Automatic detection of image file formats (file format sniffing)

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

topic	features
	Interpret clipping paths in TIFF and JPEG images[1]
	Transparent (masked) images including soft masks
	Image masks (transparent images with a color applied), colorize images with a spot color
Color	Grayscale, RGB, CMYK, CIE L*a*b* color
	Integrated support for PANTONE® colors (2006 edition[1]) and HKS® colors; user-defined spot colors
Color management	ICC-based color with ICC color profiles: honor embedded profiles in images, or apply external profiles to images
	Rendering intent for text, graphics, and raster images
	Default gray, RGB, and CMYK color spaces to remap device-dependent colors
Prepress	Generate output conforming to PDF/X-1a, PDF/X-2, and PDF/X-3, including 2003 flavors
	Embed output intent ICC profile or reference standard output intent
	Copy output intent from imported PDF documents (only PDFlib+PDI and PPS)
	Create OPI 1.3 and OPI 2.0 information for imported images
	Separation information (PlateColor)
	Settings for text knockout, overprinting etc.
Archiving	Generate output conforming to PDF/A-1a:2005 and PDF/A-1b:2005[1]
Formatting	Textflow: format arbitrary amounts of text into one or more rectangular areas with hyphenation, font and color changes, justification methods, tabs, leaders[1], control commands; wrap text around images[1]
	Flexible image placement and formatting
	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, or PDF pages, and can be formatted with ruling and shading options.[1]
	Flexible stamping function[1]
	Matchbox concept for referencing the coordinates of placed images or other objects[1]
Security	Encrypt PDF output with RC4 or AES[1] encryption algorithms
	Specify permission settings (e.g. printing or copying not allowed)
	Import encrypted documents (master password required; only PDFlib+PDI and PPS)
Interactive elements	Create form fields with all field options and JavaScript
	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 U3D format[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[1]
	Easily format large amounts of text for Tagged PDF
Metadata	Create XMP metadata from document info fields or from client-supplied XMP streams[1]
	Document information: standard fields (Title, Subject, Author, Keywords) and user-defined fields
Programming	Language bindings for Cobol, COM, C, C++, 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/PDFlib+PDI/PPS 7

1.5 Availability of Features in different Products

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

Table 1.2 Availability of features in different products

feature	API functions, parameters, and options	PDFlib Lite (open source)	PDFlib	PDFlib+PDI	PPS
basic PDF generation	(all except those listed below)	X	X	X	X
language bindings	C, C++, Java, Perl, Tcl, PHP, Python, Ruby	X	X	X	X
language bindings	Cobol, COM, .NET, REALbasic, RPG	–	X	X	X
works on EBCDIC systems	all functions	–	X	X	X
password protection and permission settings	PDF_begin_document( ) with `userpassword`, `masterpassword`, `permissions` options	–	X	X	X
linearized PDF	PDF_begin_document( ) with `linearize` option	–	X	X	X
minimize PDF file size	PDF_begin_document( ) with `optimize` option	–	X	X	X
font subsetting	PDF_load_font( ) with `subsetting` option	–	X	X	X
kerning	PDF_load_font( ) with `kerning` option	–	X	X	X
access Mac and Windows host fonts	PDF_load_font( )	–	X	X	X
access system encodings on Windows, iSeries, zSeries	PDF_load_font( )	–	X	X	X
Unicode encoding and ToUnicode CMaps	PDF_load_font( ) with `encoding=unicode`, `autocidfont`, `unicodemap` parameters	–	X	X	X
numeric, character entity and glyph name references	`charref` option in PDF_fit_textline( ), `charref` parameter	–	X	X	X
proportional glyph widths for standard CJK fonts	PDF_load_font( ) with standard CJK fonts and CMaps	–	X	X	X
glyph ID addressing	PDF_load_font( ) with `encoding=glyphid`	–	X	X	X
CJK legacy encodings	PDF_load_font( ) with standard CMaps or CJK code pages	–	X	X	X
glyph replacement	PDF_load_font( ) with `replacementchar` option	–	X	X	X
extended encoding for PostScript-based OpenType fonts	PDF_load_font( )	–	X	X	X
font properties for Type 3 fonts	PDF_begin_font( ) with options `familyname`, `stretch`, `weight`	–	X	X	X
query font details	PDF_info_font( )	–	X	X	X
Textflow	PDF_add_textflow( ),PDF_create_textflow( ), PDF_delete_textflow( ), PDF_fit_textflow( ), PDF_info_textflow( )	–	X	X	X
Table formatting	PDF_add_table_cell( ), PDF_delete_table( ), PDF_fit_table( ), PDF_info_table( )	–	X	X	X

Table 1.2 Availability of features in different products

feature	*API functions, parameters, and options*	*PDFlib Lite (open source)*	*PDFlib*	*PDFlib+PDI*	*PPS*
spot color	*PDF_makespotcolor( )*	–	X	X	X
color separations	*PDF_begin_page_ext( ) with* separationinfo *option*	–	X	X	X
form fields	*PDF_create_field( ), PDF_create_fieldgroup( ),*	–	X	X	X
JavaScript actions	*PDF_create_action( ) with* type=JavaScript	–	X	X	X
layers	*PDF_define_layer( ), PDF_begin_layer( ), PDF_end_layer( ), PDF_set_layer_dependency( ), PDF_create_action( ) with* type=SetOCGState	–	X	X	X
Multimedia/3D	*PDF_load_3ddata( ) and PDF_create_3dview( ), PDF_create_action( ) with* type=3D	–	X	X	X
Tagged PDF	*PDF_begin_item( ), PDF_end_item( ), PDF_activate_item( ), PDF_begin_document( ) with* tagged *and* lang *options*	–	X	X	X
JPEG2000 images	*PDF_load_image( ) with* imagetype=jpeg2000	–	X	X	X
clipping paths in TIFF and JPEG images	*PDF_load_image( ) with* clippingpathname *and* honorclippingpath *options*	–	X	X	X
PDF/A	*PDF_begin_document( ) with* pdfa *option*	–	X	X	X
PDF/X	*PDF_begin_document( ) with* pdfx *option*	–	X	X	X
ICC profile support	*PDF_load_iccprofile( ), PDF_setcolor( ) with* iccbasedgray/rgb/cmyk, *PDF_load_image( ) with* honoriccprofile *option,* honoriccprofile *parameter, PDF_begin/end_page_ext( ) with* defaultgray/rgb/cmyk *option*	–	X	X	X
*CIE L*a*b* color*	*PDF_setcolor( ) with* type=lab*; Lab TIFF images*	–	X	X	X
OPI support	*PDF_load_image( ) with* OPI-1.3/OPI-2.0 *options*	–	X	X	X
XMP metadata support	*PDF_begin_document( ) with* autoxmp *option;* metadata *option in several functions*	–	X	X	X
PDF import (PDI)	*PDF_open_pdi_document( ), PDF_open_pdi_callback( ), PDF_open_pdi_page( ), PDF_fit_pdi_page( ), PDF_process_pdi( )*	–	–	X	X
Query information from existing PDF with pCOS	*PDF_pcos_get_number( ), PDF_pcos_get_string( ), PDF_pcos_get_stream( )*	–	–	X	X
variable data processing and personalization with blocks	*PDF_fill_textblock( ), PDF_fill_imageblock( ), PDF_fill_pdfblock( )*	–	–	–	X
PDFlib Block plugin for Acrobat	*interactively create PDFlib blocks for use with PPS*	–	–	–	X

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 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.

2.2 COM Binding

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

2.3 C Binding

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

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);
```

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

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

An important task of the catch clause is to clean up PDFlib internals using *PDF_delete( )* and the pointer to the PDFlib object. *PDF_delete( )* will also close the output file if necessary. 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.

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.

2.4 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*. The corresponding *pdflib.cpp* module should be linked against the application which in turn should be linked against the generic PDFlib C library.

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

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

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

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

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.

Unicode in the C++ Language Binding. C++ users must be aware of a pitfall related to the compiler automatically converting literal strings to the C++ string type which is expected by the PDFlib API functions: this conversion supports embedded null characters only if an explicit length parameter is supplied. For example, the following will not work since the string will be truncated at the first null character:

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

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

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

2.5 Java Binding

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

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

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

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*. Using the source files provided in the PDFlib Lite distribution you can generate an abbreviated HTML version of the PDFlib API reference (this manual) using the *javadoc* utility since the PDFlib class contains the necessary *javadoc* comments. Comments and restrictions for using PDFlib with specific Java environments may be found in text files in the distribution set.

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

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

You can check the value of this property as follows:

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

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

- Unix: the library *libpdf_java.so* (on Mac OS X: *libpdf_java.jnilib)* must be placed in one of the default locations for shared libraries, or in an appropriately configured directory.
- Windows: the library *pdf_java.dll* must be placed in the Windows system directory, or a directory which is listed in the PATH environment variable.

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

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

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

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

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)
```

2.6 .NET Binding

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

2.7 Perl Binding

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

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

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

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

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

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

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

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

Windows. PDFlib supports the ActiveState port of Perl 5 to Windows, also known as ActivePerl.[2] Both *pdflib_pl.dll* and *pdflib_pl.pm* will be searched in the current directory, or the directory printed by the following Perl command:

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

Typical output of the above command looks like

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

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

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

[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);
...
PDF_set_parameter($p, "textformat", "utf8");
$font = PDF_load_font($p, "Arial Unicode MS", "unicode", "");
PDF_setfont($p, $font, 24.0);
PDF_set_text_pos($p, 50, 700);
PDF_show($p, uhex("U+65E5 U+672C U+8A9E"));
```

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

```
PDF_set_parameter($p, "textformat", "utf16le");
$font = PDF_load_font($p, "Arial Unicode MS", "unicode", "");
PDF_setfont($p, $font, 24.0);
PDF_set_text_pos($p, 50, 700);
PDF_show($p, "\xE5\x65\x2C\x67\x9E\x8A");
```

The third example works with Shift-JIS. Except on Windows systems it requires access to the *90ms-RKSJ-H* CMap for string conversion:

```
PDF_set_parameter($p, "SearchPath", "../../../resource/cmap");
$font = PDF_load_font($p, "Arial Unicode MS", "cp932", "");
PDF_setfont($p, $font, 24.0);
PDF_set_text_pos($p, 50, 700);
PDF_show($p, "\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 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
```

2.8 PHP Binding

Installing the PDFlib PHP Edition. Detailed information about the various flavors and options for using PDFlib with PHP[1], 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.

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

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

  ```
  extension=libpdf_php.so        ; for Unix
  extension=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
  ```

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

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

See below for more details on these PHP 5 features.

Modified error return for PDFlib functions in PHP. Since PHP uses the convention of returning the value 0 (FALSE) when an error occurs within a function, all PDFlib functions have been adjusted to return 0 instead of -1 in case of an error. This difference is noted in the function descriptions. However, take care when reading the code fragment examples in Section 3, »PDFlib Programming«, page 43, since these use the usual PDFlib convention of returning -1 in case of an error.

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

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

[1] See www.php.net

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

In order to provide platform-independent file name handling use of PDFlib's SearchPath facility (see Section 3.1.3, »Resource Configuration and File Searching«, page 46) is strongly recommended.

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

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

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

Exception handling in PHP 5. Since PHP 5 supports structured exception handling, PDFlib exceptions will be propagated as PHP exceptions. PDFlib will throw an exception of the 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;
}
```

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

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

2.9 Python Binding

Installing the PDFlib Python Edition. The Python[1] extension mechanism works by loading shared libraries at runtime. For the PDFlib binding to work, the Python interpreter must have access to the PDFlib Python wrapper:

- Unix: the library *pdflib_py.so* (on Mac OS X: *pdflib_py.dylib)* will be searched in the directories listed in the PYTHONPATH environment variable.
- Windows: the library *pdflib_py.dll* will be searched in the directories listed in the PYTHONPATH environment variable.

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

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

[1] *See www.python.org*

2.10 REALbasic Binding[1]

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

[1] *See www.realbasic.com*

2.11Ruby Binding

Installing the PDFlib Ruby Edition. The Ruby[1] 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/Linux/Unix: *PDFlib.so*; on Mac OS X: *PDFlib.dylib)* 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/<rubyversion>/
```

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 a special 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"
```

Ruby on Rails. Ruby on Rails[2] is an open-source framework which facilitates Web development with Ruby. The PDFlib extension for Ruby can be used with Ruby on Rails; examples are included in the package. Follow these steps to run the PDFlib examples for Ruby on Rails:

- Install Ruby.
- Install Ruby on Rails.
- Unpack the PDFlib package for Ruby which contains samples for Ruby on Rails.
- Change to the *bind/ruby/RubyOnRails* directory and start the Ruby web server:

  ```
  ruby script/server
  ```

- Point your browser to *http://localhost:3000*.

The code for the PDFlib samples can be found in *app/controllers/pdflib_controller.rb*.

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.

[1] *See www.ruby-lang.org/en*
[2] *See www.rubyonrails.org*

2.12 RPG Binding

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

Since all functions provided by PDFlib are implemented in the C language, you have to add *x'00'* at the end of each string value passed to a PDFlib function. All strings returned from PDFlib will have this terminating *x'00'* as well.

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

```
d/copy QRPGLESRC,PDFLIB
```

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

```
d/copy PDFsrclib/QRPGLESRC,PDFLIB
```

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

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

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

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

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

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

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

```
*****************************************************************************************
     d/copy QRPGLESRC,PDFLIB
*****************************************************************************************
     d p                 S                *
     d font              s              10i 0
      *
     d error             s              50
      *
     d errhdl            s                *   procptr
      *
      *  Prototype for exception handling procedure
      *
     d errhandler        PR
     d  p                              *   value
     d  type                         10i 0 value
     d  shortmsg                   2048
*****************************************************************************************
     c                   clear                   error
      *
*        Set the procedure pointer to the ERRHANDLER procedure.
*
     c                   eval      errhdl=%paddr('ERRHANDLER')
      *
     c                   eval      p=pdf_new2(errhdl:*null:*null:*null:*null)

...PDFlib instructions...

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

```
 *    Always call PDF_delete to clean up PDFlib
c                   callp     PDF_delete(p)
 *
c                   return
 *
p errhandler      E
```

2.13 Tcl Binding

Installing the PDFlib Tcl Edition. The Tcl [1]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\tcl8.3\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 PDFlib Programming

3.1 General Programming

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 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 the 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 conditions its behavior 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 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.
- *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
 */
```

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)

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

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

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

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

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

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

- Minimize memory usage: it is recommended to call *PDF_delete_pvf( )* immediately after the API call which accepted the virtual file name, and another time after *PDF_* *close( )*. The second call is required because PDFlib may still need access to the data so that the first call refuses to unlock the virtual file. However, in some cases the first call will already free the data, and the second call doesn't do any harm. The client may free the file contents only when *PDF_delete_pvf( )* succeeded.

- Optimize performance by reusing virtual files: some clients may wish to reuse some data (e.g., font definitions) within various output documents, and avoid multiple create/delete cycles for the same file contents. In this case it is recommended not to call *PDF_delete_pvf( )* as long as more PDF output documents using the virtual file will be generated.
- Lazy programming: if memory usage is not a concern the client may elect not to call *PDF_delete_pvf( )* at all. In this case PDFlib will internally delete all pending virtual files in *PDF_delete( )*.

In all cases the client may free the corresponding data only when *PDF_delete_pvf( )* returned successfully, or after *PDF_delete( )*.

3.1.3 Resource Configuration and File Searching

In most advanced applications PDFlib needs access to resources such as font file, encoding definition, ICC color profiles, etc. In order to make PDFlib's resource handling platform-independent and customizable, a configuration file can be supplied for describing the available resources along with the names of their corresponding disk files. In addition to a static configuration file, dynamic configuration can be accomplished at runtime by adding resources with *PDF_set_parameter( )*. For the configuration file we dug out a simple text format called *Unix PostScript Resource* (UPR) which came to life in the era of Display PostScript, and is still in use on several systems. However, we 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*
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 which are already built into PDFlib, see PDFlib Reference for a complete list)*

Redundant resource entries should be avoided. For example, do not include multiple entries for a certain font's metrics data. Also, the font name as configured in the UPR file should exactly match the actual font name in order to avoid confusion (although PDFlib does not enforce this restriction).

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

- Lines can have a maximum of 255 characters.
- A backslash '\' escapes newline characters. This may be used to extend lines. Use two backslashes in order to create a single literal backslash.
- An isolated period character ' . ' serves as a section terminator.
- All entries are case-sensitive.
- Comment lines may be introduced with a percent '%' character, and terminated by the end of the line. A preceding backslash can be used to create literal percent characters which do not start a comment.
- Whitespace is ignored everywhere except in resource names and file names.

UPR files consist of the following components:

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

  ```
  PS-Resources-1.0
  ```

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

File searching and the SearchPath resource category. PDFlib reads a variety of data items, such as raster images, font outline and metrics information, encoding definitions, PDF documents, and ICC color profiles from disk files. In addition to relative or 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 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 an entry from the registry. The following registry entry may contain a list of path names separated by a semicolon ';' character:

```
HKLM\SOFTWARE\PDFlib\PDFlib\7.0.0\SearchPath
```

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

```
/pdflib/7.0.0/fonts
/pdflib/7.0.0/bind/data
```

- On IBM zSeries systems with MVS the SearchPath feature is not supported.
- On OpenVMS logical names can be supplied as *SearchPath*.

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
.
ICCProfile
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:

- If the environment variable *PDFLIBRESOURCE* is defined PDFlib takes its value as the name of the UPR file to be read. If this file cannot be read an exception will be thrown.

- If the environment variable *PDFLIBRESOURCE* is not defined PDFlib tries to open a file with the following name:

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

  If this file cannot be read no exception will be thrown.
- On Windows PDFlib will additionally try to read the registry entry

  ```
  HKLM\SOFTWARE\PDFlib\PDFlib\7.0.0\resourcefile
  ```

  The value of this entry (which will be created by the PDFlib installer, but can also be created by other means) will be taken as the name of the resource file to be used. If this file cannot be read an exception will be thrown.
- The client can force PDFlib to read a resource file at runtime by explicitly setting the *resourcefile* parameter:

  ```
  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 5.3.1, »Searching for Fonts«, page 101.

Querying resource values. In addition to setting resource entries 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 one big chunk at the end (after *PDF_end_document( )*). Interleaving production and consumption of the PDF data has several advantages. Firstly, since not all data must be kept in memory, the memory requirements are reduced. Secondly, such a scheme can boost performance since the first chunk of data can be transmitted over a slow link while the next chunk is still being generated. However, the total length of the generated data will only be known when the complete document is finished.

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

```
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 Using PDFlib on EBCDIC-based Platforms

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

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

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

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

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

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

3.1.6 Large File Support

In this section the term »large file« is used for files with a size of more than 2 GB. Although there doesn't seem to be any need for such large files for the average user, there are actually enterprise applications which create or process single large files containing large numbers of, say, invoices or statements. In such a scenario the file size may exceed 2 GB.

PDFlib supports large output files, i.e. it can create PDF output with more than 2 GB. PDI supports processing of large input files as well. However, large file support is only available on platforms where the underlying operating system supports large files natively. Obviously, the file system in use must also support large files. Note that Acrobat 6 and older versions are unable to process large files. However, Acrobat 7 properly deals with large files.

Note *Imported files other than PDF, such as fonts and images, can not exceed the 2 GB limit. PDF output fragments fetched with the PDF_get_buffer( ) interface are also subject to this limit. Finally, PDF output files are generally limited to* 10^{10} *bytes, which is roughly 9.3 GB.*

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 an alternative, the *userunit* option in *PDF_begin/end_page_ext( )* (PDF 1.6) can be specified to supply a scaling factor for the whole page.

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.

Acrobat 6/7 (full version only, not the free Reader) also has a helpful facility. Simply choose *View, Navigation tabs, Info* to display a measurement palette. 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. To change the display units go to *Edit, Preferences, [General...], Units & Guides [or Page Units]* and choose one of Points, Inches, Millimeters, Picas, Centimeters. You can also go to *View, Navigation Tabs, Info* and select a unit from the *Options* menu.

Don't be mislead 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 the respective fit function *(PDF_fit_textline( ), 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 an angle of 45° degrees:

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

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 rectangle of the corner, 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");
```

Alternatively, the *topdown* parameter can be used, but it must not be set within a page description (but only between pages). For the sake of completeness we'll list the detailed consequences of establishing a top-down coordinate system below.

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

- All function parameters which are designated as »coordinates« in the function descriptions. Some examples: *x, y* in *PDF_moveto( ); x, y in PDF_circle( ), x, y* (but not *width* and *height!) in PDF_rect( ); llx, lly, urx, ury in PDF_create_annotation( ))*.

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

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

3.2.2 Page Size

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. PDFlib will throw a non-fatal exception if Acrobat's page size limits are exceeded. The page size limits for Acrobat are shown in Table 3.2. 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 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 Paths

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

- 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 the page size by default) with the intersection of the current clipping area and the area enclosed by the path.
- Merely terminating the path results in an invisible path, which will nevertheless be present in the PDF file. This will only rarely be required.

It is an error to construct a path without applying any of the above operations to it. PDFlib's scoping system ensures that clients obey to this restriction. 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.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.

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 159). 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();
...
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()*: this is not a big restriction since images can be opened outside of a template definition, and freely be used within a template (but not opened).
- 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.

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

3.3 Working with Color

3.3.1 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).

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

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

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

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

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

3.3.2 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 9.4, »PDF/X for Print Production«, page 202). 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.

PANTONE® colors. PANTONE Colors are well-known and widely used on a world-wide basis. PDFlib fully supports the Pantone Matching System• , totalling ca. 24 000 swatches. All color swatch names from the digital color libraries listed in Table 3.3 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, the function call will fail. 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 3.3 PANTONE spot color libraries built into PDFlib

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

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, the function call will fail. 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 */
```

3.3.3 Color Management and ICC Profiles

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

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

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

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.
- Use the file name determined in the previous step to locate a disk file by trying the following combinations one after another:

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

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

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

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

Acceptable ICC profiles. The type of acceptable ICC profiles depends on the *usage* parameter supplied to *PDF_load_iccprofile( )*:

- If *usage=outputintent,* only output device (printer) profiles will be accepted in PDF/X mode, and any profile in PDF/A mode.

[1] See www.color.org

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

The sRGB color space and sRGB ICC profile. PDFlib supports the industry-standard RGB color space called sRGB (formally IEC 61966-2-1). 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 *profilename=sRGB*.

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

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

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

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

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* parameters 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", 0, "usage=iccbased");
p.set_value("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 9.4, »PDF/X for Print Production«, page 202, and Section 9.5, »PDF/A for Archiving«, page 207.

3.4 Interactive Elements

3.4.1 Examples for Creating Interactive Elements

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:

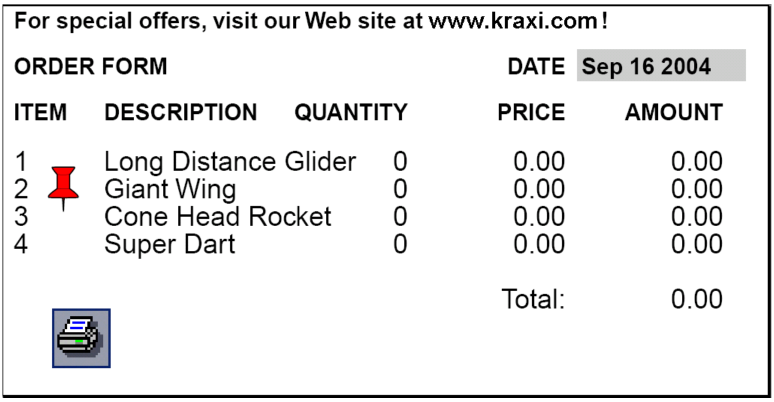
- At the top right there is an invisible Web link to *www.kraxi.com* at the text *www.kraxi.com*. Clicking this area will bring up the corresponding Web page.
- A gray form field of type text is located below the Web link. Using JavaScript code it will automatically be filled with the current date.
- 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 with a thin black border. Initially this is convenient for precise positioning, but we disabled the border with *linewidth=0*.



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

```
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 matchbox={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 7.5, »Matchboxes«, page 176.

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.

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.

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={rgb 1 1 0} " +
        "bordercolor={rgb 0 0 0} 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);
```

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 initally 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.

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

3.4.2 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 fragements 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);
```

In order to specify the various formats which are supported in Acrobat you must use appropriate functions in the JavaScript code. Table 3.4 lists the JavaScript function names for the *keystroke* and *format* actions for all supported formats; the function parameters are described in Table 3.5. These functions must be used similarly to the example above.

Table 3.4 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 3.5 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*

Table 3.5 Parameters for the JavaScript formatting functions

parameters	***explanation and possible values***	
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*

Form fields activate the document’s dirty flag. When a PDF document containing form fields is closed in Acrobat, it will ask whether you want to save the file, even if you didn’t touch any fields. In technical terms, opening a PDFlib-generated PDF with form fields will cause the document’s dirty flag to be set, i.e. Acrobat considers it as changed. While usually this doesn’t really matter since the user will want to fill the form fields anyway, some users may consider this behavior inelegant and annoying. You can work around it with a small JavaScript which resets the document’s dirty flag after loading the file. Use the following idiom to achieve this:

```
/* ...create some form fields... */
p.create_field("100, 500, 300, 600, "field1", "textfield", "..."

/* Create a JavaScript action which will be hooked up in the document */
action = p.create_action("JavaScript", "script={this.dirty=false;}");
...
```

```
String  optlist = "action={open=" + action + "}";
p.end_document(optlist);
```

4 Unicode and Legacy Encodings

4.1 Overview

Unicode support in PDFlib. PDFlib's text handling is based on the Unicode standard[1], almost identical to ISO 10646. Since most modern development environment 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. In particular, PDFlib supports traditional 8-bit encodings (e.g. Windows ANSI, Latin-1) and multi-byte CJK encodings (e.g. Shift-JIS, Big 5).

Although the majority of text will be created on PDF pages, PDFlib's concepts for string handling apply to other areas as well, e.g. text for interactive features such as bookmarks.

Many print and online publications cover the Unicode standard; some important concepts are summarized in Section 4.2, »Important Unicode Concepts«, page 74.

8-bit encodings. 8-bit encodings (also called single-byte encodings) map each byte in a text string to a single character, and are thus limited to 256 different characters at a time (the value 0 is generally not used). A common example of an 16-bit encodings is the Windows ANSI encoding, which is a superset of ISO 8859-1, also called Latin-1. 8-bit encodings are discussed in more detail in Section 4.4, »8-Bit Encodings«, page 81.

Multi-byte CJK encodings. Because of the large number of required characters 8-bit encodings are not suitable for Chinese, Japanese, and Korean text. 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. In PDF several dozens of CJK legacy encodings are supported via predefined CMaps.

CJK CMaps are discussed in more detail in Section 4.5, »Encodings for Chinese, Japanese, and Korean Text«, page 85.

[1] *See www.unicode.org*

4.2 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).

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), i.e. characters in the range U+0000...U+FFFF 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.

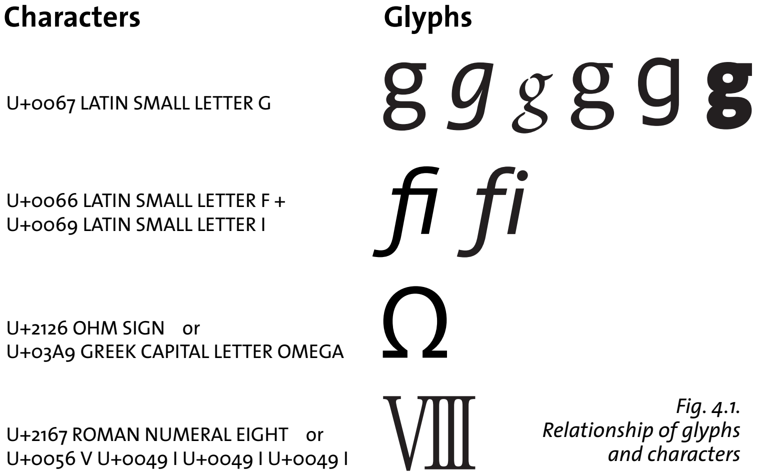

Characters
Glyphs
U+0067 LATIN SMALL LETTER G
g g g g g g
U+0066 LATIN SMALL LETTER F +
U+0069 LATIN SMALL LETTER I
ﬁ fi
U+2126 OHM SIGN or
U+03A9 GREEK CAPITAL LETTER OMEGA
Ω
U+2167 ROMAN NUMERAL EIGHT or
U+0056 V U+0049 I U+0049 I U+0049 I
Ⅷ

Fig. 4.1. Relationship of glyphs and characters

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*
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[1]
UTF-32 little-endian	*FF FE 00 00*	ÿþ ? ?[1]

[1] *There is no standard graphical representation of null bytes.*

4.3 Strings in PDFlib

4.3.1 String Types in PDFlib

PDF and operating system requirements impose different string handling in PDFlib depending on the purpose of a string. The PDFlib API defines and uses the following string types:

- Content strings: these will be used to create genuine page content (page descriptions) according to the encoding chosen by the user for a particular font. All *text* parameters of the page content functions fall in this class.
- Hypertext strings: these are mostly 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.
- 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.

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-compatible language bindings. The details of string handling depend on the language binding, and are discussed in Section 4.3.2, »Strings in Unicode-aware Language Bindings«, page 76 and Section 4.3.3, »Strings in non-Unicode-aware Language Bindings«, page 77.

4.3.2 Strings in Unicode-aware Language Bindings

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

- COM
- .NET
- Java
- 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 hypertext strings will always arrive in PDFlib in *utf16* format and *unicode* encoding.
- Since the language environment always passes strings in UTF-16 to PDFlib, UTF-8 can not be used with Unicode-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 4.5, »Encodings for Chinese, Japanese, and Korean Text«, page 85) 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 hypertext strings and name strings 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 23, provide more details regarding useful Unicode string conversion methods provided by common language environments.

4.3.3 Strings in non-Unicode-aware Language Bindings

The following PDFlib language bindings are not Unicode-aware:

- C (no native string data type available)
- C++
- Cobol (no native string data type available)
- Perl
- PHP
- old-style Python binding for compatibility with PDFlib 6
- old-style RPG binding (no native string data type available)
- Ruby

In language bindings which do not support a native string data type (i.e. C, Cobol, RPG) 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.

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 PDF_utf32_to_utf16( )* conversion functions which can be used to create UTF-8 or UTF-16 strings for passing them to PDFlib.

The language-specific sections in Chapter 2, »PDFlib Language Bindings«, page 23, 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 4.2 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*.

Table 4.2 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	*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).* *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.* *This setting will provide proper text interpretation in most environments which do not use Unicode natively.*

Although the *textformat* setting is in effect for all encodings, it will be most useful for *unicode* encoding. Table 4.3 details the interpretation of text strings for various combinations of encodings and *textformat* settings.

Table 4.3 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 81*	
U+XXXX	*8-bit codes will be added to the offset XXXX to address Unicode values*	*convert Unicode values to 8-bit codes according to the chosen Unicode offset*
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 chosen 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]*. PDFlib will throw an exception if it is not a content string and no 8-bit encoding is found in the font (8-bit encodings are available in Type 1 and Type 3 fonts).*
glyphid	*8-bit codes are glyph ids from 0 to 255*	*Unicode values will be interpreted as glyph ids*[2]

[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 issue a warning and replace it with glyph id 0.*

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 options. By looking for a BOM at the beginning of an option, PDFlib decides how to interpret it. The BOM will be used to determine the format of the string, and the string type (content string, hypertext string, or name string as defined above) will be used to determine the appropriate encoding. More precisely, interpreting a string option works as follows:

- If the option starts with a UTF-8 BOM *(0xEF 0xBB 0xBF)* it will 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).

4.4 8-Bit Encodings

8-bit encodings (also called single-byte encodings) map each byte in a text string to a single character, and are thus limited to 256 different characters at a time.

Table 4.4 lists the predefined encodings in PDFlib, and details their use with several important classes of fonts. It is important to realize that certain scripts or languages have requirements which cannot be met by common fonts. For example, Acrobat's core fonts do not contain all characters required for ISO 8859-2 (e.g. Polish), while PostScript 3, OpenType Pro, and TrueType »big fonts« do.

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

Notes on the macroman encoding. This encoding reflects the Mac OS character set, albeit with the old currency symbol at position 219 = 0xDB, and not the Euro glyph as redefined by Apple (this incompatibility is dictated by the PDF specification). The *macroman_apple* encoding is identical to *macroman* except for the following differences:

- Position 219 = 0xDB in *macroman_apple* holds the Euro glyph instead of the currency symbol.
- The *macroman_apple* encoding includes the greek/mathematical symbols as defined in the Mac OS character set. Although these are available in the *macroman_apple* encoding, the required glyphs are contained only in few fonts.

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

- on Mac OS Classic it will be mapped to *macroman;*
- on IBM eServer zSeries with MVS or USS it will be mapped to *ebcdic;*
- on IBM eServer 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.

Encoding *host* is used as the default encoding for Name strings in non-Unicode-aware language bindings, since this is the most appropriate encoding for file names etc.

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 Mac OS Classic: *macroman*
- On IBM eServer iSeries: the current job's encoding *(IBMCCSID000000000000)*
- On IBM eServer zSeries: *ebcdic* (=code page 1047).

Table 4.4 Availability of glyphs for predefined encodings in several classes of fonts: some languages cannot be represented with Acrobat's core fonts.

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

[1] Core fonts shipped with Acrobat 4/5 (original Adobe Latin character set; generally Type 1 Fonts since 1982)
[2] The information in the table relates to the Times and Helvetica font families. The Courier font family which is used in Acrobat contains fewer glyphs, and does not cover iso8859-2, iso8859-4, iso8859-9, iso8859-10, iso8859-13, and iso8859-16.
[3] Extended Adobe Latin character set (CE-Fonts), generally Type 1 Fonts shipped with PostScript 3 devices
[4] Adobe OpenType Pro fonts contain more glyphs than regular OpenType fonts.
[5] Windows TrueType fonts containing large glyph complements, e.g. Tahoma

For symbol fonts the keyword *auto* will be mapped to *builtin* encoding. While automatic encoding is convenient in many circumstances, using this method will make your PDFlib client programs inherently non-portable.

Tapping system code pages. PDFlib can be instructed to fetch code page definitions from the system and transform it appropriately for internal use. This is very convenient since it frees you from implementing the code page definition yourself. Instead of supplying the name of a built-in or user-defined encoding for *PDF_load_font( )*, simply use 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 5.6.2, »Custom CJK Fonts«, page 117, 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 eServer iSeries any *Coded Character Set Identifier* (CCSID) can be used. The CCSID must be supplied as a string, and PDFlib will apply the prefix *IBMCCSID* to the supplied code page number. PDFlib will also add leading 0 characters if the code page number uses fewer than 5 characters. Supplying 0 (zero) as the code page number will result in the current job's encoding to be used:

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

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

  ```
  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 Searching«, page 46) 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. The next example shows a snippet from a Unicode code page:

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

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

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

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

PDFlib's internal glyph list contains more than 6500 glyph names. Encoding files are required for PostScript fonts with non-standard glyph names, while code pages are more convenient when dealing with Unicode-based TrueType or OpenType fonts.

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

4.5 Encodings for Chinese, Japanese, and Korean Text

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 4.5. 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. Tables with all supported characters are available from Adobe[1].

Note Unicode-aware language bindings support only use Unicode-compatible CMaps (UCS2 or UTF16). Other CMaps can not be used (see Strings in Unicode-aware Language Bindings).

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.

For Unicode CMaps the high-order byte of a character must appear first. Alternatively, the byte ordering and text format can be selected with the *textformat* parameter provided a Unicode CMap (UCS-2 or UTF-16) is used.

Since several of the supported encodings may contain null characters in the text strings, C developers must take care not to use the *PDF_show( )* etc. functions, but instead *PDF_show2( )* etc. which allow for arbitrary binary strings along with a length parameter. For all other language bindings, the text functions support binary strings, and *PDF_show2( )* etc. are not required.

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.

[1] *See partners.adobe.com/asn/tech/type/cidfonts.jsp for a wealth of resources related to CID fonts, including tables with all supported glyphs (search for »character collection«).*

Table 4.5 Predefined CMaps for Japanese, Chinese, and Korean text (from the PDF Reference)

locale	CMap name	character set and text format
Simplified Chinese	UniGB-UCS2-H UniGB-UCS2-V	Unicode (UCS-2) encoding for the Adobe-GB1 character collection
	UniGB-UTF16-H UniGB-UTF16-V	Unicode (UTF-16BE) encoding for the Adobe-GB1 character collection. Contains mappings for all characters in the GB18030-2000 character set.
	GB-EUC-H GB-EUC-V	Microsoft Code Page 936 (charset 134), GB 2312-80 character set, EUC-CN encoding
	GBpc-EUC-H GBpc-EUC-V	Macintosh, GB 2312-80 character set, EUC-CN encoding, Script Manager code 2
	GBK-EUC-H, -V	Microsoft Code Page 936 (charset 134), GBK character set, GBK encoding
	GBKp-EUC-H GBKp-EUC-V	Same as GBK-EUC-H, but replaces half-width Latin characters with proportional forms and maps code 0x24 to dollar ($) instead of yuan (¥).
	GBK2K-H, -V	GB 18030-2000 character set, mixed 1-, 2-, and 4-byte encoding
Traditional Chinese	UniCNS-UCS2-H UniCNS-UCS2-V	Unicode (UCS-2) encoding for the Adobe-CNS1 character collection
	UniCNS-UTF16-H UniCNS-UTF16-V	Unicode (UTF-16BE) encoding for the Adobe-CNS1 character collection. Contains mappings for all of HKSCS-2001 (2- and 4-byte character codes)
	B5pc-H, -V	Macintosh, Big Five character set, Big Five encoding, Script Manager code 2
	HKscs-B5-H HKscs-B5-V	Hong Kong SCS (Supplementary Character Set), an extension to the Big Five character set and encoding
	ETen-B5-H, -V	Microsoft Code Page 950 (charset 136), Big Five with ETen extensions
	ETenms-B5-H ETenms-B5-V	Same as ETen-B5-H, but replaces half-width Latin characters with proportional forms
	CNS-EUC-H, -V	CNS 11643-1992 character set, EUC-TW encoding
Japanese	UniJIS-UCS2-H, -V	Unicode (UCS-2) encoding for the Adobe-Japan1 character collection
	UniJIS-UCS2-HW-H UniJIS-UCS2-HW-V	Same as UniJIS-UCS2-H, but replaces proportional Latin characters with half-width forms
	UniJIS-UTF16-H UniJIS-UTF16-V	Unicode (UTF-16BE) encoding for the Adobe-Japan1 character collection. Contains mappings for all characters in the JIS X 0213:1000 character set.
	83pv-RKSJ-H	Mac, JIS X 0208 with KanjiTalk6 extensions, Shift-JIS, Script Manager code 1
	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
Korean	UniKS-UCS2-H, -V	Unicode (UCS-2) encoding for the Adobe-Korea1 character collection
	UniKS-UTF16-H, -V	Unicode (UTF-16BE) encoding for the Adobe-Korea1 character collection
	KSC-EUC-H, -V	KS X 1001:1992 character set, EUC-KR encoding
	KSCms-UHC-H KSCms-UHC-V	Microsoft Code Page 949 (charset 129), KS X 1001:1992 character set plus 8822 additional hangul, Unified Hangul Code (UHC) encoding
	KSCms-UHC-HW-H KSCms-UHC-HW-V	Same as KSCms-UHC-H, but replaces proportional Latin characters with half-width forms
	KSCpc-EUC-H	Mac, KS X 1001:1992 with Mac OS KH extensions, Script Manager Code 3

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 4.6 can be used. These code pages will be mapped internally to the corresponding CMap (e.g. *cp932* will be mapped to *90ms-RKSJ-H/V)*. Because of this mapping the appropriate CMaps must be configured. The *textformat* parameter must be set to *auto*, and the text must be supplied in a format which is compatible with the chosen code page.

Table 4.6 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*

4.6 Addressing Characters and Glyphs

Some environments require the programmer to write source code in 8-bit encodings (such as *winansi, macroman,* or *ebcdic)*. This makes it cumbersome to include isolated Unicode characters in 8-bit encoded text without changing all characters in the text to multi-byte encoding. In order to aid developers in this situation, PDFlib supports character references, a method known from markup languages such as SGML and HTML.

4.6.1 Escape Sequences

PDFlib supports a method for easily incorporating arbitrary values within text strings via escape sequences (similar sequences are supported in many programming languages). 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 in symbol fonts.

Escape sequences are a shortcut for creating certain byte values. The values created from escape sequences will always be interpreted in the selected encoding (unlike character references, which will always be interpreted in Unicode). The byte values resulting from resolving 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.

Table 4.7 Escape sequences for byte values

Sequence	*Mac, Windows, Linux, Unix*	*EBCDIC platforms (zSeries/iSeries)*	*common interpretation*
\f	*0C*	*0C*	*form feed*
\n	*0A*	*15/25*	*line feed*
\r	*0D*	*0D*	*carriage return*
\t	*09*	*05*	*horizontal tabulation*
\v	*0B*	*0B*	*line tabulation*
\	*5C*	*E0*	*backslash*
\xNN	*two hexadecimal digits specifying a byte value*		
\NNN	*three octal digits (up to \377) specifying a byte value*		

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

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

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= utf16le* or *utf16be* escape sequences must be expressed as two byte values according to the selected format. 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 *glyphcheck* and *errorpolicy* settings.

4.6.2 Character References and Glyph Name References

HTML-style character references. PDFlib supports all numeric character references and character entity references defined in HTML 4.0[1]. Numeric character references can be supplied in decimal or hexadecimal notation; they will always be interpreted as Unicode values.

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 only control characters.*

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

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

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

In addition to the HTML-style references mentioned above PDFlib supports custom character entity references which can be used to specify control characters for Textflows. Table 4.8 lists these additional character references.

If a character reference cannot be resolved (e.g. *&#* followed by invalid decimal digits, or & followed by an unknown character name) an exception will be thrown. For content strings the behavior is controlled by the *glyphcheck* and *errorpolicy* settings.

Glyph name references. A font may contain glyphs which are not directly accessible because the corresponding Unicode values are not known in advance (e.g. PUA assignments) or because they do not even have Unicode values in the font. Although all glyphs in a font can be addressed via the *glyphid* encoding, this is very cumbersome and does not fit Unicode workflows. As a useful facility glyph name references can be used. These are similar to character references, but use a slightly different syntax and refer to the glyph by name (note that the first period character is part of the syntax, while the second is part of the glyph name in the examples):

```
&.T.swash;
&.orn.15;
```

[1] *See www.w3.org/TR/REC-html40/charset.html#h-5.3*

Table 4.8 Control characters and their meaning in Textflows

Unicode character	*entity name*	*equiv. Textflow option*	*meaning within Textflows in Unicode-compatible fonts*
U+0020	*SP, space*	*space*	*align words and break lines*
U+00A0	*NBSP, nbsp*	*(none)*	*(no-break space) space character which will not break lines*
U+0009	*HT, hortab*	*(none)*	*horizontal tab: will be processed according to the* ruler, tabalignchar, *and* tabalignment *options*
U+002D	*HY, hyphen*	*(none)*	*separator character for hyphenated words*
U+00AD	*SHY, shy*	*(none)*	*(soft hyphen) hyphenation opportunity, only visible at line breaks*
U+000B *U+2028*	*VT, verttab* *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*	*nextparagraph*	*(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.

Glyph name references are useful for alternate forms (e.g. swash characters, tabular figures) and glyphs without any specific Unicode semantics (symbols, icons, and ornaments). The general syntax is *&.<name>;* where *name* is a glyph name which will be substituted as follows:

- Font-specific glyph names from OpenType fonts (but not OpenType CID fonts) can be used for content strings (since these are always related to a particular font);
- Glyph names used in encodings can be used for content strings;
- Names from the Adobe Glyph List (including the *uniXXXX* and *uXXXX* forms) plus certain common »misnamed« glyph names will always be accepted for content strings and hypertext strings;

If no glyph can be found for the name specified in a glyph name reference an exception will be thrown. For content strings the behavior is controlled by the *glyphcheck* and *errorpolicy* settings. Glyph name references cannot be used with *glyphid* or *builtin* encoding.

Using character and glyph name references. Character and glyph name references can be used in all content strings, hypertext strings, and name strings, e.g. in text which will be placed on the page using the *show* or *Textflow* functions, as well as in text supplied to the hypertext functions. Character references will not be processed in text with *builtin* encoding. However, you can use glyph name references for symbolic fonts by using *unicode* encoding. In this case all glyphs must be addressed by name; you cannot mix numerical codes and glyph names.

Character and glyph name references will not be converted by default; you must explicitly set the *charref* parameter or option to *true* if you want to use character or glyph name references in all content strings:

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

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

Character and glyph name references can also be enabled for Textflow processing by supplying the *charref* option to *PDF_add_textflow( )* (either directly or as an inline option), *PDF_fit_textline( ),* or *PDF_fill_textblock( ).*

If character and glyph name references are enabled, you can supply numeric references, entity references, and glyph name references in 8-bit-encoded text:

```
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);
```

Character references will not be substituted in option lists, but they will be recognized in options with the *Unichar* data type (without the '&' and ';' decoration). This recognition will always be enabled; it is not subject to the *charref* parameter or option.

When an & character is found in the text which does not introduce a numerical reference, character reference, or glyph name reference, an exception will be thrown if *glyphcheck=error*. In other words, by setting *glyphcheck=none* you can use both character or glyph name references and isolated & characters in the same text.

4.6.3 Glyph Checking and Substitution

Content strings will be visualized on the page with a particular font. However, no single font contains all characters contained in the latest Unicode standard. While obtaining suitable fonts is obviously a task of the PDFlib user, PDFlib tries to work around some common problems by substituting certain characters with visually similar glyphs if the original glyph is not available in the font. This process can be controlled in detail by the *glyphcheck* and *errorpolicy* parameters and options, as well as the *replacementchar* option of *PDF_load_font( ).*

The *glyphcheck* parameter and option provides control for situations where the selected font does not contain any glyph for a code contained in a content string for text output:

- *glyphcheck=none:* fast, but unsafe; text output may contain the missing glyph symbol (often a hollow or crossed rectangle);
- *glyphcheck=replace:* silent approach which creates the most appropriate text output (see below).
- *glyphcheck=error:* safest approach; an exception will be thrown to alert the user of the problem. However, the output document will no longer be usable because of the exception.

The detailed behavior of *glyphcheck* depends on the type of encoding, and is described in Table 4.9.

Glyph replacement. If *glyphcheck=replace,* unavailable glyphs will recursively be replaced as follows:

- Select a similar glyph according to the Unicode value from PDFlib's internal replacement table. The following (incomplete) list contains some of these glyph mappings. If the first character in the list is unavailable in a font, it will automatically be replaced with the second:

Table 4.9 Glyph checking details for various encodings

glyphcheck	*8-bit encodings*	*builtin*	*glyphid*	*unicode*	*Unicode CMaps*	*other CMaps*
none	*unknown code or Unicode value will be replaced with 0*		*invalid glyph id is replaced with 0*	*invalid Unicode value is replaced with 0*	*unknown Unicode value is replaced with 0*	*invalid code sequence triggers an exception*
replace	*see below*	*invalid code is replaced with 0 or* replacementchar	*same as* none	*see below*	*same as* none	*same as* none
error	*The API functions will throw an exception if an error occurs. A detailed error message can be queried with PDF_get_errmsg( ) even if the function does not return -1.*					

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 with U+0066 U+0066)
- Select glyphs with the same Unicode semantics according to their glyph name (e.g. replace *A.swash* with *A*, replace *f_f_i* with the sequence *f, f, i*

If no replacement was found, the character specified in the *replacementchar* option will be used. If the corresponding glyph itself is not available in the font, U+00A0 (NO-BREAK SPACE), U+0020 (SPACE) will be tried; If these are still unavailable, U+0000 (missing glyph symbol) will be used.

4.6.4 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 (assuming the font has been successfully loaded with *PDF_load_font( )* earlier. Note that the *unicode* option expects a Unichar, and glyph name references can be used for Unichars without the *&* and *;* decoration. This is safer than checking the glyph name which may be *euro*, *Euro*, or something different:

```
if (p.info_font(font, "code", "unicode=euro") == -1)
{
        /* no glyph for Euro sign available in the font */
}
```

Alternatively, you can call *PDF_info_textline( )* to check the number of unmapped glyphs 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 character (which is expressed with a glyph name reference). If one unmapped glyph is found this means that the font does not contain a glyph for the Euro sign:

```
String optlist = "font=" + font + " charref";

if (p.info_textline(font, "&euro;", "unmappedglyphs", optlist) == 1)
{
        /* no glyph for Euro sign available in the font */
}
```

5 Font Handling

5.1 Overview of Fonts and Encodings

Font handling is one of the most complex aspects of document formats. In this section we will summarize PDFlib's main characteristics with regard to font handling.

5.1.1 Supported Font Formats

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

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

TrueType fonts. PDFlib supports vector-based TrueType fonts, but not those based on bitmaps. The TrueType font file must be supplied in Windows TTF or TTC format, or must be installed in the Mac or Windows operating system.

OpenType fonts. OpenType is a modern font format which combines PostScript and TrueType technology, and uses a platform-independent file format. There are two flavors of OpenType fonts, both of which are supported by PDFlib:

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

Contrary to PostScript Type 1 fonts, TrueType and OpenType fonts do not require any additional metrics file since the metrics information is included in the font file itself.

Chinese, Japanese, and Korean (CJK) fonts. In addition to Acrobat's standard CJK fonts (see Section 5.6, »Chinese, Japanese, and Korean Fonts«, page 116), PDFlib supports custom CJK fonts in the TrueType and OpenType formats. Generally these fonts are treated similarly to Western fonts.

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

- bitmap fonts,
- custom graphics, such as logos can easily be printed using simple text operators,
- Japanese gaiji (user-defined characters) which are not available in any predefined font or encoding.

5.1.2 Font Encodings

All fonts for text on a page must be used with a suitable encoding. The encoding defines how the actual bytes in a string will be interpreted by PDFlib and Acrobat, and how they translate into text on a page. PDFlib supports a variety of encoding methods.

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

Note Not all encodings can be used with a given font. The user is responsible for making sure that the font contains all characters required by a particular encoding. This can even be problematic with Acrobat's core fonts (see Table 4.4).

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

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

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

8-Bit encodings. 8-bit encodings are discussed in detail in Section 4.4, »8-Bit Encodings«, page 81. They can be pulled from various sources:

- A large number of predefined encodings according to Table 4.4. These cover the most important encodings currently in use on a variety of systems, and in a variety of locales.
- User-defined encodings which can be supplied in an external file or constructed dynamically at runtime with *PDF_encoding_set_char( )*. These encodings can be based on glyph names or Unicode values.
- Encodings pulled from the operating system, also known as *system encoding*. This feature is only available on Windows, IBM eServer iSeries, and zSeries.
- Abbreviated Unicode-based encodings which can be used to conveniently address any Unicode range of 256 consecutive characters with 8-bit values.
- Encodings specific to a particular font. These are also called *font-specific* or *builtin* encodings.

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

- Purely Unicode-based addressing via the *unicode* encoding keyword. In this case the client directly supplies Unicode strings to PDFlib. The Unicode strings may be formatted according to one of several standard methods (such as UTF-16, UTF-8) and byte orderings (little-endian or big-endian).
- CMap-based addressing for a variety of Chinese, Japanese, and Korean standards. PDFlib supports all CMaps supported by Acrobat (see Section 5.6, »Chinese, Japanese, and Korean Fonts«, page 116).
- Glyph id addressing for TrueType and OpenType fonts via the *glyphid* encoding keyword. This is useful for advanced text processing applications which need access to individual glyphs in a font without reference to any particular encoding scheme, or must address glyphs which do not have any Unicode mapping. The number of valid glyph ids in a font can be queried with the *maxcode* keyword in *PDF_info_font( )*.
- Direct CID addressing: this is mainly useful for creating CJK character collection tables.

5.2 Font Format Details

5.2.1 PostScript Type 1 Fonts

PostScript font 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. While AFM-based font metrics can be rearranged to any encoding supported by the font, common PFM metrics files for Western fonts (code page 1252) can only be used with the following encodings: *auto, winansi, iso8859-1, unicode, ebcdic*. PFM files for symbol fonts can be used with *builtin* encoding. PFM files for other code pages can be used with an encoding which matches the code page in the PFM (or any subset thereof), or with encoding *builtin* to choose the PFM's internal code page, or with *unicode*. For example, a PFM for a cyrillic font can be used with the encodings *cp1250, builtin,* and *unicode*. Encoding *auto* will select an appropriate encoding automatically.
- 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). PostScript host fonts can be used with the followings encodings: *auto, macroman, macroman_apple, unicode,* and *builtin*. However, *macroman* and *macroman_apple* may not be accepted for some fonts subject to the glyph complement of the font.
  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. Note that PostScript host fonts are not supported in Carbon-less (Classic) builds of PDFlib.
- OpenType fonts with PostScript outlines *(*.otf)*.

PostScript font names. If you are working with font files you can use arbitrary alias names (see Section , »Sources of Font Data«, page 101). 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.
- If you are working with Windows 2000/XP or Mac OS X 10.4 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). Also, the font name as given in any Windows .inf file is not relevant for use with PDF.

PostScript glyph names. In order to write a custom encoding file or find fonts which can be used with one of the supplied encodings you will have to find information about the exact definition of the character set to be defined by the encoding, as well as the glyph names used in the font files. You must also ensure that a chosen font provides all necessary characters for the encoding. If you happen to have the FontLab[1] font editor (by the way, a great tool for dealing with all kinds of font and encoding issues), you may use it to find out about the encodings supported by a given font (look for »code pages« in the FontLab documentation).[2]

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

In order to address glyphs in a font by their name you can use PDFlib's syntax for glyph names, see Section 4.6.2, »Character References and Glyph Name References«, page 89.

5.2.2 TrueType and OpenType Fonts

TrueType and OpenType file formats. TT and OT font files are self-contained: they contain all required information in a single file. PDFlib supports the following file formats for TrueType and OpenType fonts:

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

TrueType and OpenType font names. If you are working with font files you can use arbitrary alias names (see Section , »Sources of Font Data«, page 101). 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).*

5.2.3 User-Defined (Type 3) Fonts

Type 3 fonts in PDF (as opposed to PostScript Type 3 fonts) are not actually a file format. Instead, the glyphs in a Type 3 font must be defined at runtime with standard PDFlib graphics functions. Since all PDFlib features for vector graphics, raster images, and even text output can be used in Type 3 font definitions, there are no restrictions regarding

[1] See www.fontlab.com

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

the contents of the characters in a Type 3 font. Combined with the PDF import library PDI you can even import complex drawings as a PDF page, and use those for defining a character in a Type 3 font.

Note PostScript Type 3 fonts are not supported.

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

```
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();
```

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 *PDF_show()* or a similar function, and the encoding contains *glyphname* at position *x,* then *glyphname* must have been defined via *PDF_begin_glyph()*. This restriction affects only Type 3 fonts; missing glyphs in PostScript Type 1, TrueType, or OpenType fonts will simply be ignored.
- Some PDF consumers (this is not true for Acrobat) require a glyph named *.notdef* if codes will be used for which the corresponding glyph names are not defined in the font. The *.notdef* glyph must be present, but it may simply contain an empty glyph description.
- When normal bitmap data is used to define characters, unused pixels in the bitmap will print as white, regardless of the background. In order to avoid this and have the original background color shine through, use the *mask* parameter for constructing the bitmap image.
- The *interpolate* option for images may be useful for enhancing the screen and print appearance of Type 3 bitmap fonts.
- 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()*.

5.3 Locating, Embedding and Subsetting Fonts

5.3.1 Searching for Fonts

Sources of Font Data. PDFlib loads fonts with the function *PDF_load_font()* or appropriate font options in *PDF_fit_textline()* and *PDF_add/create_textflow()*. You can use a font's native name, or work with arbitrary custom font names which will be used to locate the font data. Custom font names should 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; otherwise a new font handle will be created for the same font name. PDFlib supports the following sources of font data:

- Disk-based font files
- Fonts pulled from the Windows or Mac operating system (host fonts)
- PDF standard fonts: these are from a small set of Latin and CJK fonts with well-known names
- Type 3 fonts which have been defined with *PDF_begin_font()* and related functions
- Font data passed by the client directly in memory by means of a PDFlib virtual file (PVF). This is useful for advanced applications which have the font data already loaded into memory and want to avoid unnecessary disk access by PDFlib (see Section 3.1.2, »The PDFlib Virtual File System (PVF)«, page 45 for details on virtual files).

The font name supplied to PDFlib is a name string. The choice of font name depends on the method for locating the font data. PDFlib searches for fonts according to the steps described below (in the specified order).

Standard CJK fonts. Acrobat supports various standard fonts for CJK text; see Section 5.6.1, »Standard CJK Fonts«, page 116, for more details and a list of standard CJK fonts. If the requested font name matches one of the standard CJK fonts the font will be selected, for example:

```
font = p.load_font("KozGoPro-Medium", "90msp-RKSJ-H", "");
```

Type 3 fonts. Type 3 fonts must be defined at runtime by defining its glyphs with standard PDFlib graphics functions (see Section 5.2.3, »User-Defined (Type 3) Fonts«, page 99). If the font name supplied to *PDF_begin_font()* matches the font name requested with *PDF_load_font()* the font will be selected, for example:

```
font = p.load_font("PDFlibLogoFont", "logoencoding", "");
```

Font outline files. The font name is related to the name of a disk-based or virtual font outline file via the *FontOutline* resource, for example:

```
p.set_parameter("FontOutline", "f1=/usr/fonts/DFHSMincho-W3.ttf");
font = font = p.load_font("f1", "unicode", "");
```

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 Searching«, page 46). In order to avoid absolute file names you can use the *Search-*

Path 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=DFHSMincho-W3.ttf");
font = font = p.load_font("f1", "unicode", "");
```

For PostScript fonts the corresponding resource configuration must relate the font metrics and outline data (the latter only if embedding is requested, see Section 5.3.3, »Font Embedding«, page 105) to the corresponding disk file(s):

```
p.set_parameter("FontOutline", "f1=LuciduxSans.pfa");
p.set_parameter("FontPFM", "f1=LuciduxSans.pfm");
font = font = p.load_font("f1", "unicode", "embedding");
```

In order to select a font which is contained in a TrueType Collection (TTC, see Section 5.6.2, »Custom CJK Fonts«, page 117) file you directly specify its name:

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

The font name can be encoded in ASCII or Unicode, and will be matched against all 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:

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

PostScript font metric files. The font name is related to the name of a disk-based or virtual PostScript font metric file via the *FontAFM* or *FontPFM* resource. This is sufficient if font embedding is not required, for example:

```
p.set_parameter("FontOutline", "f2=carta.afm");
font = p.load_font("f2", "builtin", "");
```

Host font aliases. The font name is related to the name of a host font via the *HostFont* resource. For example, to replace one of the Latin core fonts (see below) with a host font installed on the system you must configure the font in the *HostFont* resource category. The following line 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", "builtin", "embedding");
```

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

The following code fragment requests one of the core fonts without any configuration:

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

Host fonts. If the font name matches the name of a system font (also known as a host font) on Windows or Mac it will be selected. See Section 5.3.2, »Host Fonts on Windows and Mac«, page 103, 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", "");
```

Host font names can be encoded in ASCII. On Windows Unicode can also be used.

Extension-based search for font files. 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):

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

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

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

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

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

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", "");
```

5.3.2 Host Fonts on Windows and Mac

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 5.2.1, »PostScript Type 1 Fonts«, page 98, and Section 5.2.2, »TrueType and OpenType Fonts«, page 99.

Finding host font names on Windows. You can easily find the name of an installed font by double-clicking the font file, and taking note of the full font name which will be displayed in the first line of the resulting window. 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)*.

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.

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", "");
```

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.*

[1] *See www.microsoft.com/typography/TrueTypeProperty21.mspx*

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.

Finding 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. However, in some cases Font Book does not display the proper QuickDraw name of a font which is required by PDFlib.

For this reason we recommend Apple's freely available Font Tools[1]. This suite of command-line utilities contains a program called *ftxinstalledfonts* which is useful for determining the exact QuickDraw name of all installed fonts. In order to determine the font name expected by PDFlib, install Font Tools and issue the following statement in a terminal window:

```
ftxinstalledfonts -q
```

Potential problem 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.

5.3.3 Font Embedding

PDFlib is capable of embedding font outlines into the generated PDF output. Font embedding is controlled via the *embedding* option of *PDF_load_font( )* (although in some cases PDFlib will enforce font embedding):

```
font = p.load_font("WarnockPro", "winansi", "embedding");
```

Alternatively, a font descriptor containing only the character metrics and some general information about the font (without the actual glyph outlines) can be embedded. If a font is not embedded in a PDF document, Acrobat will take it from the target system if available, or construct a substitute font according to the font descriptor. Table 5.1 lists different situations with respect to font usage, each of which poses different requirements on the font and metrics files required by PDFlib. In addition to the requirements listed in Table 5.1 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.

When a font with font-specific encoding (a symbol font) or one containing glyphs outside Adobe's Standard Latin character set is used, but not embedded in the PDF output, the resulting PDF will be unusable unless the font is already natively installed on the target system (since Acrobat can only simulate Latin text fonts). Such PDF files are

[1] *See developer.apple.com/textfonts/download*

inherently nonportable, although they may be of use in controlled environments, such as intra-corporate document exchange.

Table 5.1 Different font usage situations and required files

font usage	***font metrics file must be available?***	***font outline file must be available?***
one of the 14 core fonts	*no*	*only if embedding is desired*
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 is desired*
TrueType fonts	*n/a*	*yes*
OpenType fonts, incl. CJK TrueType and OpenType fonts	*n/a*	*yes*
standard CJK fonts[1]	*no*	*no*

[1] See Section 5.6, »Chinese, Japanese, and Korean Fonts«, page 116, for more information on CJK fonts.

Forced font embedding. PDF requires font embedding for certain combinations of font and encoding. PDFlib will therefore force font embedding (and ignore the *embedding* option) in several cases (see *PDF_load_font( )).* The requirement for font embedding is caused by the internal conversion to a CID font, which (for 8-bit encodings) can be disabled by setting the *autocidfont* parameter to *false*. Doing so will also disable forced embedding. Note that in this case not all Latin characters will be accessible, and characters outside the Adobe Glyph List (AGL) won't work at all.

Legal aspects of font embedding. It's important to note that mere possession of a font file may not justify embedding the font in PDF, even for holders of a legal font license. Many font vendors restrict embedding of their fonts. Some type foundries completely forbid PDF font embedding, others offer special online or embedding licenses for their fonts, while still others allow font embedding provided subsetting is applied to the font. Please check the legal implications of font embedding before attempting to embed fonts with PDFlib. PDFlib will honor embedding restrictions which may be specified in a TrueType or OpenType font. If the embedding flag in a TrueType font is set to *no embedding*[1], PDFlib will honor the font vendor's request, and reject any attempt at embedding the font.

5.3.4 Font Subsetting

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

- TrueType fonts
- OpenType fonts with PostScript or TrueType outlines
- Type 3 fonts (special handling required, see below)

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

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

- The default subsetting behavior is controlled by the *autosubsetting* parameter. If it is *true*, subsetting will be enabled for all fonts where subsetting is possible (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.
- If *autosubsetting=false*, but subsetting is desired for a particular font nevertheless, the *subsetting* option must be supplied to *PDF_load_font()*:

  ```
  font = p.load_font("WarnockPro", "winansi", "subsetting");
  ```

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

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

Type 3 font subsetting. Type 3 fonts must be defined and therefore embedded before they can be used in a document; on the other hand, subsetting only works when the font is embedded after creating all pages (since the required set of glyphs must be known). In order to avoid this potential deadlock, PDFlib supports the concept of 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, and defines only the widths of the glyphs. The second pass must be done after creating all text in this font, and defines the actual glyph outlines or bitmaps. At the second pass PDFlib will already know which glyphs are required in the document, and will only embed those glyph descriptions which are actually part of the font subset. The following code fragment demonstrates Type 3 font definition, use, and subsetting:

```
/* pass 1: create a widths-only font */
p.begin_font("T3font", 0.001, 0, 0, 0.001, 0, 0, "widthsonly");

p.begin_glyph("a", 1000, 0, 0, 1000, 1000);
p.end_glyph();
```

```
p.begin_glyph("b", 400, 0, 0, 400, 400);
p.end_glyph();

/* ...define all glyph widths...*/

p.end_font();

/* use the font in the document */
p.begin_page_ext(595.0, 842.0, "");
font = p.load_font("T3font", "winansi", "subsetting");
p.setfont(font, 24);
p.set_text_pos(50, 700);
p.show("a");
p.end_page_ext("");

/* pass 2: supply glyph descriptions for the font */
p.begin_font("T3font", 0, 0.001, 0, 0, 0.001, 0, 0, "");

p.begin_glyph("a", 1000, 0, 0, 1000, 1000);
p.arc(500, 500, 500, 0, 360);
p.fill();
p.end_glyph();

p.begin_glyph("b", 400, 0, 0, 400, 400);
p.arc(200, 200, 200, 0, 360);
p.stroke();
p.end_glyph();

/* ...define all glyph descriptions... */

p.end_font();

p.end_document("");
```

Please note the following when working with Type 3 font subsets:

- The exact same set of glyphs must be provided in pass 1 and pass 2. It is strongly recommended to supply the *.notdef* glyph as the first glyph in both glyph sets.
- In the first pass *(widthsonly=true)* all font and glyph metrics (i.e. the font matrix in *PDF_begin_font( )* and *wx* and the glyph bounding box in *PDF_begin_glyph( ))* must be supplied and must accurately describe the actual glyphs; in the second pass *(widthsonly=false)* the font and glyph metrics will be ignored.
- A Type 3 font with subsetting can only be loaded once with *PDF_load_font( )*.
- If subsetting is performed, a *.notdef* glyph must be defined as the first glyph of the font.

5.4 Miscellaneous Topics

5.4.1 Symbol Fonts and Font-specific Encodings

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

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

```
EncodingScheme FontSpecific
```

Text fonts can be reencoded (adjusted to a certain code page or character set), while symbolic fonts can't, and must use *builtin* encoding instead.

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

Nevertheless, all symbolic fonts can be used with encoding *unicode* in PDFlib. In this situation PDFlib will assign Unicode values from the Private Use Area (PUA). Although these PUA values are not available for clients, *unicode* encoding for symbol fonts allows the use of character references (see Section 4.6.2, »Character References and Glyph Name References«, page 89) with font-specific glyph names. This is a big advantage since it allows to select symbol glyphs based on their names, without getting bogged down in Unicode/encoding problems.

As an exception, the widely used *Symbol* and *ZapfDingbats* fonts have standardized Unicode values (outside of the PUA). If they are loaded with *unicode* encoding the glyphs can be addressed with the Unicode values U+2700 and up.

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

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

5.4.2 Glyph ID Addressing for TrueType and OpenType Fonts

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

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

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

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

5.4.3 The Euro Glyph

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

- With *unicode* encoding use the character U+20AC. Alternatively, you can address the Euro glyph with the corresponding character reference *&euro;* (by name) or *&#x20AC;* (by numerical value).
- In *winansi* encoding the location is 0x80 (hexadecimal) or 128 (decimal).
- The common *iso8859-1* encoding does not contain the Euro character. However, the *iso8859-15* encoding is an extension of *iso8859-1* which adds the Euro character at 0xA4 (hexadecimal) or 164 (decimal).
- The original *macroman* encoding does not contain the Euro character. However, Apple modified this encoding and replaced the old currency glyph which the Euro glyph at 0xDB (hexadecimal) or 219 (decimal). In order to use this modified Mac encoding use *macroman_apple* instead of *macroman*.

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

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

5.4.4 Unicode-compatible Fonts

Precise Unicode semantics are important for PDFlib's internal processing, and crucial for properly extracting text from a PDF document, or otherwise reusing the document, e.g., converting the contents to another format. This is especially important when creating Tagged PDF which has strict requirements regarding Unicode compliance (see Section 9.6.1, »Generating Tagged PDF with PDFlib«, page 214). In addition to Tagged PDF Unicode compatibility is relevant for the Textflow feature.

Unicode-compatible fonts. A font – more precisely: a combination of font and encoding – is considered Unicode-compatible if the encoding used for loading the font complies to all of the following conditions:

- The encoding *builtin* is only allowed for the *Symbol* and *ZapfDingbats* fonts and PostScript-based OpenType fonts.
- The encoding is not *glyphid*.
- If the encoding is one of the predefined CMaps in Table 4.5 it must be one of the UCS2 or UTF16 CMaps.

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

- All fonts used in the document must be Unicode-compatible as defined above, or use one of the predefined CMaps in Table 4.5.
- If the encoding has been constructed with *PDF_encoding_set_char( )* and glyph names without corresponding Unicode values, or loaded from an encoding file, all glyph names must be contained in the Adobe Glyph List or the list of well-known glyph names in the Symbol font.
- The *unicodemap* parameter or option is *true*.
- All text strings must have clearly defined semantics according to the Unicode standard, i.e. characters from the Private Use Area (PUA) are not allowed.
- PDF pages imported with PDI must be Unicode-compatible. PDI does not change the Unicode compatibility status of imported pages: it will neither remove nor add Unicode information.

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

5.5 Font Metrics and Text Variations

5.5.1 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.

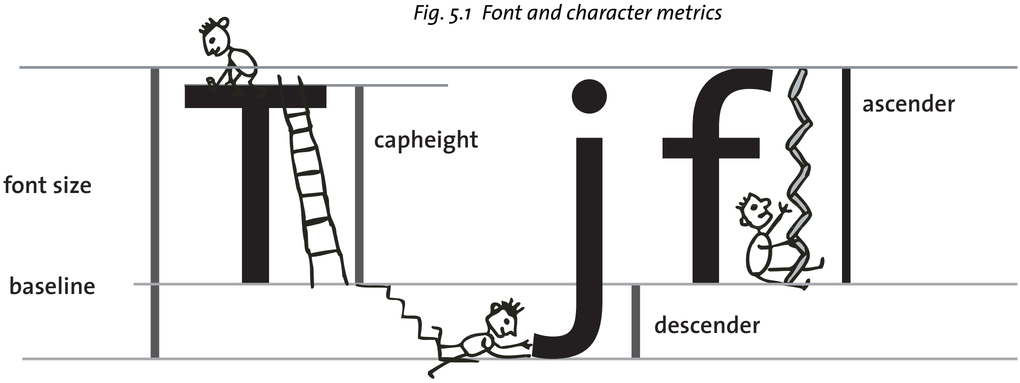
The *leading* (line spacing) specifies the vertical distance between the baselines of adjacent lines of text. By default it is set to the value of the font size. The *capheight* is the height of capital letters such as *T* or *H* in most Latin fonts. The *xheight* is the height of lowercase letters such 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. 5.1 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.

CPI calculations. While most fonts have varying character widths, so-called mono-spaced fonts use the same widths for all characters. In order to relate PDF font metrics to the characters per inch (CPI) measurements often used in high-speed print environments, some calculation examples for the mono-spaced Courier font may be helpful. In Courier, all characters have a width of 600 units with respect to the full character cell of 1000 units per point (this value can be retrieved from the corresponding AFM metrics file). For example, with 12 point text all characters will have an absolute width of

```
12 points * 600/1000 = 7.2 points
```

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

5.5.2 Kerning

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

- By default, kerning information in a font is not read when loading a font. If kerning is desired the *kerning* option must be set in the respective call to *PDF_load_font( )*. This instructs PDFlib to read the font's kerning data (if available).

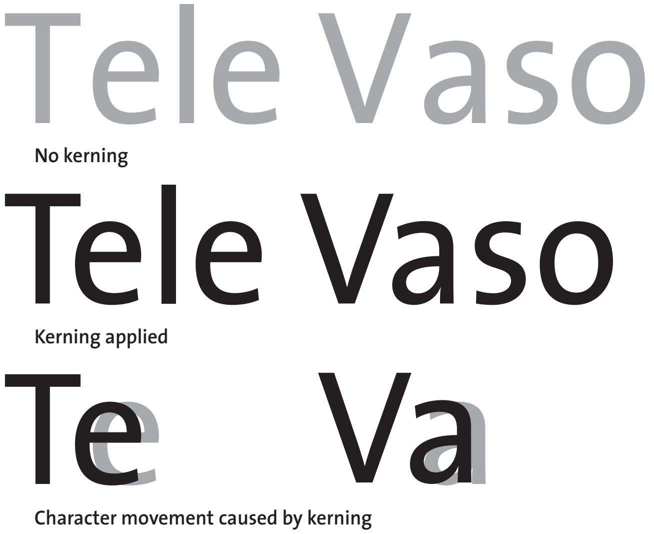


Fig. 5.2 Kerning

- When a font for which kerning data has been read is used with any text output function, the positional corrections provided by the kerning data will be applied. However, kerning can also be disabled by setting the *kerning* parameter to *false*:

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

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

Kerning is applied in addition to any character spacing, word spacing, and horizontal scaling which may be activated. PDFlib does not have any limit for the number of kerning pairs in a font.

5.5.3 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 feature for fonts other than the standard CJK fonts is not recommended. Also note that the fontstyle feature 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 Section 5.3.3, »Font Embedding«, page 105). Font styles can not be applied to TrueType Collections (TTC).
- The encoding is *winansi, macroman,* or one of the predefined CJK CMaps listed in Table 4.5 (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.

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.

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 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:

```
font = p.load_font("Doxy", "italicangle=-12");          /* create fake italic font */
```

Note *The italicangle parameter is not supported for vertical writing mode.*

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 *underlineposition* and *underlinewidth* parameters and options can be used for fine-tuning.

The current stroke color is used for drawing the bars. The current linecap and dash parameters are ignored, however. Aesthetics alert: in most fonts underlining will touch descenders, and overlining will touch diacritical marks atop ascenders.

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 function 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.

Note *Text rendering mode 7 (use text as clipping path) will not have any effect when creating text output with PDF_fit_textline( ) or PDF_fit_textflow( ).*

Text color. Text will usually be display in the current fill color, which can be set using *PDF_setcolor( )*. However, if a rendering mode other than 0 has been selected, both stroke and fill color may affect the text depending on the selected rendering mode.

5.6 Chinese, Japanese, and Korean Fonts

5.6.1 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 5.2 along with applicable CMaps (see Section 4.5, »Encodings for Chinese, Japanese, and Korean Text«, page 85, for more details on CJK CMaps).

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 5.5.3, »Text Variations«, page 114).*

Table 5.2 Acrobat's standard fonts and CMaps (encodings) for Japanese, Chinese, and Korean text

locale	***font name***	***sample***	***supported CMaps (encodings)***
Simplified Chinese	*STSong-Light*[1] *STSongStd-Light-Acro*[2] *AdobeSongStd-Light-Acro*[3] *AdobeSongStd-Light*[6]	国际 国际	*GB-EUC-H, GB-EUC-V, GBpc-EUC-H, GBpc-EUC-V, GBK-EUC-H, GBK-EUC-V, GBKp-EUC-H*[4]*, GBKp-EUC-V*[2]*, GBK2K-H*[2]*, GBK2K-V*[2]*, UniGB-UCS2-H, UniGB-UCS2-V, UniGB-UTF16-H*[5]*, UniGB-UTF16-V*[5]
Traditional Chinese	*MHei-Medium*[1] *MSung-Light*[1] *MSungStd-Light-Acro*[2] *AdobeMingStd-Light-Acro*[3] *AdobeMingStd-Light*[6]	中文 中文 中文	*B5pc-H, B5pc-V, HKscs-B5-H*[4]*, HKscs-B5-V*[4]*, 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*[5]*, UniCNS-UTF16-V*[5]
Japanese	*HeiseiKakuGo-W5*[1] *HeiseiMin-W3*[1] *KozMinPro-Regular-Acro*[2, 7] *KozGoPro-Medium-Acro*[3, 7] *KozGoPro-Medium*[6] *KozMinProVI-Regular*[6]	日本語 日本語 日本語	*83pv-RKSJ-H, 90ms-RKSJ-H, 90ms-RKSJ-V, 90msp-RKSJ-H, 90msp-RKSJ-V, 90pv-RKSJ-H, Add-RKSJ-H, Add-RKSJ-V, EUC-H, EUC-V, Ext-RKSJ-H, Ext-RKSJ-V, H, V, UniJIS-UCS2-H, UniJIS-UCS2-V, UniJIS-UCS2-HW-H*[7]*, UniJIS-UCS2-HW-V*[7]*, UniJIS-UTF16-H*[5]*, UniJIS-UTF16-V*[5]
Korean	*HYGoThic-Medium*[1] *HYSMyeongJo-Medium*[1] *HYSMyeongJoStd-Medium-Acro*[2] *AdobeMyungjoStd-Medium-Acro*[3] *AdobeMyungjoStd-Medium*[6]	한국 한국 한국	*KSC-EUC-H, KSC-EUC-V, KSCms-UHC-H, KSCms-UHC-V, KSCms-UHC-HW-H, KSCms-UHC-HW-V, KSCpc-EUC-H, UniKS-UCS2-H, UniKS-UCS2-V, UniKS-UTF16-H*[5]*, UniKS-UTF16-V*[5]

1. Available in Acrobat 4; Acrobat 5 and 6 will substitute these with different fonts.
2. Available in Acrobat 5 only
3. Available in Acrobat 6 only
4. Only available when generating PDF 1.4 or above
5. Only available when generating PDF 1.5 or above
6. Only available when generating PDF 1.6 or above
7. 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.

Horizontal and vertical writing mode. PDFlib supports both horizontal and vertical writing modes. 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* option to *PDF_load_font( )*.

Note Some PDFlib functions change their semantics according to the writing mode. For example, PDF_continue_text( ) should not be used in vertical writing mode, and the character spacing must be negative in order to spread characters apart in vertical writing mode.

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 5.2), and that Unicode-aware language bindings support only UCS2-compatible CMaps. The *KozMinPro-Regular-Acro* sample in Table 5.2 can been 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 Shift-JIS-compatible CMap *(Ext-RKSJ)* and horizontal writing mode *(H)*. The *fontname* parameter must be the exact name of the font without any encoding or writing mode suffixes. The *encoding* parameter is the name of one of the supported CMaps (the choice depends on the font) and will also indicate the writing mode (see above). PDFlib supports all of Acrobat's default CMaps, and will complain when it detects a mismatch between the requested font and the CMap. For example, PDFlib will reject a request to use a Korean font with a Japanese encoding.

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.

5.6.2 Custom CJK Fonts

In addition to Acrobat's standard CJK fonts PDFlib supports custom CJK fonts (fonts outside the list in Table 5.2) in the TrueType (including TrueType Collections, TTC) and OpenType formats. A custom CJK font 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.

- By default, font subsetting will be applied for embedded custom CJK fonts unless they are used with a standard CMap and *keepnative=false*. This can be controlled with various parameters, see Section 5.3, »Locating, Embedding and Subsetting Fonts«, page 101.
- 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.

Note Windows EUDC fonts (end-user defined characters) are supported, but linking individual end-user defined characters into all fonts is not supported (see »End-user defined characters (EUDC)«, page 119).

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);
```

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 , »Sources of Font Data«, page 101):

```
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 Section , »Sources of Font Data«, page 101) 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 5.3.1, »Searching for Fonts«, page 101):

```
p.set_parameter("FontOutline", "msgothic=msgothic.ttc");
```

End-user defined characters (EUDC). PDFlib does not support linking end-user defined characters into fonts, but you can use the EUDC editor available in Windows to create custom characters for use with PDFlib. Proceed as follows:

- Use the *eudcedit.exe* to create one or more custom characters at the desired Unicode position(s).
- Locate the *EUDC.TTE* file in the directory *\Windows\fonts* and copy it to some other directory. 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 5.3.1, »Searching for Fonts«, page 101):

  ```
  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 the preceding step you can use the following function call to configure the font file directly from the Windows directory. This way you will always access the current EUDC font used in Windows:

  ```
  p.set_parameter("FontOutline", "EUDC=C:\Windows\fonts\EUDC.TTE");
  ```

- Use the following call to load the font:

  ```
  font = p.load_font("EUDC", "unicode", "");
  ```

  and supply the Unicode character codes chosen in the first step to output the characters.

6 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 159.

6.1 Importing Raster Images

6.1.1 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 argument to the *PDF_fit_image()* function is an option list which supports a variety of options for positioning, scaling, and rotating the image. Details regarding these options are discussed in Section 7.3, »Placing Images and Imported PDF Pages«, page 159.

Re-using image data. PDFlib supports an important PDF optimization technique for using repeated raster images. Consider a layout with a constant logo or background on multiple pages. In this situation it is possible to include the actual image data only once in the PDF, and generate only a reference on each of the pages where the image is used. Simply load the image file once, and call *PDF_fit_image()* every time you want to place the logo or background on a particular page. You can place the image on multiple pages, or use different scaling factors for different occurrences of the same image (as long as the image hasn't been closed). Depending on the image's size and the number of occurrences, this technique can result in enormous space savings.

Inline images. As opposed to reusable images, which are written to the PDF output as image XObjects, inline images are written directly into the respective content stream (page, pattern, template, or glyph description) . This results in some space savings, but should only be used for small amounts of image data (up to 4 KB) per a recommendation in the PDF reference. The primary use of inline images is for bitmap glyph descriptions in Type 3 fonts.

Inline images can be generated with the *PDF_load_image()* interface by supplying the *inline* option. Inline images cannot be reused, i.e., the corresponding handle must not be supplied to any call which accepts image handles. For this reason if the *inline* option has been provided *PDF_load_image( )* internally performs the equivalent of the following code:

```
p.fit_image(image, 0, 0, "");
p.close_image(image);
```

Scaling and dpi calculations. PDFlib never changes the number of pixels in an imported image. Scaling either blows up or shrinks image pixels, but doesn't do any downsampling (the number of pixels in an image will always remain the same). A scaling factor of 1 results in a pixel size of 1 unit in user coordinates. In other words, the image will be imported with its native resolution (or 72 dpi if it doesn't contain any resolution information) if the user coordinate system hasn't been scaled (since there are 72 default units to an inch).

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 conversions is required it must be applied to the image data before loading the image in PDFlib.

6.1.2 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. PNG images which make use of interlacing or contain an alpha channel (which will be lost anyway, see below) will have to be uncompressed, which takes significantly longer than pass-through mode. If a PNG image contains transparency information, the transparency is retained in the generated PDF (see Section 6.1.4, »Image Masks and Transparency«, page 125). However, alpha channels are not supported by PDFlib.

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 critical types of JPEG images, but transcoding can also be controlled via the *passthrough* option of *PDF_load_image( )*. 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.

Note PDFlib does not interpret color space or resolution information from JPEG images in the SPIFF or Exif formats.

JPEG2000 images. JPEG2000 images (ISO 15444-2) require PDF 1.5 or above, and are always handled in pass-through mode. PDFlib supports JPEG2000 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 JPEG2000 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 JPEG2000 image, i.e. the *iccprofile* option must not be used. ICC profiles contained in the JPEG2000 image file will always be kept, i.e. the *honoriccprofile* option is always *true*.
- The *colorize* option is not supported for JPEG2000 images.

*Note Raw JPEG2000 code streams without JPX wrapper (often *.j2k) and JPM compound image files according to ISO 15444-6 (usually *.jpm) are not supported.*

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 recompressed with Flate compression.

TIFF images. PDFlib will handle most TIFF images in pass-through mode. PDFlib supports the following flavors of TIFF images:

- compression schemes: uncompressed, CCITT (group 3, group 4, and RLE), ZIP (=Flate), and PackBits (=RunLength) are handled in pass-through mode; other compression schemes, such as LZW and JPEG, are handled by uncompressing.
- color: black and white, grayscale, RGB, CMYK, CIELab, and YCbCr images; any alpha channel or mask which may be present in the file will be ignored.
- TIFF files containing more than one image (see Section 6.1.6, »Multi-Page Image Files«, page 128)
- Color depth must be 1, 2, 4, 8, or 16 bits per color sample. In PDF 1.5 mode 16 bit color depth will be retained in most cases with pass-through mode, but reduced to 8 bit for

certain image files (ZIP compression with little-endian/Intel byte order and 16-bit palette images).

Multi-strip TIFF images are converted to multiple images in the PDF file which will visually exactly represent the original image, but can be individually selected with Acrobat's TouchUp object tool. Multi-strip TIFF images can be converted to single-strip images with the *tiffcp* command line tool which is part of the TIFFlib package.[1] The ImageMagick[2] tool always writes single-strip TIFF images.

PDFlib fully interprets the orientation tag which specifies the desired image orientation in some TIFF files. PDFlib can be instructed to ignore the orientation tag (as many applications do) by setting the *ignoreorientation* option to true.

PDFlib will read clipping paths from TIFF images created with Adobe Photoshop.

Some TIFF features (e.g., spot color) and certain combinations of features (e.g., CMYK images with a mask) are not supported. Although TIFF images with JPEG compression are generally supported, some flavors of so-called old-style TIFF-JPEG will be rejected.

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 x 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 x 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.

6.1.3 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.

[1] *See www.libtiff.org*
[2] *See www.imagemagick.org*

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 6.1). In order to prevent the default clipping path from being applied set the *honorclippingpath* 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.

6.1.4 Image Masks and Transparency

Transparency in PDF. PDF supports various transparency features, all of which are implemented in PDFlib:

- Masking by position: an image may carry the intrinsic information »print the foreground or the background«. This is realized by a 1-bit mask image, and is often used in catalog images.
- Masking by color value: pixels of a certain color are not painted, but the previously painted part of the page shines through instead (»ignore all blue pixels in the image«). In TV and video technology this is also known as bluescreening, and is most often used for combining the weather man and the map into one image.
- PDF 1.4 introduced alpha channels or soft masks. These can be used to create a smooth transition between foreground and background, or to create semi-transparent objects (»blend the image with the background«). Soft masks are represented by 1-component images with 1-8 bit per pixel.

PDFlib supports three kinds of transparency information in images: implicit transparency, explicit transparency, and image masks.

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.

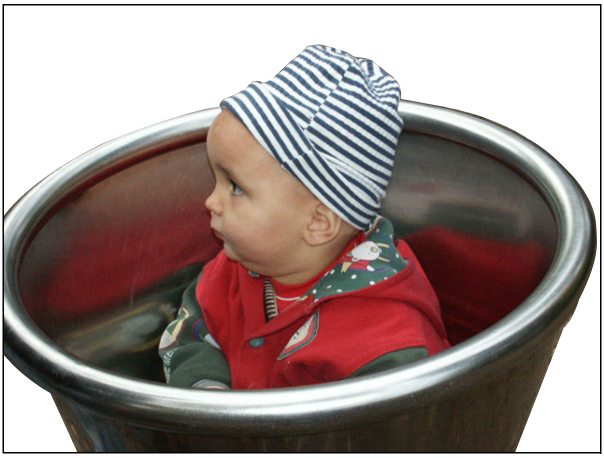


Fig. 6.1
Using a clipping path to separate foreground and background

Implicit transparency. In the implicit case, the transparency information from an external image file is respected, provided the image file format supports transparency or an alpha channel (this is not the case for all image file formats). Transparency information is detected in the following image file formats:

- GIF image files may contain a single transparent color value which is respected by PDFlib.
- PNG image files may contain several flavors of transparency information, or a full alpha channel. PDFlib will retain single transparent color values; if multiple color values with an attached alpha value are given, only the first one with an alpha value below 50 percent is used. A full alpha channel is ignored.

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 transparency mask. This is accomplished by opening the image with the *mask* option. In PDF 1.3, which supports only 1-bit masks, using this option is required; in PDF 1.4 it is optional. The following kinds of images can be used for constructing a mask:

- PNG images
- TIFF images (only single-strip)
- raw image data

Pixel values of 0 (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 for a transparency effect. In the second step the mask is applied to another image which itself is acquired through one of the image functions:

```
mask = p.load_image("png", maskfilename, "mask");
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, "");
```

Note the different use of the option list for *PDF_load_image()*: *mask* for defining a mask, and *masked* for applying a mask to another image.

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

Note PDFlib converts multi-strip TIFF images to multiple PDF images, which would be masked individually. Since this is usually not intended, this kind of images will be rejected both as a mask as well as a masked target image. Also, it is important to not mix the implicit and explicit cases, i.e., don't use images with transparent color values as mask.

Image masks. Image masks are images with a bit depth of 1 (bitmaps) in which 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 colorized with the current fill color. The following kinds of images can be used as image masks:

- PNG images
- TIFF images (single- or multi-strip)
- JPEG images (only as soft mask, see below)
- BMP; note that BMP images are oriented differently than other image types. For this reason BMP images must be reflected along the *x* axis before they can be used as a mask.
- raw image data

Image masks are simply opened with the *mask* option, and placed on the page after the desired fill color has been set:

```
mask = 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 6.1.5, »Colorizing Images«, page 127).

Soft masks. Soft masks generalize the concept of image masks to masks with more than 1 bit. They have been introduced in PDF 1.4 and blend the image against some existing background. PDFlib accepts all kinds of single-channel (grayscale) images as soft mask. They can be used the same way as image masks, provided the PDF output compatibility is at least PDF 1.4.

Ignoring transparency. Sometimes it is desirable to ignore any transparency information which may be contained in an image file. For example, Acrobat's anti-aliasing feature (also known as »smoothing«) isn't used for 1-bit images which contain black and transparent as their only colors. For this reason imported images with fine detail (e.g., rasterized text) may look ugly when the transparency information is retained in the generated PDF. In order to deal with this situation, PDFlib's automatic transparency support can be disabled with the *ignoremask* option when opening the file:

```
image = p.load_image("gif", filename, "ignoremask");
```

6.1.5 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 in the following formats:

- BMP
- PNG
- JPEG
- TIFF
- GIF

For images with an RGB palette, colorizing is only reasonable when the palette contains only gray values, and the palette index is identical to the gray value.

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("both", "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, "");
}
```

6.1.6 Multi-Page Image Files

PDFlib supports TIFF files which contain more than one image, also known as multi-page files. In order to use multi-page TIFFs, additional string and numerical parameters are used in the call to *PDF_load_image():*

```
image = p.load_image("tiff", filename, "page=2");
```

The *page* option indicates that a multi-image file is to be used. The last parameter specifies the number of the image to use. The first image is numbered 1. This option may be increased until *PDF_load_image()* returns -1, signalling that no more images are available in the file.

A code fragment similar to the following can be used to convert all images in a multi-image TIFF file to a multi-page PDF file:

```
for (frame = 1; /* */ ; frame++)
{
        String optlist = "page=" + frame;
        image = p.load_image("tiff", filename, optlist);
        if (image == -1)
            break;
        p.begin_page_ext(width, height, "");
        p.fit_image(image, 0.0, 0.0, "");
        p.close_image(image);
        p.end_page_ext("");
}
```

6.1.7 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( ).

6.2 Importing PDF Pages with PDI (PDF Import Library)

Note All functions described in this section require PDFlib+PDI. The PDF import library (PDI) is not contained in PDFlib or PDFlib Lite. Although PDI is integrated in all precompiled editions of PDFlib, a license key for PDI (or PPS, which includes PDI) is required.

6.2.1 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 Chapter 8, »The pCOS Interface«, page 181)

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 10, »Variable Data and Blocks«, page 223).

6.2.2 Using PDI Functions with PDFlib

General considerations. It is important to understand that PDI will only import the actual page contents, but not any 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.

You can not re-use individual elements of imported pages with other PDFlib functions. For example, re-using fonts from imported documents for some other content is not possible. Instead, all required fonts must be configured in PDFlib. If multiple imported documents contain embedded font data for the same font, PDI will not remove any duplicate font data. On the other hand, if fonts are missing from some imported

PDF, they will also be missing from the generated PDF output file. As an optimization you should keep the imported document open as long as possible in order to avoid the same fonts to be embedded multiple times in the output document.

PDI does not change the color of imported PDF documents in any way. For example, if a PDF contains ICC color profiles these will be retained in the output document.

PDFlib uses the template feature for placing imported PDF pages on the output page. Since some third-party PDF software does not correctly support the templates, restrictions in certain environments other than Acrobat may apply (see Section 3.2.4, »Templates«, page 56).

PDFlib-generated output which contains imported pages from other PDF documents can be processed with PDFlib+PDI again. However, due to restrictions in PostScript printing the nesting level should not exceed 10.

Code fragments for importing PDF pages. Dealing with pages from existing PDF documents is possible with a very simple code structure. The following code snippet opens a page from an existing document, and copies the page contents to a new page in the output PDF document (which must have been opened before):

```
int     doc, page, pageno = 1;
String  filename = "input.pdf";

if (p.begin_document(outfilename, "") == -1) {...}
...

doc = p.open_pdi_document(infilename, "");
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 argument to *PDF_fit_pdi_page( )* is an option list which supports a variety of options for positioning, scaling, and rotating the imported page. Details regarding these options are discussed in Section 7.3, »Placing Images and Imported PDF Pages«, page 159.

Dimensions of imported PDF pages. Imported PDF pages are regarded similarly to imported raster images, and can be placed on the output page using *PDF_fit_pdi_page( )*. By default, PDI will import the page exactly as it is displayed in Acrobat, in particular:

- cropping will be retained (in technical terms: if a CropBox is present, PDI favors the CropBox over the MediaBox; see Section 3.2.2, »Page Size«, page 54);
- rotation which has been applied to the page will be retained.

Alternatively, you can use the *pdiusebox* option to explicitly instruct PDI to use any of the MediaBox, CropBox, BleedBox, TrimBox or ArtBox entries of a page (if present) for determining the size of the imported page.

Imported PDF pages with layers. Acrobat 6 (PDF 1.5) introduced the layer functionality (technically known as *optional content)*. PDI will ignore any layer information which may be present in a file. All layers in the imported page, including invisible layers, will be visible in the generated output.

Imported PDF with OPI information. OPI information present in the input PDF will be retained in the output unmodified.

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 when 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.

6.2.3 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( )* will 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 can not be imported with PDI:

- PDF documents which use a higher PDF version number than the PDF output document that is currently being generated. The reason is that PDFlib can no longer make sure that the output will actually conform to the requested PDF version after a PDF with a higher version number has been imported. Solution: set the version of the output PDF to the required level using the *compatibility* option in *PDF_begin_document( )*.
- PDF documents with PDF/X or PDF/A conformance level which is not compatible to the PDF/A or PDF/X level of the current output document.

In addition, 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: PDF 1.6 documents created with the Distiller setting »Object Level Compression: Maximum«)

- Tagged PDF when the *tagged* option in *PDF_begin_document( )* is *true*.
- PDF/A or PDF/X documents which are not compatible to the PDF/A or PDF/X level of the current output document.

7 Formatting Features

7.1 Placing and Fitting Single-Line Text

The function *PDF_fit_textline( )* for placing a single line of text on a page offers a wealth of formatting options. The most important options will be discussed in this section using some common application examples. A complete description of these options can be found in the *PDFlib 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 159, 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 extended to the left and right or top and bottom using the *margin* option. The margin will be scaled along with the text line.

In the examples below, the coordinates of the reference point are supplied as *x, y* parameters of *PDF_fit_textline( )*.

7.1.1 Simple Text 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)*.

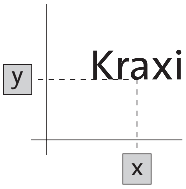
```
p.fit_textline(text, 30, 20, "position={center bottom}");
```

Figure 7.1 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.

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");
```

Fig. 7.1
Centered text



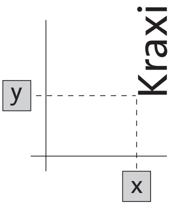


Fig. 7.2
Simple text with orientation west

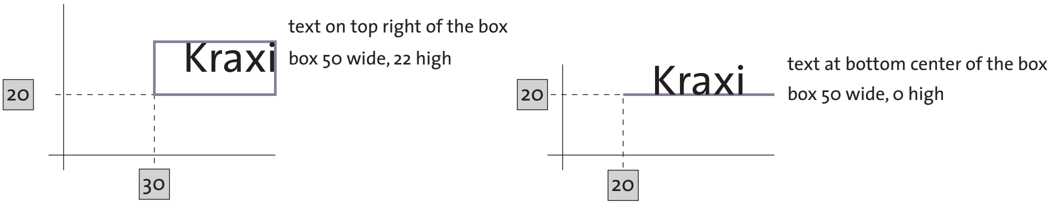

text on top right of the box
box 50 wide, 22 high
Kraxi
20
30
text at bottom center of the box
box 50 wide, 0 high
Kraxi
20
20

Fig. 7.3 Positioning text in a box

Figure 7.2 illustrates simple text placement with orientation.

7.1.2 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 7.3 illustrates the general behaviour.

Positioning text in the box. We define a box and place the text within the 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 7.4a, 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 7.4b.

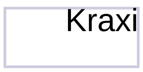
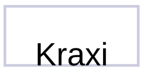
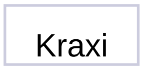
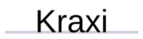
To achieve some distance between the text and the box we can add the *margin* option (see Figure 7.4c).

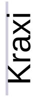
Note that the blue box or line depicted for visualizing the box size in the figures is not part of the actual output.

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 7.4d 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 7.4e.

Fig. 7.4 Placing text in a box subject to various positioning options

Generated output	*Option list for PDF_fit_textline( )*
a) 	`boxsize={50 22} position={right top}`
b) 	`boxsize={50 22} position={center bottom}`
c) 	`boxsize={50 22} position={center bottom} margin={0 3}`
d) 	`boxsize={50 0} position={center bottom}`

Generated output	*Option list for PDF_fit_textline( )*
e) 	`boxsize={0 35} position={left center} orientate=west"`

7.1.3 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 7.5a).

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 7.5b).

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 7.5c 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 7.5d shows the text being shrunk down to 75 percent of its original length.

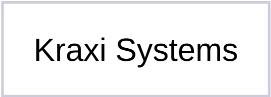
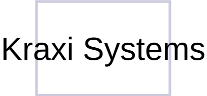
When decreasing the box width further down to 30 units the text will not fit even if shrinking is applied. Then the *meet* method will be applied. The *meet* method will decrease the font size until the text fits completely into the box. This case is shown in Figure 7.5e.

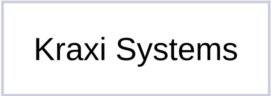
Fitting the text into the box with increased font size. You might want to fit the text so that it covers the whole width (or height) of the box but maintains its proportions. Using *fitmethod=meet* with a box larger than the text, the text will be increased until its width matches the box width. This case is illustrated in Figure 7.5f.

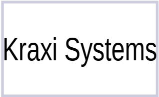
Completely fitting text into a box. We can further fit the text so that it completely fills the box. However, this combination will rarely be used since the text will most probably be distorted (see Figure 7.5g).

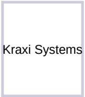
Fitting text into a box with clipping. In another rare case you might want to fit the text in its original size and clip the text if it exceeds the box. In this case, *fitmethod=clip* can be used. In Figure 7.5h the text is placed at the bottom left of a box which is not broad enough. The text will be clipped on the right.

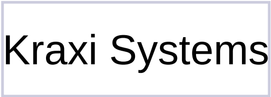
Fig. 7.5 Fitting text into a box on the page subject to various options

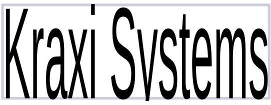
Generated output		*Option list for PDF_fit_textline( )*
a)		`boxsize={100 35} position=center fontsize=12`
b)		`boxsize={50 35} position=center fontsize=12`

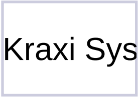
c)		`boxsize={100 35} position=center fontsize=12`
		`fitmethod=auto`

d)		`boxsize={58 35} position=center fontsize=12`
		`fitmethod=auto`

e)		`boxsize={30 35} position=center fontsize=12`
		`fitmethod=auto`

f)		`boxsize={100 35} position=center fontsize=12`
		`fitmethod=meet`

g)		`boxsize={100 35} position=center fontsize=12`
		`fitmethod=entire`

h)		`boxsize={50 35} position={left center}`
		`fontsize=12 fitmethod=clip`

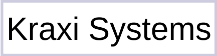
Horizontally centering text. The text height in *PDF_fit_textline( )* is the capheight, i.e. the height of the capital letter *H*, by default. If the text is positioned in the center of a box it will be vertically centered according to its capheight (see Figure 7.6a).

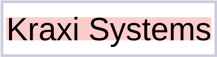
The *matchbox* option of *PDF_fit_textline( )* define the height of a textline which is the capheight of the given font size, by default. The height of the matchbox is calculated according to its *boxheight* suboption. The *boxheight* suboption determines the extent of the text above and below the baseline. *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.

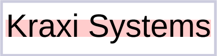
To illustrate the size of the matchbox we will fill it with red color (see Figure 7.6b). Figure 7.6c vertically centers the text according to the *xheight* by defining a matchbox with a corresponding box height.

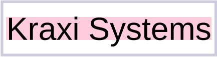
Figure 7.6d–f shows the matchbox (red) with various useful *boxheight* settings to determine the height of the text to be centered in the box (blue).

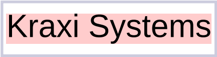
Fig. 7.6 Fitting text proportionally into a box according to different box heights

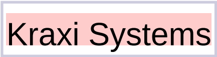
Generated output	***Option list for PDF_fit_textline( )***
a) 	`boxsize={80 20} position=center fitmethod=auto`

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}}`

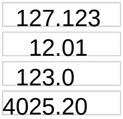
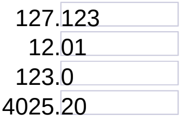
7.1.4 Aligning Text at a Character

Aligning text at a character. You might want to align text at a certain character, e.g. at the decimal point in a number. In the following code fragment the text is positioned at the center of the fitbox. Using the *alignchar*=. option the numbers are aligned at the dot character (see Figure 7.7a).

```
/* align floating point numbers at the dot character */
String  optlist =
        "font=" + normalfont + " fontsize=8 boxsize={70 8} " +
        "position={center bottom} alignchar=.";

p.fit_textline("127.123", 10, 20, optlist);
p.fit_textline("12.01", 10, 30, optlist);
p.fit_textline("123.0", 10, 40, optlist);
p.fit_textline("4025.20", 10, 50, optlist);
```

To align the dots at the reference point, you can omit the *position* option. In this case, the default *position={left bottom}* will be used which aligns the dots at the reference point (see Figure 7.7b).

Fig. 7.7 Aligning a textline to the dot character

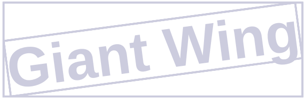
	Generated output	***Option list for PDF_fit_textline( )***
a)		`boxsize={70 8} position={center bottom} alignchar=.`
b)		`boxsize={70 8} position={left bottom} alignchar=.`

7.1.5 Placing a 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, the following code fragment will fit the text from the lower left to the upper right corner *(ll2r)* of the fitbox. The borders are shown to illustrate the fitbox size as well (see Figure 7.8).

```
/* fit the text line and add a stamp from the lower left to the upper right */
String  optlist = "font=" + normalfont + " fontsize=8 boxsize={160 50} " +
                  "showborder=true stamp=ll2ur";

p.fit_textline("Giant Wing", 5, 5, optlist);
```

Fig. 7.8 Fitting a text line like a stamp from the lower left to the upper right

Generated output	***Option list for PDF_fit_textline( )***
	`showborder=true stamp=ll2ur`

7.1.6 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.. The following code fragment places a text line. Using the *leader* option with the *alignment={none right}* suboption, leaders are appended to the right, and repeated until the right border of the text box. There will be an equal distance between leader and right border while the distance between text and leader may differ (see Figure 7.9a).

```
/* fit the text line and add a leader like in the entries of a table of contents
String  optlist =
        "font=" + normalfont + " fontsize=8 boxsize={200 10} " +
        "leader={alignment={none right}}";

p.fit_textline("Features of Giant Wing", 10, 20, optlist);
```

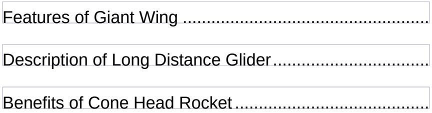
```
p.fit_textline("Description of Long Distance Glider", 10, 40, optlist);
p.fit_textline("Benefits of Cone Head Rocket", 10, 60, optlist);
```

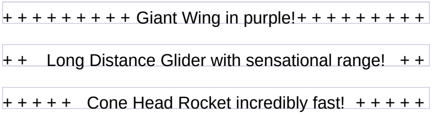
Leaders in a news ticker.. In another use case you might want to create a news ticker effect. In this case we use »+ « as leader. The text is placed in the center, and the leader is printed before and after the text *(alignment={left right})*. The left leader is aligned to the left border, but might have a varying distance to the text. The right leader is aligned to the right border (see Figure 7.9b).

```
/* fit the text line and add a Ticker-like leader */
String  optlist =
        "font=" + normalfont + " fontsize=8 boxsize={200 10} " +
        "position={center bottom} leader={alignment={left right} text={+ }}";

p.fit_textline("Giant Wing in purple!", 10, 20, optlist);
p.fit_textline("Long Distance Glider with sensational range!", 10, 40, optlist);
p.fit_textline("Cone Head Rocket incredibly fast!", 10, 60, optlist);
```

Fig. 7.9 Fitting a text line using leaders

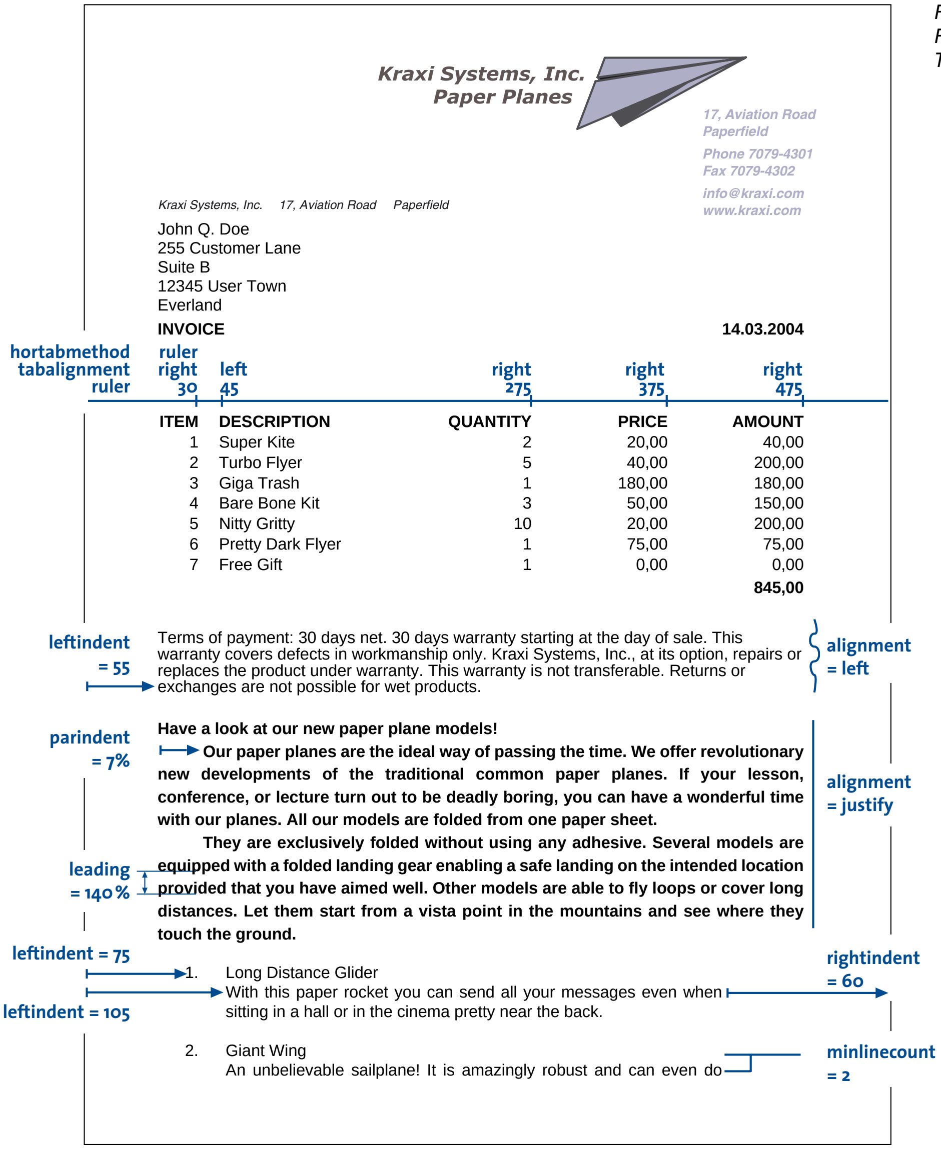
	Generated output	***Option list for PDF_fit_textline( )***

a)	 Features of Giant Wing ....
	Description of Long Distance Glider.................................
	Benefits of Cone Head Rocket...
		boxsize={200 10}
		leader={alignment={none right}}

b)	 + + + + + + + + + Giant Wing in purple!+ + + + + + + + +
	+ + Long Distance Glider with sensational range! + +
	+ + + + + Cone Head Rocket incredibly fast! + + + + +
		boxsize={200 10}
		position={center bottom}
		leader={alignment={left right}}
		text={+ }

7.2 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 7.10 and Figure 7.11 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.



Kraxi Systems, Inc.
Paper Planes

17, Aviation Road
Paperfield

Phone 7079-4301
Fax 7079-4302

info@kraxi.com
www.kraxi.com

Kraxi Systems, Inc. 17, Aviation Road Paperfield

John Q. Doe
255 Customer Lane
Suite B
12345 User Town
Everland

INVOICE **14.03.2004**

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
4	Bare Bone Kit	3	50,00	150,00
5	Nitty Gritty	10	20,00	200,00
6	Pretty Dark Flyer	1	75,00	75,00
7	Free Gift	1	0,00	0,00
				845,00

Terms of payment: 30 days net. 30 days warranty starting at the day of sale. This warranty covers defects in workmanship only. Kraxi Systems, Inc., at its option, repairs or replaces the product under warranty. This warranty is not transferable. Returns or exchanges are not possible for wet products.

Have a look at our new paper plane models!

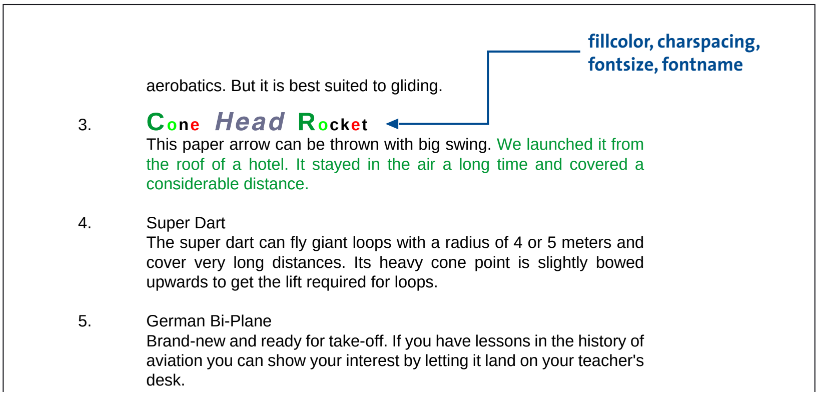
Our paper planes are the ideal way of passing the time. We offer revolutionary new developments of the traditional common paper planes. If your lesson, conference, or lecture turn out to be deadly boring, you can have a wonderful time with our planes. All our models are folded from one paper sheet.

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.

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

Fig. 7.10 Formatting Textflows



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.

4. Super Dart
The super dart can fly giant loops with a radius of 4 or 5 meters and cover very long distances. Its heavy cone point is slightly bowed upwards to get the lift required for loops.

5. German Bi-Plane
Brand-new and ready for take-off. If you have lessons in the history of aviation you can show your interest by letting it land on your teacher's desk.

Fig. 7.11 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 function 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 7.1, »Placing and Fitting Single-Line Text«, page 135. In the below sections we will focus on options related to multi-line text.

7.2.1 Placing Textflows in the Fitbox

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.

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, num, "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(-1, text, "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 7.12).

```
/* 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.
*/
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);
```

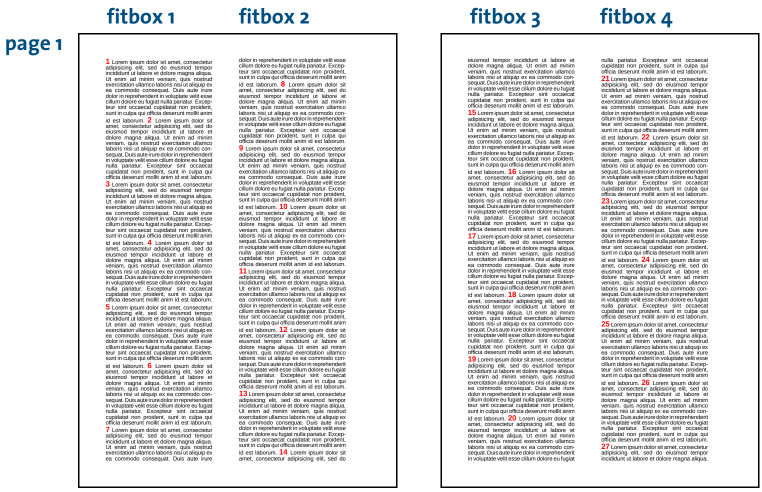
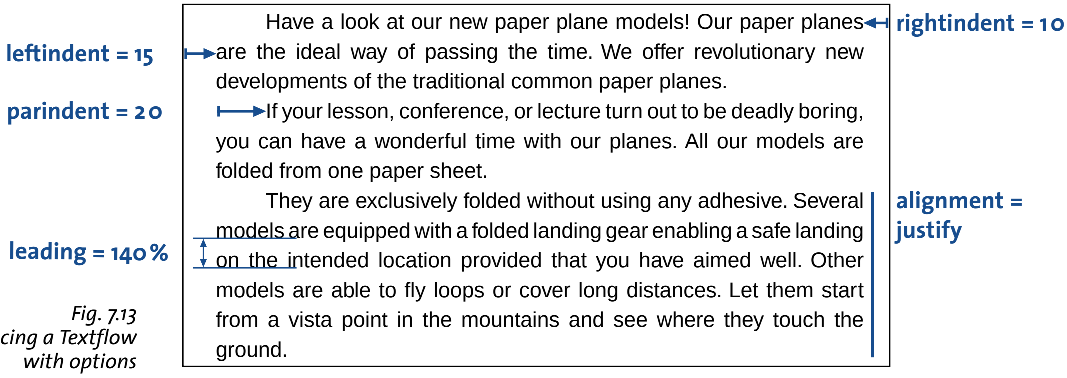


Fig. 7.12
Placing a Textflow in two fitboxes



Have a look at our new paper plane models! Our paper planes are the ideal way of passing the time. We offer revolutionary 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. 7.13
Placing a Textflow with options

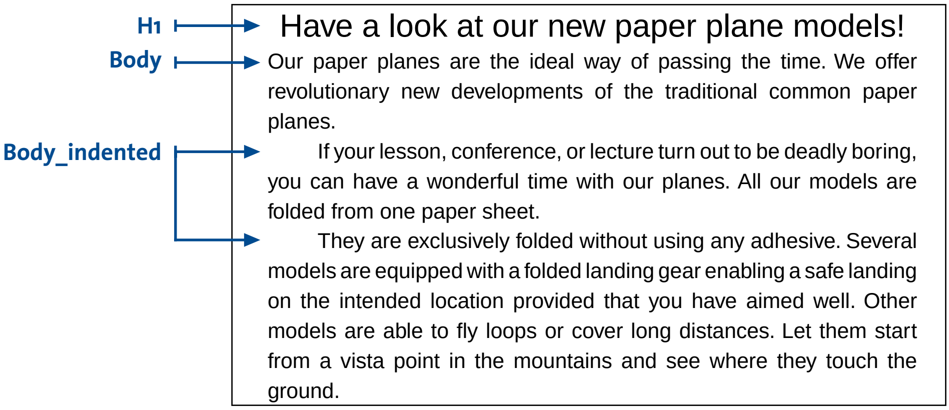
```
        /* Fill the second column if we have more text*/
        if (!result.equals("_stop"))
            result = p.fit_textflow(tf, llx2, lly2, urx2, ury2, optlist);

        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: Textflow box too small");
        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 '" + result + "' found in Textflow");
        }
}
p.delete_textflow(tf);
```

7.2.2 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 10 units. The text should be justified against both margins, and the leading increased to 140%. Finally, we'll reduce the font size to 8 points. To achieve this, extend the option list for *PDF_add_textflow()* as follows (see Figure 7.13):

```
String  optlist =
        "leftindent=15 rightindent=10 parindent=20 alignment=justify " +
        "leading=140% fontname=Helvetica fontsize=8 encoding=unicode";
```



Have a look at our new paper plane models!

Our paper planes are the ideal way of passing the time. We offer revolutionary 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. 7.14
Combining inline options with macros

7.2.3 Inline Option Lists and Macros

The text in Figure 7.13 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_texflow( ). 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.

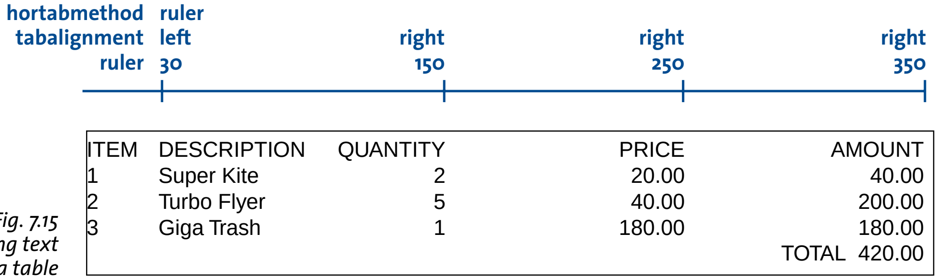
Macros. The text above contains several different types of paragraphs, such as heading or body text with or without indentation. Each of these paragraph types is formatted differently and occurs multiply in longer Textflows. In order to avoid starting each paragraph with the corresponding inline options, we can combine these in macros, and refer to the macros in the text via their names. As shown in Figure 7.14 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



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

Fig. 7.15 Placing text as a table

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.

7.2.4 Tab Stops

In the next example we will place a table with left- and right-aligned columns using tab characters. The table contains the following lines of text, where individual entries are separated from each other with a tab character (indicated by arrows):

```
ITEM → DESCRIPTION → QUANTITY → PRICE → AMOUNT ↵
1 → Super Kite → 2 → 20.00 → 40.00 ↵
2 → Turbo Flyer → 5 → 40.00 → 200.00 ↵
3 → Giga Trash → 1 → 180.00 → 180.00 ↵
→ → → → TOTAL  420.00
```

The following code fragment places the table, using the *ruler* option for defining the tab positions, *tabalignment* for specifying the alignment of tab stops, and *hortabmethod* for specifying the method used to process tab stops (the result can be seen in Figure 7.15):

```
/* assemble option list */
String  optlist =
        "ruler       ={30     150    250    350} " +
        "tabalignment={left right right right} " +
        "hortabmethod=ruler leading=120% fontname=Helvetica fontsize=9 encoding=winansi";

/* place Textflow in fitbox */
textflow = p.add_textflow(-1, table, 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);
```

Note PDFlib's table feature is recommended for creating complex tables (see Section 7.4, »Table Formatting«, page 165).

7.2.5 Numbered Lists and Paragraph Spacing

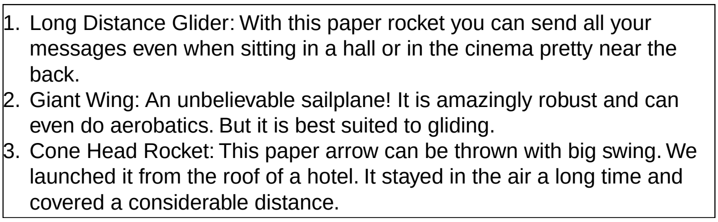
The following example demonstrates how to format a numbered list using the inline option *leftindent* (see Figure 7.16):

```
1.<leftindent 10>Long Distance Glider: With this paper rocket you can send all
your messages even when sitting in a hall or in the cinema pretty near the back. ↵
<leftindent 0>2.<leftindent 10>Giant Wing: An unbelievable sailplane! It is amazingly
robust and can even do aerobatics. But it is best suited to gliding. ↵
<leftindent 0>3.<leftindent 10>Cone Head Rocket: This paper arrow can be thrown with big
swing. We launched it from the roof of a hotel. It stayed in the air a long time and
covered a considerable distance.
```

Setting and resetting the indentation value is cumbersome, especially since it is required for each paragraph. A more elegant solution defines a macro called *list*. For convenience it defines a macro *indent* which is used as a constant. The macro definitions are as follows:

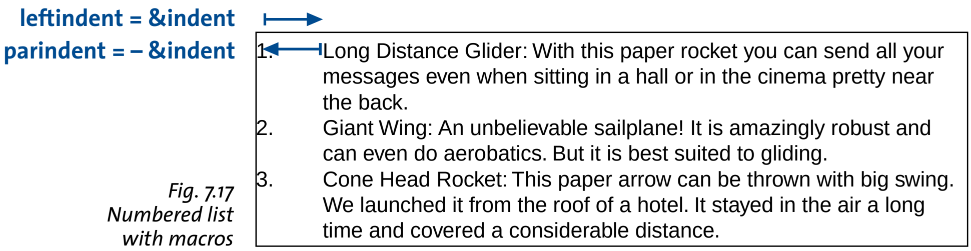
```
<macro {
indent {25}

list {parindent=-&indent leftindent=&indent hortabsize=&indent
hortabmethod=ruler ruler={&indent}}
}>
<&list>1. → Long Distance Glider: With this paper rocket you can send all your messages
even when sitting in a hall or in the cinema pretty near the back. ↵
2. → Giant Wing: An unbelievable sailplane! It is amazingly robust and can even do
aerobatics. But it is best suited to gliding. ↵
3. → Cone Head Rocket: This paper arrow can be thrown with big swing. We launched
it from the roof of a hotel. It stayed in the air a long time and covered a
considerable distance.
```

The *leftindent* option specifies the distance from the left margin. The *parindent* option, which is set to the negative of *leftindent*, cancels the indentation for the first line of each paragraph. The options *hortabsize*, *hortabmethod*, and *ruler* specify a tab stop which corresponds to *leftindent*. It makes the text after the number to be indented with the



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. 7.16 Numbered 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.
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. 7.17
Numbered list with macros

amount specified in *leftindent*. Figure 7.17 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:

```
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>2. → Giant Wing: An unbelievable sailplane! It is
amazingly robust and can even do
aerobatics. But it is best suited to gliding.
```

7.2.6 Control Characters, Character Mapping, and Symbol Fonts

Control characters in Textflows. Various characters are given special treatment in Textflows. PDFlib supports symbolic character names which can be used instead of the corresponding character codes in the *charmapping* option (which replaces characters in the text before processing it, see below). The *PDFlib Reference* lists all control characters which are evaluated by the Textflow functions along with their symbolic names, and explains their meaning. An option must only be used once per option list, but multiple option lists can be provided one after the other. For example, the following sequence will create an empty line:

```
<nextline><nextline>
```

Replacing characters or sequences of characters. The *charmapping* option can be used to replace some characters in the text with others. Let's start with an easy case where we will replace all tabs in the text with space characters. The *charmapping* option to achieve this looks as follows:

```
charmapping={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 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}}
```

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:

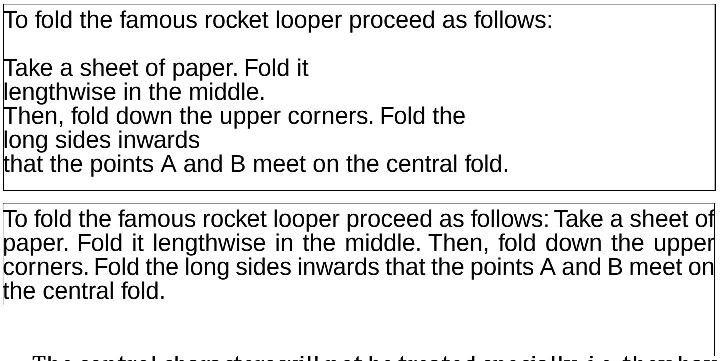
```
To fold the famous rocket looper proceed as follows: ↵ ↵
Take a sheet of paper. Fold it ↵
lengthwise in the middle. ↵
Then, fold down the upper corners. Fold the ↵
long sides inwards ↵
that the points A and B meet on the central fold.
```

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 7.18 shows Textflow output with the unmodified text and the improved version with the *charmapping* option.

Symbol fonts in Textflows and the textlen option. Symbol fonts, more precisely: text in a font which is not Unicode-compatible according to Section 5.4.4, »Unicode-compatible Fonts«, page 111, deserves some special attention when used within Textflows:



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.

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. 7.18
Top: text with redundant line breaks

Bottom: replacing the linebreaks with the `charmapping` *option*

- The control characters will not be treated specially, i.e. they have no special meaning.
- Some Textflow options will be ignored since they do not make sense for symbol fonts, e.g. *tabalignchar*.
- Since inline option lists cannot be used in text portions with symbol fonts (since the symbols don't have any intrinsic meaning it would be impossible to locate and interpret option lists), the length of text fragments consisting of symbol characters must explicitly be specified using the *textlen* option.
- After *textlen* characters a new inline option list must be placed in the text. Usually the next option list will switch to another font/encoding combination, but this is not required.

Omitting the *textlen* option for Symbol fragments, or failing to supply another inline option list immediately after the Symbol fragment will result in an exception.

The following fragment contains a Greek character from the Symbol font inserted between Latin characters:

```
<fontname=Helvetica fontsize=12 encoding=winansi>The Greek letter <fontname=Symbol
encoding=builtin textlen=1>A<fontname=Helvetica encoding=winansi> symbolizes beginning.
```

Using characters with codes greater than 127 (0x7F) can get a bit tricky subject to the syntax requirements of the programming language in use. The following examples create a right arrow from the ZapfDingbats font. This character has glyph name *a161* and code *0xD5* which corresponds to the character Õ in *winansi*.

The following example uses PDFlib's escape sequence syntax *\xD5*. If used directly in a C language program, the backslash must be preceded by another backslash. Processing escape sequences must be enabled with the *escapesequence* option. The length of the fragment (after \\ processing) is 4 bytes:

```
<escapesequence fontname=ZapfDingbats encoding=builtin textlen=4>\\xD5<fontname=Helvetica
encoding=winansi>
```

The following example uses the *\u* syntax of Java and other languages. The length of the text fragment (after *\u* expansion) is 1 Unicode character:

```
<fontname=ZapfDingbats encoding=builtin textlen=1>\u00D5<fontname=Helvetica
encoding=winansi>
```

The following example uses a literal character, assuming the source code is compiled in the *winansi/cp1252* codepage (e.g. *javac -encoding 1252)*. Again, the length of the text fragment is 1:

```
<fontname=ZapfDingbats encoding=builtin textlen=1>Õ<fontname=Helvetica encoding=winansi>
```

Instead of numerically addressing the character we can refer to its glyph name, using PDFlib's glyph name reference syntax (see Section 4.6.2, »Character References and Glyph Name References«, page 89) which requires *unicode* encoding. Glyph name processing must be enabled with the *charref* option, The length of the text fragment is 7 characters since the complete contents of the glyph name reference are counted. In Unicode-aware language bindings the following example will do the trick:

```
<charref fontname=ZapfDingbats encoding=unicode textlen=7>&.a161;<fontname=Helvetica
encoding=winansi>
```

In non-Unicode-aware language bindings we must set the text format to *bytes* since otherwise two bytes per character would be required for *unicode* encoding:

```
<charref fontname=ZapfDingbats encoding=unicode textformat=bytes textlen=7>&.a161;
<fontname=Helvetica encoding=winansi>
```

7.2.7 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:

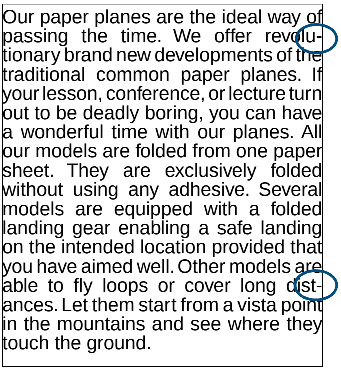
- In all *cp1250 – cp1258* (including *winansi)* and *iso8859-1 – iso8859-16* encodings the soft hyphen is at decimal 173, octal 255, or hexadecimal 0xAD.
- In *ebcdic* encoding the soft hyphen is at decimal 202, octal 312, or hexadecimal 0xCA.
- A character entity reference can be used if an encoding does not contain the soft hyphen character (e.g. *macroman)*: *&shy;*

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):

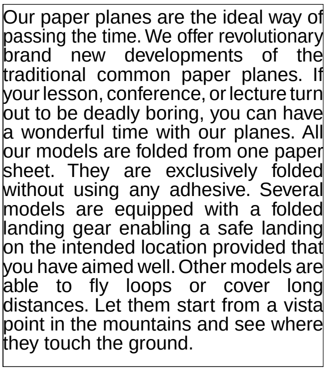
```
Our paper planes are the ideal way of pas – sing the time. We offer revolu – tionary
brand new dev – elop – ments of the tradi – tional common paper planes. If your lesson,
confe – rence, or lecture turn out to be deadly boring, you can have a wonder – ful time
with our planes. All our models are 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 7.19 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 7.20, the output looks okay even without explicit soft hyphens. The option list in both cases looks as follows:



Our paper planes are the ideal way of
passing the time. We offer revolu-
tionary 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 dist-
ances. Let them start from a vista point
in the mountains and see where they
touch the ground.

Fig. 7.19
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. 7.20
Justified text without soft hyphens, using default settings and a wide fitbox.

```
fontname=Helvetica fontsize=9 encoding=winansi alignment=justify
```

7.2.8 Controlling the Linebreak Algorithm

PDFlib implements a sophisticated line-breaking algorithm.[1] Table 7.1 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.

Table 7.1 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`	
	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.*
avoidbreak	*(Boolean) If* `true`*, avoid any line breaks until* `avoidbreak` *is reset to* `false`*. Default:* `false`	

[1] *For interested users we'll note that PDFlib honors the recommendations in »Unicode Standard Annex #14: Line Breaking Properties« (see www.unicode.org/reports/tr14). Combining marks are not taken into account.*

Table 7.1 Options for controlling the line-breaking algorithm

option	*explanation*
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):* ***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 & }`
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*
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*

For parentheses and quotation marks it depends on whether we have an opening or closing character: opening parentheses and quotations marks do not offer any break opportunity. In order to find out whether a quotation mark starts or ends a sequence, pairs of quotation marks are examined.

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.

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%
```

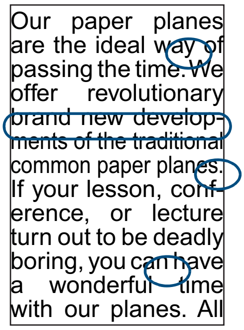

Our paper planes
are the ideal way of
passing the time. We
offer revolutionary
brand new develop-
ments of the traditional
common paper planes.
If your lesson, conf-
erence, or lecture
turn out to be deadly
boring, you can have
a wonderful time
with our planes. All

Fig. 7.21 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)

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 7.21 demonstrates the results of the various methods for justifying text in a narrow fitbox. The option settings in Figure 7.21 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.

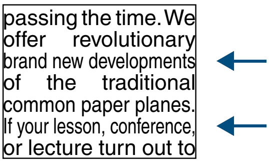
If the fitbox is so narrow that occasionally forced hyphenations occur, you should consider inserting soft hyphens, or modify the options which control justified text.

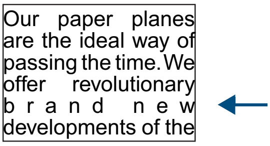
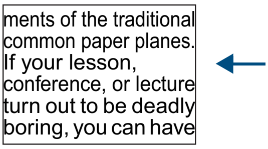
Option shrinklimit for justified text. The most visually pleasing solution is to reduce the *shrinklimit* option which specifies a lower limit for the shrinking factor applied by the *shrink* method. Figure 7.22a shows how to avoid forced hyphenation by compressing text down to *shrinklimit=50%*.

Option spreadlimit for justified text. Expanding text, which is achieved by the *spread* method and controlled by the *spreadlimit* option, is another method for controlling line breaks. This unpleasing method should be rarely used, however. Figure 7.22b 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 7.22c shows the generated text output with a minimum text width of 50%.

Fig. 7.22 Options for justified text in a narrow fitbox

Generated output		*Option list for PDF_fit_textflow( )*
a)	 passing the time. We offer revolutionary brand new developments of the traditional common paper planes. If your lesson, conference, or lecture turn out to	`alignment=justify shrinklimit=50%`

Generated output		*Option list for PDF_fit_textflow( )*
b)	 Our paper planes are the ideal way of passing the time. We offer revolutionary brand new developments of the	`alignment=justify spreadlimit=5`
c)	 ments of the traditional common paper planes. If your lesson, conference, or lecture turn out to be deadly boring, you can have	`alignment=justify nofitlimit=50`

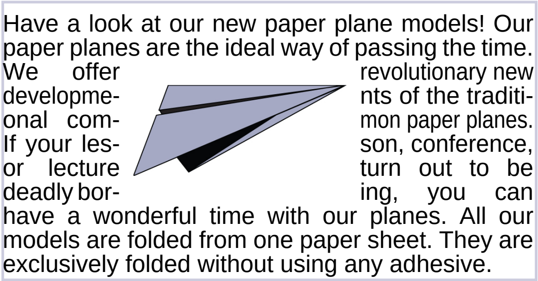
7.2.9 Wrapping Text

The wrapping feature can be used to place graphics within the Textflow and wrap the text around it.

Wrapping text around an image. In this example we will place an image within the Textflow and run the text around the image. To accomplish this we load the image and fit it into a box, define that box as the area to be left without text and fit the text around the image (see Figure 7.23).

```
image = p.load_image("auto", "plane.jpg", "");
if (image == -1)
        throw new Exception("Error: " + p.get_errmsg());
p.fit_image(image, 50, 35,
        "boxsize={80 46} fitmethod=meet position=center matchbox={name=img margin=-5}");

textflow = p.add_textflow(-1, text,
        "fontname=Helvetica fontsize=9 encoding=unicode alignment=justify");
if (textflow == -1)
        throw new Exception("Error: " + p.get_errmsg());
result = p.fit_textflow(textflow, left_x, left_y, right_x, right_y,
        "wrap={usematchboxes={img}}");
if (!result.equals("_stop"))
        { /* ... */ }
p.delete_textflow(textflow);
```

First, the image is loaded and fit into the box at the desired position on the page. To reference the image box later, in *PDF_fit_image( )* a matchbox called *img* with a margin of 5 units is defined with *matchbox={name=img margin=-5}*. Then, the textflow is added and



Have a look at our new paper plane models! Our paper planes are the ideal way of passing the time. We offer revolutionary 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.

Fig. 7.23
Wrapping the textflow around an image

placed using the *wrap* option with the image's matchbox *img* as the area to run around `wrap={usematchboxes={{img all}}}`.

Before placing the text you can fit further images using the same matchbox name. In this case the text will run around all images.

7.3 Placing Images and Imported PDF Pages

The function *PDF_fit_image( )* for placing raster image and templates as well as *PDF_fit_pdi_page( )* for placing imported PDF pages offer a wealth of options for controlling the placement on the page. This section demonstrates the most important options by looking at some common application tasks. A complete list and descriptions of all options can be found in the *PDFlib 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.

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 *(0, 0)*.

```
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 *(0, 0)* 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.

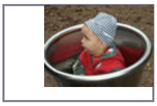
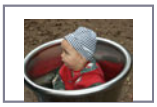
7.3.2 Positioning an Object in a Box

In order to position an object, an additional box with predefined width and height can be used. Figure 7.24 shows the output of the examples described below. Note that the blue box or line is depicted for visualising the box size; 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 7.24a).

Similarly, we can place the image at the center of the bottom. This case is depicted in Figure 7.24b.

Fig. 7.24 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}`

7.3.3 Fitting an Object into a Box

In this section we fit the object into the box by using various fit methods. Let's start with the default case, where no fit method will be 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 7.25a illustrate that simple case.

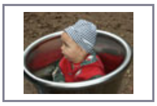
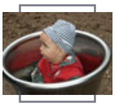
Decreasing the box width from 70 to 35 units doesn't have any effect on the output. The image will remain in its original size and will exceed the box (see Figure 7.25b).

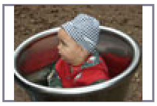
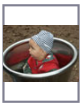
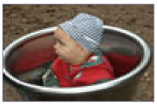
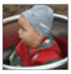
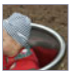
Fitting an image in the center of a box. In order to center an image within a predefined rectangle you don't have to do any calculations, but can achieve this with suitable options. With *position=center* we place the image in the center of the box, 70 units wide and 45 high *(boxsize={70 45})*. Using *fitmethod=meet*, the image is proportionally resized until its height completely fits into the box (see Figure 7.25c).

Decreasing the box width from 70 to 35 units will scale down the image until its width completely fits into the box (see Figure 7.25d).

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 7.25e).

Fig. 7.25 Fitting an image into a box subject to various fit methods

Generated output	***Option list for PDF_fit_image( )***
a) 	`boxsize={70 45} position=center`
b) 	`boxsize={35 45} position=center`

Generated output		*Option list for PDF_fit_image( )*
c)		`boxsize={70 45} position=center fitmethod=meet`
d)		`boxsize={35 45} position=center fitmethod=meet`
e)		`boxsize={70 45} position=center fitmethod=entire`
f)		`boxsize={30 30} position=center fitmethod=clip`
g)		`boxsize={30 30} position={right top} fitmethod=clip`

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 7.25f).

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 7.25g).

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.

7.3.4 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 7.28a). Since we have not specified any fit method the image will be output in its original size and will exceed the box.

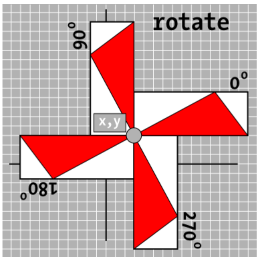


Fig. 7.26
The rotate option

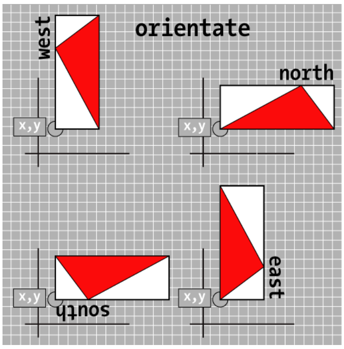


Fig. 7.27
The orientate option

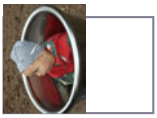
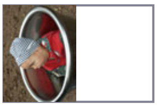
Fitting an image proportionally into a box with orientation. Our next goal is to orientate the image to the west with a predefined size. We define a box of the desired size and fit the image into the box with the image's proportions being unchanged *(fitmethod=meet)*. The orientation is specified as *orientate=west*. By default, the image will be placed in the lower left corner of the box (see Figure 7.28b).

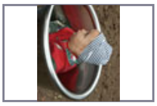
The *orientate* option supports the direction keywords *north, east, west,* and *south* as demonstrated in Figure 7.27.

Note that the *orientate* option has no influence on the whole coordinate system but only on the placed object.

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 7.28c).

Fig. 7.28 Orientating an image

Generated output		***Option list for PDF_fit_image( )***
a)		`boxsize {70 45} orientate=west`
b)		`boxsize {70 45} orientate=west fitmethod=meet`

c)		`boxsize {70 45} position={center bottom} orientate=east`
		`fitmethod=clip`

7.3.5 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 7.29a.

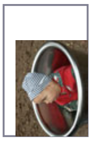
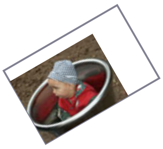
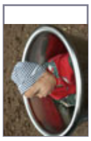
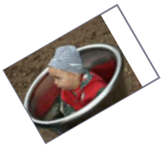
Since the rotation affects the whole coordinate system, the box will be rotated as well. Similarly, we can rotate the image by 30° counterclockwise (see Figure 7.29b).

Figure 7.26 demonstrates the general behaviour of the *rotate* option.

Fitting an image with rotation. Our the next goal is to fit the image rotated by 90° counterclockwise into the box while maintaining its proportions. This is sccomplished using *fitmethod=meet* (see Figure 7.29c).

Similarly, we can rotate the image by 30° counterclockwise and proportionally fit the image into the box (see see Figure 7.29d).

Fig. 7.29 Rotating an image

Generated output		*Option list for PDF_fit_image( )*
a)		`boxsize {70 45} rotate=90`
b)		`boxsize {70 45} rotate=30`
c)		`boxsize {70 45} rotate=90 fitmethod=meet`
d)		`boxsize {70 45} rotate=30 fitmethod=meet`

7.3.6 Adjusting the Page Size

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 7.30) 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");
```

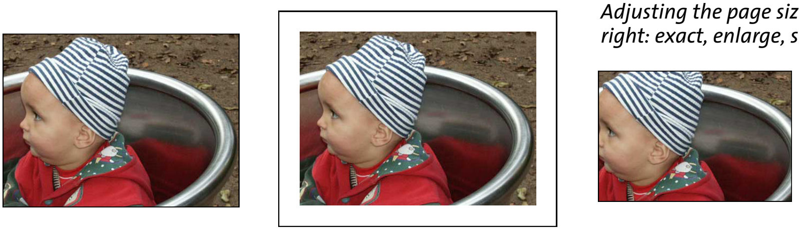


Fig. 7.30
Adjusting the page size. Left to right: exact, enlarge, shrink

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");
```

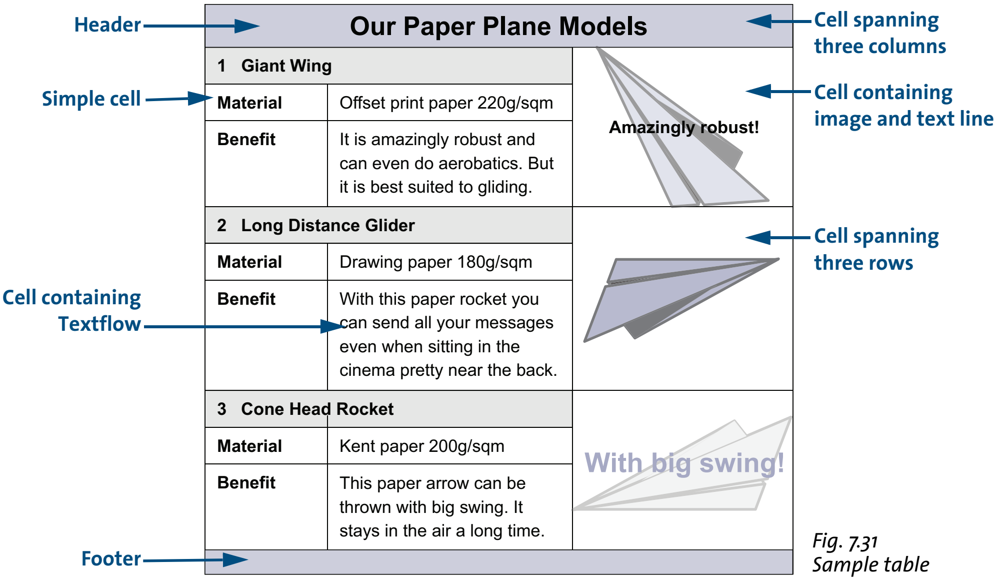
In addition to placing by means of *x* and *y* coordinates (which specify the object's distance from the page edges, or the coordinate axes in the general case) you can also specify a target box. This is a rectangular area in which the object will be placed subject to various formatting rules. These can be controlled with the *boxsize*, *fitmethod* and *position* options.

7.4 Table Formatting

The table formatting feature can be used to automatically forat 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.

General aspects of a table. The description of the table formatter is based on the following concepts and terms (see Figure 7.31):

- 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 Figure 7.37).
- 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. 7.31
Sample table

As an example, all aspects of creating the table in Figure 7.31 will be explained. A complete description of the table formatting options can be found in the *PDFlib 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 »Matchboxes«, page 176, 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 graphics state. Table cells can be defined in `document` *scope while the actual table placement must be done in* `page` *scope.*

7.4.1 Contents of a Table Cell

When adding a cell to a table with *PDF_add_table_cell(),* you can specify various kinds of cell contents. For example, the cells of the paper plane table contain the elements illustrated in Figure 7.32.

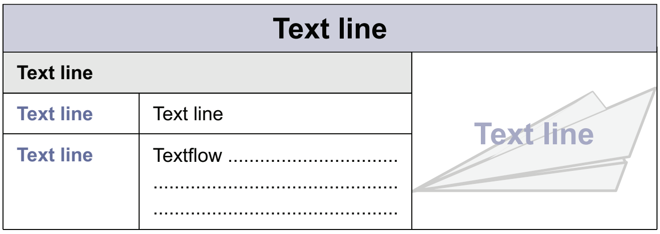


Fig. 7.32 Contents of the table cells

Single-line text. The text is supplied in the *text* parameter of *PDF_add_table_cell().* In the *fittextline* option all of the 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 given it will be calculated as the font size times 1.5. 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 of the 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().* The image or template handle is supplied in the *image* option. In the *fitimage* option all of the 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 of the 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.

Multiple content types in a cell. Table cells can contain one or more of those content types at the same time. Additional ruling and shading is available, as well as matchboxes for interactive features such as links or form fields.

Positioning cell contents. 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 of the margins is defined, the cell contents will be placed with respect to the inner cell box (see Figure 7.33). If no margins are defined the inner cell box will be identical to the cell box.

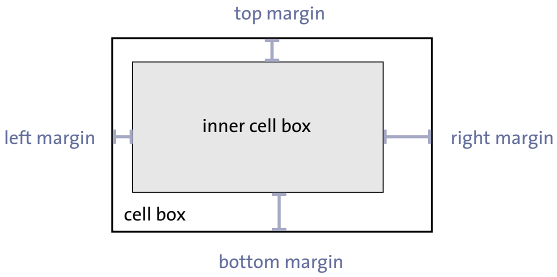


Fig. 7.33
Fitting contents in the inner cell box

In addition, the cell contents may be subject to further options supplied in the content-specific fit options, as described in section »Large Table Example«, page 171.

7.4.2 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 7.34 shows an example of a table containing three columns and four rows.

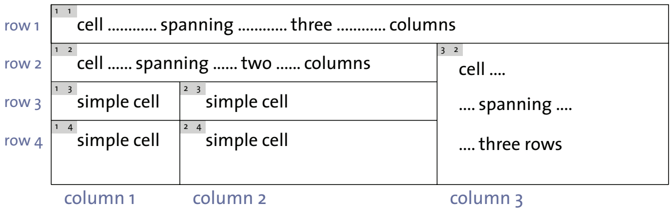


Fig. 7.34
Simple cells and cells spanning several rows or columns

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 7.35 shows an example.

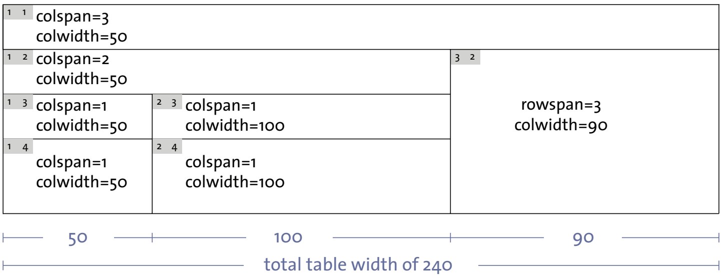
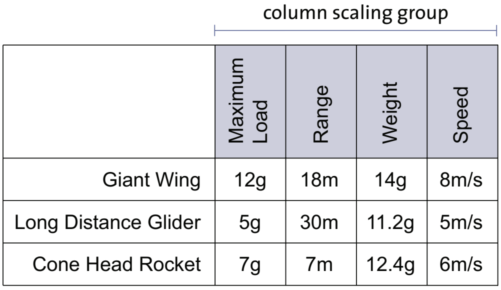

1 1 colspan=3 colwidth=50
1 2 colspan=2 colwidth=50
3 2
1 3 colspan=1 colwidth=50
2 3 colspan=1 colwidth=100
rowspan=3 colwidth=90
1 4 colspan=1 colwidth=50
2 4 colspan=1 colwidth=100
50
100
90
total table width of 240

Fig. 7.35
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 7.36),


column scaling group

	Maximum 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. 7.36
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.

7.4.3 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. as a multi-column layout, and on several pages.

The table in Figure 7.37 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.

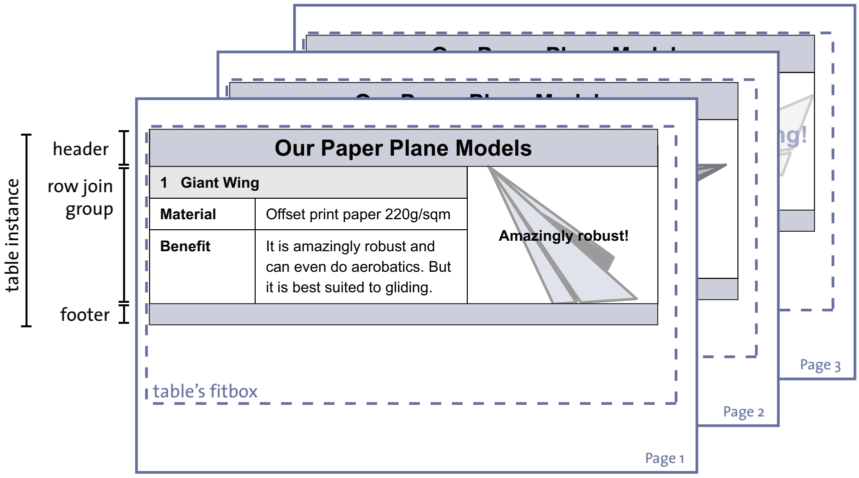


Fig. 7.37
Table broken into several table instances placed in one fitbox each.

Defining 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.

If the fitbox is too small to hold the required header, footer, and at least the next body row it will remain empty, and *PDF_fit_table( )* will return the string *_empty*.

Splitting a Textflow cell. If a cell containing a Textflow 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%. In this default case the Textflow will not be split and the cell will completely be placed into the next table instance.

In order to distribute the Textflow contents across several table instances you can decrease the *minrowheight* value to split the cell with the given percentage of contents in the last body row of the first instance, and place the remaining contents in the first body row of the next instance. If the affected row contains contents other than Textflows they will be repeated in the next table instance.

Figure 7.38 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.

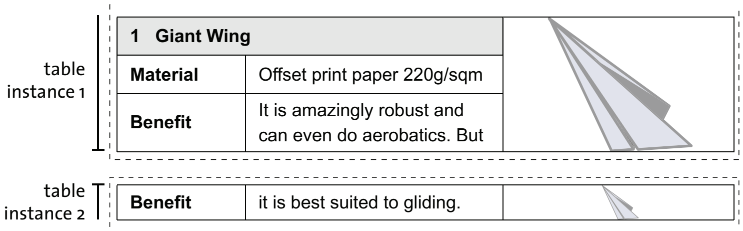


table instance 1
1 Giant Wing
Material
Offset print paper 220g/sqm
Benefit
It is amazingly robust and can even do aerobatics. But
table instance 2
Benefit
it is best suited to gliding.

Fig. 7.38 Splitting a Textflow cell

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 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. In order to avoid any shrinking use *horshrinklimit=100%*.

7.4.4 Creating a Simple Table

As a first simplistic example, we will create a simple table containing a row with a single cell with the centered text line *Our Paper Plane Models*. We add the cell to the table by providing the required options *font* and *fontsize* in the *fittextline* option list. We define a row height of 18, and left and right cell margins of 2. Since we don't supply any column widths, it will be calculated automatically from the text line plus the margins.

In the next step we place the table while visualizing the table borders with black lines and filling the table completely with color.

```
String  optlist =
        "fittextline={font=" + boldfont + " fontsize=12 position=center} rowheight=18 " +
        "marginleft=2 marginright=2";

tbl = p.add_table_cell(tbl, 1, 1, "Our Paper Plane Models", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());

optlist = "fill={{area=table fillcolor={rgb 0.8 0.8 0.87}}} " +
          "stroke={{line=frame linewidth=0.3} {line=other linewidth=0.3}}";

result = p.fit_table(tbl, 0, tableheight, tablewidth, 0, optlist);
if (result.equals("_error"))
        throw new Exception ("Couldn't place table: " + p.get_errmsg());
```

In addition to the *rowheight* value you can use the *margintop* and *marginbottom* options to increase the vertical distance between the text line and the cell top or bottom.

7.4.5 Large Table Example

In the following sections we will create the sample table shown in Figure 7.39 step by step. As a prerequisite we need to load two fonts, define the table size and start a new page in DIN A4 format:

```
int tablewidth = 240, tableheight = 150;

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 header cell. We start with the header 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. The text line is centered vertically and horizontally. The following code fragment shows how to add the header cell.

```
String  optlist =
        "fittextline={font=%d fontsize=12 position=center} " +
        "rowheight=18 colspan=3 colwidth=50";

tbl = p.add_table_cell(tbl, 1, 1, "Our Paper Plane Models", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());
```

When placing the table later after having added more cells, the option list of *PDF_fit_table()* will be extended by the option *header=1* to indicate that the first row of the table will be used as header.

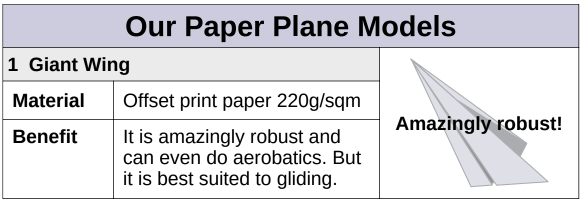
Step 2: Adding one cell spanning two columns. In the next step we add the cell containing the text line *1 Giant Wing*. It will be placed in the first column of the second row and spans two columns. The first column has a width of 50. The row height is 14. The cell is positioned on the left with a margin of 2.

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 7.5, »Matchboxes«, page 176.)

The following code fragment demonstrates how to add the *Giant Wing* heading cell:

```
String  optlist =
        "fittextline={font=" + boldfont + " fontsize=8 position={left center}} " +
        "rowheight=14 marginleft=2 colspan=2 colwidth=50 " +
        "matchbox={fillcolor={gray .92}}";

tbl = p.add_table_cell(tbl, 1, 2, "1  Giant Wing", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());
```

Fig. 7.39 Adding table cells with different properties step by step

Generated table	*Generation steps*
	*Step 1: Add the header cell spanning 3 columns* *Step 2: Add one 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*

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.	

Step 3: Add three more text line 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. The cell contents are vertically positioned in the center and have a distance of 2 from the left margin.

```
String  optlist =
        "fittextline={font=" + boldfont + " fontsize=8 position={left center}}" +
        " rowheight=14 marginleft=2 colwidth=50 rowjoingroup=giantwing";

tbl = p.add_table_cell(tbl, 1, 3, "Material", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());

tbl = p.add_table_cell(tbl, 1, 4, "Benefit", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());

optlist = "fittextline={font=" + normalfont + " fontsize=8 position={left center}}" +
          "rowheight=14 marginleft=2 colwidth=120 rowjoingroup=giantwing";

tbl = p.add_table_cell(tbl, 2, 3, "Offset print paper 220g/sqm", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());
```

Step 4: Add the Textflow cell. The following code fragment adds the *It is amazingly...* Textflow cell.

```
String  tftext =
        "It is amazingly robust and can even do aerobatics. " +
        "But it is best suited to gliding.";

String  optlist = "font=" + normalfont + " fontsize=8 leading=110%";

tflow = p.add_textflow(-1, tftext, optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());

optlist =
        "textflow=" + tflow + " marginbottom=2 marginleft=2 marginright=2 colwidth=120";

tbl = p.add_table_cell(tbl, 2, 4, "", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());
```

Step 5: Add the image cell with a text line. In the last 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. The cell margins are set to 4. In the first case we place a JPEG image in the cell.

```
giant_wing_image = p.load_image("auto", "giant_wing.jpg", "");
if (giant_wing_image == -1)
        throw new Exception("Error loading image: " + p.get_errmsg());

String  optlist =
        "fittextline={font=" + boldfont + " fontsize=8} image=" + giant_wing_image +
        " colwidth=90 rowspan=3 margin=4;

tbl = p.add_table_cell(tbl, 3, 2, "Amazingly robust!", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + 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("giant_wing.pdf", "", 0);
if (tbl == -1)
        throw new Exception("Error opening pdi document: " + p.get_errmsg());

int page = p.open_pdi_page(doc, pageno, "");
if (tbl == -1)
        throw new Exception("Error opening pdi page: " + p.get_errmsg());


String  optlist =
        "fittextline={font=" + boldfont + " fontsize=8} pdipage=" + page +
        " colwidth=90 rowspan=3 margin=4";

tbl = p.add_table_cell(tbl, 3, 2, "Amazingly robust!", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());
```

Fine-tuning the alignment of table cells. We want to make sure that all cells will have the same distance from the cell borders regardless of whether they are a Textflow or a text line cell. Some considerations are needed.

In the first step we place the *Material* and *Benefit* text line cells with the following option list:

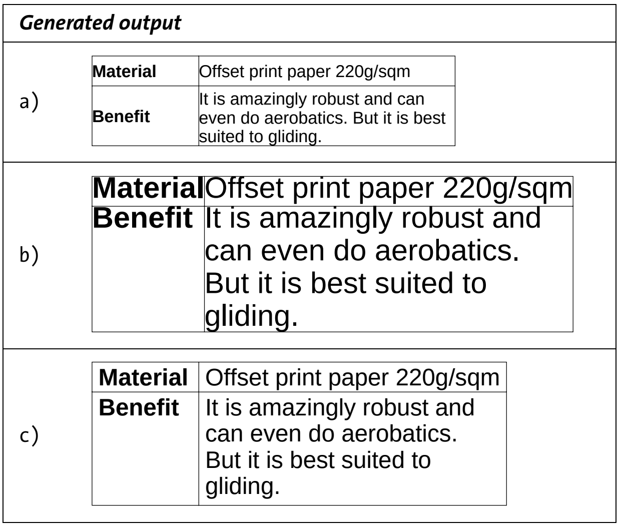
```
fittextline={position={left center} fontsize=8} rowheight=14 colwidth=50
```

The *Offset print paper ...* cell is added with:

```
fittextline={position={left center} fontsize=8} rowheight=14 colwidth=120
```

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 (see Figure 7.40a).

Fig. 7.40 Aligning text lines and Textflow in table cells



Generated output

a)

Material	Offset print paper 220g/sqm
Benefit	It is amazingly robust and can even do aerobatics. But it is best suited to gliding.

b)

Material	Offset print paper 220g/sqm
Benefit	It is amazingly robust and can even do aerobatics. But it is best suited to gliding.

c)

Material	Offset print paper 220g/sqm
Benefit	It is amazingly robust and can even do aerobatics. But it is best suited to gliding.

Instead, 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.

In the first step, we align all text without any distance from the cell borders and define a fixed row height. The text line cells are added with the following option list:

```
fittextline={position={left top} fontsize={capheight=8}} rowheight=14
```

The height of the *Material* row is determined by using the *rowheight* option with a value of 14. In contrast, the height of the *Benefit* row is determined by the height of the Textflow cell. We create the Textflow with *fontsize={capheight=8}*. To avoid unwanted space at the top we add the Textflow cell using *fittextflow={firstlinedist=capheight}*. Let's first create the table without any cell margins (see Figure 7.40b).

To create some space between text and cell borders we add a margin of 3 units to all table cells (see Figure 7.40c). The following code fragment shows how all text lines and the Textflow are aligned to have the same distance from the upper cell border:

```
String optlist =
        "fittextline={position={left top} font=" + boldfont +
        " fontsize={capheight=8}} rowheight=14 colwidth=50 margin=3";
tbl = p.add_table_cell(tbl, 1, 1, "Material", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());

tbl = p.add_table_cell(tbl, 1, 2, "Benefit", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());

optlist = "fittextline={position={left top} font=" + normalfont +
          " fontsize={capheight=8}} rowheight=14 colwidth=120 margin=3";

tbl = p.add_table_cell(tbl, 2, 1, "Offset print paper 220g/sqm", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());
```

```
optlist = "font=" + normalfont + " fontsize={capheight=8} leading=110%";
tflow = p.add_textflow(-1, tf_text, optlist);
if (tflow == -1)
        throw new Exception("Error adding textflow: " + p.get_errmsg());

optlist = "textflow=" + tflow + " fittextflow={firstlinedist=capheight} " +
          "colwidth=120 margin=3";
tbl = p.add_table_cell(tbl, 2, 2, "", optlist);
if (tbl == -1)
        throw new Exception("Error adding cell: " + p.get_errmsg());
p.begin_page_ext(0, 0, "width=250 height=160");

/* Place the table instance */
optlist = "stroke={{line=frame linewidth=0.3} {line=other linewidth=0.3}}";
result = p.fit_table(tbl, llx, lly, urx, ury, optlist);
if (result.equals("_error"))
         throw new Exception ("Couldn't place table : " + p.get_errmsg());
p.end_page_ext("");
```

7.5 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 actual element itself. As a result, the element may obscure the effect of the matchbox options, but not vice versa.

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 Reference*.

7.5.1 Decorating a Text Line

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 7.41a).

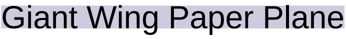
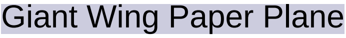
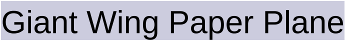
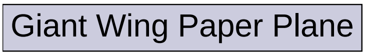
```
String  optlist =
        "font=" + normalfont + " fontsize=8 position={left top} " +
        "matchbox={fillcolor={rgb 0.8 0.8 0.87} boxheight={capheight none}}";

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 7.41b).

To increase the box height to match the font size we can use *boxheight={fontsize descender}* (see Figure 7.41c).

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 7.41d).

Fig. 7.41 Decorating a text line using a matchbox with various suboptions

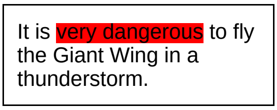
Generated output		***Suboptions of the matchbox option of PDF_fit_textline( )***
a)	 Giant Wing Paper Plane	`boxheight={capheight none}`
b)	 Giant Wing Paper Plane	`boxheight={ascender descender}`
c)	 Giant Wing Paper Plane	`boxheight={fontsize descender}`
d)	 Giant Wing Paper Plane	`boxheight={fontsize descender} borderwidth=0.3 offsetleft=-2 offsetright=2 offsetbottom=-2`

7.5.2 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 (Figure 7.42).

```
String  tftext =
        "It is <matchbox={boxheight={ascender descender} fillcolor={rgb 1 0 0}}>" +
        "very dangerous<matchbox=end> to fly the Giant Wing in a thunderstorm";

String  optlist = "font=" + normalfont + " fontsize=8 leading=110%";

tflow = p.create_textflow(tftext, optlist);
if (tflow == -1)
        throw new Exception("Error: " + p.get_errmsg());
p.fit_textflow(tflow, 0, 0, 100, 30, "");
if (!result.equals("_stop"))
        { /* ... */ }
```

Fig. 7.42 Textflow with matchbox inline option

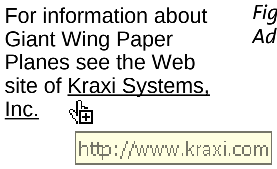
Generated output	***Text and inline options for PDF_create_textflow( )***
 It is very dangerous to fly the Giant Wing in a thunderstorm.	`It is <matchbox={fillcolor={rgb 1 0 0} boxheight={ascender descender}}>very dangerous <matchbox=end>"to fly the Giant Wing in a thunderstorm.`

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).

```
/* create and fit Textflow with matchbox "kraxi" */
String  tftext =
        "For more information about the Giant Wing Paper Plane see " +
        "<underline=true matchbox={name=kraxi boxheight={fontsize descender}}>" +
        "www.kraxi.com<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 *www.kraxi.com* text spans several lines the appropriate number of link annotations will be created automatically with a single call to *PDF_create_annotation( )*. The result in shown in Figure 7.43.



For information about Giant Wing Paper Planes see the Web site of Kraxi Systems, Inc.

http://www.kraxi.com

Fig. 7.43
Add Weblinks to parts of a Textflow

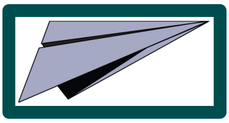
7.5.3 Matchboxes and Images

Drawing a frame around an image. In this example we completely fit an image into a box while maintaining its proportions using *fitmethod=meet*. We use the *matchbox* option with the *borderwidth* suboption to draw a thick rectangle around the image. The *strokecolor* suboption determines the border color, and the *linecap* and *linejoin* suboptions are used to round the corners.

The matchbox is always drawn before the image which means it would be 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 (see Figure 7.44).

```
image = p.load_image("auto", "giant_wing.jpg", "");
if (image == -1)
        throw new Exception("Error: " + p.get_errmsg());

String  optlist =
        "boxsize={60 60} position={center} fitmethod=meet " +
"       "matchbox={name=giantwing borderwidth=4 offsetleft=-3 offsetright=3 " +
        "offsetbottom=-3 offsettop=3 linecap=round linejoin=round " +
        "strokecolor {rgb 0.0 0.3 0.3}}";

p.fit_image(image, 6, 5, optlist);
p.close_image(image);
```

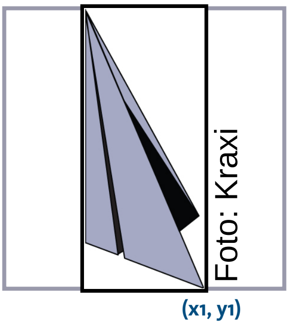
Fig. 7.44 Using the matchbox feature to draw a frame around the image

Generated output	***Option list for PDF_fit_image( )***
	`boxsize={60 60} position={center} fitmethod=meet matchbox={name=giantwing borderwidth=4 offsetleft=-3 offsetright=3 offsetbottom=-3 offsettop=3 linecap=round linejoin=round strokecolor {rgb 0.0 0.3 0.3}}`

Drawing a frame based on matchbox coordinates. In the next example the image is orientated to the west and the fit method *meet* is selected to fit the image proportionally to the supplied box. Then we retrieve the actual coordinates of the fitbox with *PDF_info_matchbox( )* and place a vertical text line relative to the lower right *(x1, y1)* corner of the fitbox. The border of the box is stroked in blue while the border of the image's actual fitbox is shown in black. The coordinates *(x1, y1)* of the fitbox are retrieved from the matchbox info (see Figure 7.45).

```
/* load and fit the image */
String  optlist =
        "boxsize={130 130} position={center} orientate=west " +
        "fitmethod=meet matchbox={name=giantwing borderwidth=1 " +
        "offsetleft=-0.5 offsetright=0.5 offsetbottom=-0.5 offsettop=0.5}";

image = p.load_image("auto", "giant_wing.jpg", "");
if (image == -1)
        throw new Exception("Error: " + p.get_errmsg());

p.fit_image(image, 10, 10, optlist);
p.close_image(image);

/* retrieve the coordinates of the third matchbox corner */
if ((int) p.info_matchbox("giantwing", 1, "exists") == 1)
{
        x1 = p.info_matchbox("giantwing", 1, "x1");
        y1 = p.info_matchbox("giantwing", 1, "y1");
}
/* start the text line at that corner */
optlist = "font=" + normalfont + " fontsize=8 orientate=west";
p.fit_textline("Foto: Kraxi", x1+2, y1+2, optlist);
```

Fig. 7.45 Using the matchbox feature to retrieve the coordinates to fit the text line

Generated output	***Generation steps***
	*Step 1: Fit image with matchbox* *Step 2: Get matchbox info for coordinates x1, y1* *Step 3: Fit text line starting at the retrieved coordinates with option* `orientate=west`

Adding a link to an image. Similar to the Textflow matchbox above, the image matchbox can be used to add a Web link to the area covered by the image. You can use almost the same code as in the Web link example above, only the name has to be adjusted to the image matchbox name *giantwing*.

```
/* load and fit the image */
String  optlist = "boxsize={130 130} fitmethod=meet matchbox={name=giantwing}";

image = p.load_image("auto", "giant_wing.jpg", "");
if (image == -1)
        throw new Exception("Error: " + p.get_errmsg());

p.fit_image(image, 10, 10, optlist);
p.close_image(image);

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

/* create link annotation on matchbox giantwing */
optlist = "action={activate " + act + "} linewidth=0 usematchbox={giantwing}";
p.create_annotation(0, 0, 0, 0, "Link", optlist);
```

8 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, hypertext, etc. pCOS users are assumed to have some basic knowledge of internal PDF structures and dictionary keys, but do not have to deal with PDF syntax and parsing details.

We strongly recommend that pCOS users obtain a copy of the PDF reference, which is available as follows:

Adobe Systems Incorporated: PDF Reference, Fifth Edition: Version 1.6. Downloadable PDF from *partners.adobe.com/public/developer/pdf/index_reference.html*

8.1 Simple pCOS Examples

Assuming a valid TET context and PDF document handle are available, the pCOS functions *PDF_pcos_get_number( )*, *PDF_pcos_get_string( )*, and *PDF_pcos_get_stream( )* can be used to retrieve information from a PDF using the pCOS path syntax. Table 8.1 lists some common pCOS paths and their meaning.

Table 8.1 pCOS paths for commonly used PDF objects

pCOS path	***type***	***explanation***
length:pages	*number*	*number of pages in the document*
/Info/Title	*string*	*document info field Title*
/Root/Metadata	*stream*	*XMP stream with the document's metadata*
fonts[...]/name	*string*	*name of a font; the number of entries can be retrieved with length:fonts.*
fonts[...]/embedded	*boolean*	*embedding status of a font*
pages[...]/width	*number*	*width of the visible area of the page*

Number of pages. The total number of pages in a document can be queried as follows:

```
pagecount = p.pcos_get_number(tet, doc, "length:pages");
```

Document info fields. Document information fields can be retrieved with the following code sequence:

```
objtype = p.pcos_get_string(doc, "type:/Info/Title");
if (objtype.equals("string"))
{
        /* Document info key found */
        title = p.pcos_get_string(doc, "/Info/Title");
}
```

Page size. Although the *MediaBox*, *CropBox*, and *Rotate* entries of a page can directly be obtained via pCOS, they must be evaluated in combination in order to find the actual size of a page. Determining the page size is much easier with the *width* and *height* keys of the *pages* pseudo object. The following code retrieves the width and height of page 3 (note that indices for the *pages* pseudo object start at 0):

```
pagenum = 2
width = p.pcos_get_number(doc, "pages[" + pagenum + "]/width");
height = p.pcos_get_number(doc, "pages[" + pagenum + "]/height");
```

Listing all fonts in a document. The following sequence creates a list of all fonts in a document along with their embedding status:

```
fontcount = p.pcos_get_number(doc, "length:fonts");

for (i=0; i < fontcount; i++)
{
        printf("%s", PDF_pcos_get_string(doc, "fonts[" + i + "]/name"));
        embedded = p.pcos_get_number(doc, "fonts[" + i + "]/embedded");
}
```

Encryption status. You can query the *pcosmode* pseudo object to determine the pCOS mode for the document:

```
if (p.pcos_get_number(doc, "pcosmode") == 2)
{
        /* full pCOS mode */
}
```

XMP meta data. A stream containing XMP meta data can be retrieved with the following code sequence:

```
objtype = p.pcos_get_number(doc, "type:/Root/Metadata");
if (objtype.equals("stream"))
{
        /* XMP meta data found */
        metadata = p.pcos_get_stream(doc, "", "/Root/Metadata");
}
```

8.2 Handling Basic PDF Data Types

pCOS offers the three functions *PDF_pcos_get_number( ), PDF_pcos_get_string( ), and PDF_pcos_get_stream( )*. These can be used to retrieve all basic data types which may appear in PDF documents.

Numbers. Objects of type *integer* and *real* can be queried with *PDF_pcos_get_number( )*. pCOS doesn't make any distinction between integer and floating point numbers.

Names and strings. Objects of type *name* and *string* can be queried with *PDF_pcos_get_string( )*. Name objects in PDF may contain non-ASCII characters and the # syntax (decoration) to include certain special characters. pCOS deals with PDF names as follows:

- Name objects will be undecorated (i.e. the # syntax will be resolved) before they are returned.
- Name objects will be returned as Unicode strings in most language bindings. However, in the C and C++ language bindings they will be returned as UTF-8.

Since the majority of strings in PDF are text strings *PDF_pcos_get_string( )* will treat them as such. However, in rare situations strings in PDF are used to carry binary information. In this case strings should be retrieved with the function *PDF_pcos_get_stream( )* which preserves binary strings and does not modify the contents in any way.

Booleans. Objects of type *boolean* can be queried with *PDF_pcos_get_number( )* and will be returned as 1 (true) or 0 (false). *PDF_pcos_get_string( )* can also be used to query boolean objects; in this case they will be returned as one of the strings *true* and *false*.

Streams. Objects of type *stream* can be queried with *PDF_pcos_get_stream( )*. Depending on the pCOS data type *(stream* or *fstream)* the contents will be compressed or uncompressed. Using the *keepfilter* option of *PDF_pcos_get_string( )* the client can retrieve compressed data even for type *stream*.

Stream data in PDF may be preprocessed with one or more filters. The list of filters present at the stream can be queried from the stream dictionary; for images this information is much easier accessible in the image's *filterinfo* dictionary. If a stream's filter chain contains only supported filters its type will be *stream*. When retrieving the contents of a *stream* object, *PDF_pcos_get_stream( )* will remove all filters and return the resulting unfiltered data.

Note pCOS does not support the following stream filters: CCITTFax, JBIG2, and JPX.

If there is at least one unsupported filter in a stream's filter chain, the object type will be reported as *fstream* (filtered stream). When retrieving the contents of an *fstream* object, *PDF_pcos_get_stream( )* will remove the supported filters at the beginning of a filter chain, but will keep the remaining unsupported filters and return the stream data with the remaining unsupported filters still applied. The list of applied filters can be queried from the stream dictionary, and the filtered stream contents can be retrieved with *PDF_pcos_get_stream( )*. Note that the names of supported filters will not be removed when querying the names of the stream's filters, so the client should ignore the names of supported filters.

8.3 Composite Data Structures and IDs

Objects with one of the basic data types can be arranged in two kinds of composite data structures: arrays and dictionaries. pCOS does not offer specific functions for retrieving composite objects. Instead, the objects which are contained in a dictionary or array can be addressed and retrieved individually.

Arrays. Arrays are one-dimensional collections of any number of objects, where each object may have arbitrary type.

The contents of an array can be enumerated by querying the number *N* of elements it contains (using the *length* prefix in front of the array's path, see Table 8.2), and then iterating over all elements from index 0 to *N-1*.

Dictionaries. Dictionaries (also called associative arrays) contain an arbitrary number of object pairs. The first object in each pair has the type *name* and is called the key. The second object is called the value, and may have an arbitrary type except *null)*.

The contents of a dictionary can be enumerated by querying the number *N* of elements it contains (using the *length* prefix in front of the dictionary's path, see Table 8.2), and then iterating over all elements from index 0 to *N-1*. Enumerating dictionaries will provide all dictionary keys in the order in which they are stored in the PDF using the *.key* suffix at the end of the dictionary's path. Similarly, the corresponding values can be enumerated with the *.val* suffix. Inherited values (see below) and pseudo objects will not be visible when enumerating dictionary keys, and will not be included in the *length* count.

Some page-related dictionary entries in PDF can be inherited across a tree-like data structure, which makes it difficult to retrieve them. For example the *MediaBox* for a page is not guaranteed to be contained in the page dictionary, but may be inherited from an arbitrarily complex page tree. pCOS eliminates this problem by transparently inserting all inherited keys and values into the final dictionary. In other words, pCOS users can assume that all inheritable entries are available directly in a dictionary, and don't have to search all relevant parent entries in the tree. This merging of inherited entries is only available when accessing the pages tree via the *pages[ ]* pseudo object; accessing the */Pages* tree, the *objects[ ]* pseudo object, or enumerating the keys via *pages[ ][ ]* will return the actual entries which are present in the respective dictionary, without any inheritance applied.

pCOS IDs for dictionaries and arrays. Unlike PDF object IDs, pCOS IDs are guaranteed to provide a unique identifier for an element addressed via a pCOS path (since arrays and dictionaries can be nested an object can have the same PDF object ID as its parent array or dictionary). pCOS IDs can be retrieved with the *pcosid* prefix in front of the dictionary's or array's path (see Table 8.2).

The pCOS ID can therefore be used as a shortcut for repeatedly accessing elements without the need for explicit path addressing. For example, this will improve performance when looping over all elements of a large array. Use the *objects[]* pseudo object to retrieve the contents of an element identified by a particular ID.

8.4 Path Syntax

The backbone of the pCOS interface is a simple path syntax for addressing and retrieving any object contained in a PDF document. In addition to the object data itself pCOS can retrieve information about an object, e.g. its type or length. Depending on the object's type (which itself can be queried) one of the functions *PDF_pcos_get_number( )*, *PDF_pcos_get_string( )*, and *PDF_pcos_get_stream( )* can be used to obtain the value of an object. The general syntax for pCOS paths is as follows:

```
[<prefix>:][pseudoname[<index>]]/<name>[<index>]/<name>[<index>] ... [.key|.val]
```

The meaning of the various path components is as follows:

- The optional *prefix* can attain the values listed in Table 8.2.
- The optional *pseudo object name* may contain one of the values in Section 8.5, »Pseudo Objects«, page 187.
- The *name* components are dictionary keys found in the document. Multiple names are separated with a / character. An empty path, i.e. a single / denotes the document's Trailer dictionary. Each name must be a dictionary key present in the preceding dictionary. Full paths describe the chain of dictionary keys from the initial dictionary (which may be the Trailer or a pseudo object) to the target object.
- Paths or path components specifying an array or dictionary can have a numerical index which must be specified in decimal format between brackets. Nested arrays or dictionaries can be addressed with multiple index entries. The first entry in an array or dictionary has index 0.
- Paths or path components specifying a dictionary can have an index qualifier plus one of the suffixes *.key* or *.val*. This can be used to retrieve a particular dictionary key or the corresponding value of the indexed dictionary entry, respectively. If a path for a dictionary has an index qualifier it must be followed by one of these suffixes.

When a path component contains any of the characters */, [,]*, or #, these must be expressed by a number sign # followed by a two-digit hexadecimal number.

Path prefixes. Prefixes can be used to query various attributes of an object (as opposed to its actual value). Table 8.2 lists all supported prefixes.

The *length* prefix and content enumeration via indices are only applicable to plain PDF objects and pseudo objects of type *array*, but not any other pseudo objects. The *pcosid* prefix cannot be applied to pseudo objects. The *type* prefix is supported for all pseudo objects.

Table 8.2 pCOS path prefixes

prefix	***explanation***
length	*(Number) Length of an object, which depends on the object's type:* ***array*** *Number of elements in the array* ***dict*** *Number of key/value pairs in the dictionary* ***stream*** *Number of key/value pairs in the stream dict (not the stream length; use the* Length *key to determine the length of stream data in bytes)* ***fstream*** *Same as stream* ***other*** *0*
pcosid	*(Number) Unique pCOS ID for an object of type dictionary or array.* *If the path describes an object which doesn't exist in the PDF the result will be -1. This can be used to check for the existence of an object, and at the same time obtaining an ID if it exists.*
type	*(String or number) Type of the object as number or string:* ***0, null*** *Null object or object not present (use to check existence of an object)* ***1, boolean*** *Boolean object* ***2, number*** *Integer or real number* ***3, name*** *Name object* ***4, string*** *String object* ***5, array*** *Array object* ***6, dict*** *Dictionary object (but not stream)* ***7, stream*** *Stream object which uses only supported filters* ***8, fstream*** *Stream object which uses one or more unsupported filters*

8.5 Pseudo Objects

Pseudo objects extend the set of pCOS paths by introducing some useful elements which can be used as an abbreviation for information which is present in the PDF, but cannot easily be accessed by reading a single value. The following sections list all supported pseudo objects. Pseudo objects of type *dict* can not be enumerated.

Universal pseudo objects. Universal pseudo objects are always available, regardless of encryption and passwords. This assumes that a valid document handle is available, which may require setting the option *requiredmode* suitably in *PDF_open_document( )*. Table 8.3 lists all universal pseudo objects.

Table 8.3 Universal pseudo objects

object name	***explanation***
encrypt	*(Dict) Dictionary with keys describing the encryption status of the document:* ***length*** *(Number) Length of the encryption key in bits* ***algorithm*** *(Number)* ***description*** *(String) Encryption algorithm number or description:* *-1 Unknown encryption* *0 No encryption* *1 40-bit RC4 (Acrobat 2-4)* *2 128-bit RC4 (Acrobat 5)* *3 128-bit RC4 (Acrobat 6)* *4 128-bit AES (Acrobat 7)* *5 Public key on top of 128-bit RC4 (Acrobat 5) (unsupported)* *6 Public key on top of 128-bit AES (Acrobat 7) (unsupported)* *7 Adobe Policy Server (Acrobat 7) (unsupported)* ***master*** *(Boolean)* `True` *if the PDF requires a master password to change security settings (permissions, user or master password),*`false` *otherwise* ***user*** *(Boolean)* `True` *if the PDF requires a user password for opening,* `false` *otherwise* ***noaccessible, noannots, noassemble, nocopy, noforms, nohiresprint, nomodify, noprint*** *(Boolean)* `True` *if the respective access protection is set,* `false` *otherwise* ***plainmetadata*** *(Boolean)* `True` *if the PDF contains unencrypted meta data,* `false` *otherwise*
filename	*(String) Name of the PDF file as supplied to PDF_pcos__open_document( )*
filesize	*(Number) Size of the PDF file in bytes*
linearized	*(Boolean)* `True` *if the PDF document is linearized,* `false` *otherwise*
major ***minor*** ***revision***	*(Number) Major, minor, or revision number of the library, respectively.*
pcosinterface	*(Number) Interface number of the underlying pCOS implementation. This specification describes interface number 3. The following table details which product versions implement various pCOS interface numbers:* ***1*** *TET 2.0, 2.1* ***2*** *pCOS 1.0* ***3*** *PDFlib+PDI 7, PPS 7,*

Table 8.3 Universal pseudo objects

object name	explanation
pcosmode ***pcosmodename***	*(Number/string) pCOS mode as number or string:* ***0*** *minimum* ***1*** *restricted* ***2*** *full*
pdfversion	*(Number) PDF version number multiplied by 10, e.g. 16 for PDF 1.6.*
version	*(String) Full library version string in the format <major>.<minor>.<revision>, possibly suffixed with additional qualifiers such as beta, rc, etc.*

Object, page and hypertext pseudo objects. Table 8.4 lists pseudo objects which can be used for retrieving object or page information, or serve as shortcuts for various hypertext elements.

Table 8.4 Object, page, and hypertext pseudo objects

object name	***explanation***
articles	*(Array of dicts) Array containing the article thread dictionaries for the document. The array will have length 0 if the document does not contain any article threads. In addition to the standard PDF keys pCOS supports the following pseudo key for dictionaries in the articles array:* ***beads*** *(Array of dicts) Bead directory with the standard PDF keys, plus the following:* ***destpage*** *(Number) Number of the target page (first page is 1)*
bookmarks	*(Array of dicts) Array containing the bookmark (outlines) dictionaries for the document. In addition to the standard PDF keys pCOS supports the following pseudo keys for dictionaries in the bookmarks array:* ***level*** *(Number) Indentation level in the bookmark hierarchy* ***destpage*** *(Number) Number of the target page (first page is 1) if the bookmark points to a page in the same document, -1 otherwise.*
fields	*(Array of dicts) Array containing the form fields dictionaries for the document. In addition to the standard PDF keys in the field dictionary and the entries in the associated Widget annotation dictionary pCOS supports the following pseudo keys for dictionaries in the fields array:* ***level*** *(Number) Level in the field hierarchy (determined by ».« as separator)* ***fullname*** *(String) Complete name of the form field. The same naming conventions as in Acrobat 7 will be applied.*
names	*(Dict) A dictionary where each entry provides simple access to a name tree. The following name trees are supported:* `AP`, `AlternatePresentations`, `Dests`, `EmbeddedFiles`, `IDS`, `JavaScript`, `Pages`, `Renditions`, `Templates`, `URLS`. *Each name tree can be accessed by using the name as a key to retrieve the corresponding value, e.g.:* `names/Dests[0].key` *retrieves the name of a destination* `names/Dests[0].val` *retrieves the corresponding destination dictionary* *In addition to standard PDF dictionary entries the following pseudo keys for dictionaries in the* `Dests` *names tree are supported:* ***destpage*** *(number) Number of the target page (first page is 1) if the destination points to a page in the same document, -1 otherwise.* *In order to retrieve other name tree entries these must be queried directly via* `/Root/Names/Dests` *etc. since they are not present in the name tree pseudo objects.*
objects	*(Array) Address an element for which a pCOS ID has been retrieved earlier using the pcosid prefix. The ID must be supplied as array index in decimal form; as a result, the PDF object with the supplied ID will be addressed. The length prefix cannot be used with this array.*

Table 8.4 Object, page, and hypertext pseudo objects

object name	explanation
pages	(Array of dicts) Each array element addresses a page of the document. Indexing it with the decimal representation of the page number minus one addresses that page (the first page has index 0). Using the length prefix the number of pages in the document can be determined. A page object addressed this way will incorporate all attributes which are inherited via the `/Pages` tree. The entries `/MediaBox` and `/Rotate` are guaranteed to be present. In addition to standard PDF dictionary entries the following pseudo entries are available for each page: **colorspaces, extgstates, fonts, images, patterns, properties, shadings, templates** (Arrays of dicts) High-level page resources according to Table 8.5. **annots** (Array of dicts) In addition to the standard PDF keys pCOS supports the following pseudo key for dictionaries in the `annots` array: **destpage** (Number; only for `Subtype=Link` and if a `Dest` entry is present) Number of the target page (first page is 1) **height** (Number) Height of the page. The MediaBox or the CropBox (if present) will be used to determine the height. Rotate entries will also be applied. **isempty** (Boolean) `True` if the page is empty, and `false` if the page is not empty **label** (String) The page label of the page (including any prefix which may be present). Labels will be displayed as in Acrobat. If no label is present (or the PageLabel dictionary is malformed), the string will contain the decimal page number. Roman numbers will be created in Acrobat's style (e.g. VL), not in classical style which is different (e.g. XLV). If `/Root/PageLabels` doesn't exist, the document doesn't contain any page labels. **width** (Number) Width of the page (same rules as for height) The following entries will be inherited: `CropBox`, `MediaBox`, `Resources`, `Rotate`.
pdfa	(String) PDF/A conformance level of the document (e.g. `PDF/A-1a:2005`) or `none`
pdfx	(String) PDF/X conformance level of the document (e.g. `PDF/X-1a:2001`) or `none`
tagged	(Boolean) `True` if the PDF document is tagged, `false` otherwise

Pseudo objects for simplified resource handling. Resources are a key concept for managing various kinds of data which are required for completely describing the contents of a page. The resource concept in PDF is very powerful and efficient, but complicates access with various technical concepts, such as recursion and resource inheritance. pCOS greatly simplifies resource retrieval and supplies several groups of pseudo objects which can be used to directly query resources. In addition, some of these pseudo resource dictionaries contain entries in addition to the standard PDF keys in order to further simplify resource information retrieval.

pCOS supports two groups of pseudo objects for resource retrieval. Global resource arrays contain all resources in a PDF document, while page resources contain only the resources used by a particular page. The resource entries in the global and page-based resource arrays reflect resources from the user's point of view. They differ from native PDF resources in several ways:

- Some entries may be added (e.g. inline images, simple color spaces) or deleted (e.g. the parts of multi-strip images).
- In addition to the original PDF dictionary keys resource dictionaries may contain some user-friendly keys for auxiliary information (e.g. embedding status of a font, number of components of a color space).

The following list details the two categories using the *images* resource type as an example; the same scheme applies to all resource types listed in Table 8.5:

- A list of image resources in the document is available in *images[ ].*
- A list of image resources on each page is available in *pages[ ]/images[ ].*

Table 8.5 Pseudo objects for resource retrieval; each pseudo object P in this table creates two arrays with high-level resources `P[ ]` *and* `pages[ ]/P[ ]`.

object name	***explanation***
colorspaces	*(Array of dicts) Array containing dictionaries for all color spaces on the page or in the document. In addition to the standard PDF keys in color space and ICC profile stream dictionaries the following pseudo keys are supported:* ***alternateid*** *(Integer; only for* `name=Separation` *and* `DeviceN`*) Index of the underlying alternate color space in the* `colorspaces[]` *pseudo object.* ***alternateonly*** *(Boolean) If* `true`*, the colorspace is only used as the alternate color space for (one or more) Separation or DeviceN color spaces, but not directly.* ***baseid*** *(Integer; only for* `name=Indexed`*) Index of the underlying base color space in the* `colorspaces[]` *pseudo object.* ***colorantname*** *(Name; only for* `name=Separation`*) Name of the colorant* ***colorantnames*** *(Array of names; only for* `name=DeviceN`*) Names of the colorants* ***components*** *(Integer) Number of components of the color space* ***name*** *(String) Name of the color space* ***csarray*** *(Array; not for* `name=DeviceGray/RGB/CMYK`*) Array describing the underlying native color space.* *High-level color space resources will include all color spaces which are referenced from any type of object, including the color spaces which do not require any native PDF resources (i.e.* `DeviceGray`, `DeviceRGB`, *and* `DeviceCMYK`*).*

Table 8.5 Pseudo objects for resource retrieval; each pseudo object P in this table creates two arrays with high-level resources `P[ ]` and `pages[ ]/P[ ]`.

object name	explanation
extgstates	(Array of dicts) Array containing the dictionaries for all extended graphics states (ExtGstates) on the page or in the document
fonts	(Array of dicts) Array containing dictionaries for all fonts on the page or in the document. In addition to the standard PDF keys in font dictionaries, the following pseudo keys are supported: **name** (String) PDF name of the font without any subset prefix. Non-ASCII CJK font names will be converted to Unicode. **embedded** (Boolean) Embedding status of the font **type** (String) Font type **vertical** (Boolean) `true` for fonts with vertical writing mode, `false` otherwise
images	(Array of dicts) Array containing dictionaries for all images on the page or in the document. High-level image resources will include all image XObjects and inline images, while native PDF resources contain only image XObjects. In addition to the standard PDF keys the following pseudo keys are supported: **bpc** (Integer) The number of bits per component. This entry is usually the same as BitsPerComponent, but unlike this it is guaranteed to be available. **colorspaceid** (Integer) Index of the image's color space in the `colorspaces[]` pseudo object. This can be used to retrieve detailed color space properties. **filterinfo** (Dict) Describes the remaining filter for streams with unsupported filters or when retrieving stream data with the `keepfilter` option set to `true` (see PDF_pcos__get_stream( )). If there is no such filter no filterinfo dictionary will be available. The dictionary contains the following entries: **name** (Name) Name of the filter **supported** (Boolean) `True` if the filter is supported **decodeparms** (Dict) The `DecodeParms` dictionary if one is present for the filter **maskid** (Integer) Index of the image's mask in the `images[]` pseudo object if the image is masked, otherwise -1 **maskonly** (Boolean) If `true`, the image is only used as a mask for (one or more) other images, but not directly
patterns	(Array of dicts) Array containing dictionaries for all patterns on the page or in the document
properties	(Array of dicts) Array containing dictionaries for all properties on the page or in the document
shadings	(Array of dicts) Array containing dictionaries for all shadings on the page or in the document. In addition to the standard PDF keys in shading dictionaries the following pseudo key is supported: **colorspaceid** (Integer) Index of the underlying color space in the `colorspaces[]` pseudo object.
templates	(Array of dicts) Array containing dictionaries for all templates (Form XObjects) on the page or in the document

8.6 Encrypted PDF Documents

pCOS supports encrypted and unencrypted PDF documents as input. However, full object retrieval for encrypted documents requires the appropriate master password to be supplied when opening the document. Depending on the availability of user and master password, encrypted documents can be processed in one of several pCOS modes:

Full pCOS mode (mode 0): Encrypted PDFs can be processed without any restriction provided the master password has been supplied upon opening the file. All objects will be returned unencrypted. Unencrypted documents will always be opened in full pCOS mode.

Restricted pCOS mode (mode 1). If the document has been opened without the appropriate master password and does not require a user password (or the user password has been supplied) pCOS operations are subject to the following restriction: The contents of objects with type *string*, *stream*, or *fstream* can not be retrieved with the following exceptions:

- The objects */Root/Metadata* and */Info/** (document info keys) can be retrieved if *nocopy=false* or *plainmetadata=true*.
- The objects *bookmarks[...]/Title* and *annots[...]/Contents* (bookmark and annotation contents) can be retrieved if *nocopy=false*, i.e. if text extraction is allowed for the main text on the pages.

Minimum pCOS mode (mode 2). Regardless of the encryption status and the availability of passwords, the universal pCOS pseudo objects listed in Table 8.3 are always available. For example, the *encrypt* pseudo object can be used to query a document's encryption status. Encrypted objects can not be retrieved in minimum pCOS mode.
Table 8.6 lists the resulting pCOS modes for various password combinations. Depending on the document's encryption status and the password supplied when opening the file, PDF object paths may be available in minimum, restricted, or full pCOS mode. Trying to retrieve a pCOS path which is inappropriate for the respective mode will raise an exception.

Table 8.6 Required passwords for various pCOS operations on encrypted documents

If you know...	***...pCOS will run in...***
none of the passwords	*restricted pCOS mode if no user password is set, minimum pCOS mode otherwise*
only the user password	*restricted pCOS mode*
the master password	*full pCOS mode*

9 Generating various PDF Flavors

9.1 Acrobat and PDF Versions

At the user's option PDFlib generates output according to PDF 1.3 (Acrobat 4), PDF 1.4 (Acrobat 5), PDF 1.5 (Acrobat 6), or PDF 1.6 (Acrobat 7). In addition, PDF 1.7 (Acrobat 8) files can be created, but at this time no specific features of PDF 1.7 are supported. The PDF output version can be controlled with the *compatibility* option in *PDF_begin_document( )*.

PDFlib output features for PDF 1.4 or above. In PDF 1.3 compatibility mode the PDFlib features for higher PDF versions are not available. Trying to use one of these features in PDF 1.3 mode will result in an exception.

Table 9.1 PDFlib features for PDF 1.4 which are not available in PDF 1.3 compatibility mode

Feature	***PDFlib API functions and parameters***
smooth shadings (color blends)	*PDF_shading_pattern( ), PDF_shfill( ), PDF_shading( )*
soft masks	*PDF_load_image( ) with the* masked *option referring to an image with more than 1 bit pixel depth*
128-bit encryption	*PDF_begin_document( ) with the* userpassword, masterpassword, permissions *options*
extended permission settings	*PDF_begin_document( ) with* permissions *option, see Table 9.4*
certain CMaps for CJK fonts	*PDF_load_font( ), see Table 4.5*
transparency and other graphics state options	*PDF_create_gstate( ) with options* alphaisshape, blendmode, opacityfill, opacitystroke, textknockout
certain options for actions	*PDF_create_action( )*
certain options for annotations	*PDF_create_annotation( )*
certain field options	*PDF_create_field( ) and PDF_create_fieldgroup( )*
Tagged PDF	tagged *option in PDF_begin_document( )*

PDFlib output features for PDF 1.5 or above. In PDF 1.3 or 1.4 compatibility modes the PDFlib features for higher PDF versions are not available. Trying to use one of these features in PDF 1.3 or PDF 1.4 mode will result in an exception.

Table 9.2 PDFlib features for PDF 1.5 which are not available in lower compatibility modes

Feature	***PDFlib API functions and parameters***
certain field options	*PDF_create_field( ) and PDF_create_fieldgroup( )*
page layout	*PDF_begin/end_document( ): option* pagelayout=twopageleft/right
certain annotation options	*PDF_create_annotation( )*
extended permission settings	permissions=plainmetadata *in PDF_begin_document( ), see Table 9.4*
certain CMaps for CJK fonts	*PDF_load_font( ), see Table 4.5*

Table 9.2 PDFlib features for PDF 1.5 which are not available in lower compatibility modes

Feature	***PDFlib API functions and parameters***
Tagged PDF	*certain options for PDF_begin_item( )* *PDF_begin/end_page_ext( ): option* taborder
Layers	*PDF_define_layer( ), PDF_begin_layer( ), PDF_end_layer( ), PDF_layer_dependency( )*
JPEG2000 images	imagetype=jpeg2000 *in PDF_load_image( )*

PDFlib output features for PDF 1.6 or above. In PDF compatibility modes up to PDF 1.5 the PDFlib features for PDF 1.6 are not available. Trying to use one of these features in lower PDF modes will result in an exception.

Table 9.3 PDFlib features for PDF 1.6 which are not available in lower compatibility modes

Feature	***PDFlib API functions and options***
user units	*PDF_begin/end_document( ): option* userunit
print scaling	*PDF_begin/end_document( ): suboption* printscaling *for* viewerpreferences *option*
document open mode	*PDF_begin/end_document( ): option* openmode=attachments
AES encryption	*PDF_begin_document( ): AES encryption will automatically be used with* compatibility=1.6 *or above when the* masterpassword, userpassword, *or* permissions *option is supplied*
attachment description	*PDF_begin/end_document( ): suboption* description *for option* attachments
embed U3D models	*PDF_load_3ddata( ), PDF_create_3dview( ); PDF_create_annotation( ):* type=3D

PDF version of documents imported with PDI. In all compatibility modes only PDF documents with the same or a lower compatibility level can be imported with PDI. If you must import a PDF with a newer level you must set the *compatibility* option accordingly (see Section 6.2.3, »Acceptable PDF Documents«, page 132.

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 do this with Acrobat; the details depend on the version of Acrobat that you are using:

- Acrobat 7 Professional: You can save the file in the formats PDF 1.3 - PDF 1.6 using *Advanced, PDF Optimizer, Make compatible with*.
- Acrobat 6 and Acrobat 7 Standard: You can save the file as PDF 1.3 - PDF 1.5 using *File, Reduce File Size...* .
- Acrobat 5: For converting the document to PDF 1.3 use an additional plugin by callas software called *pdfSaveAs1.3*. Fully functional demo versions are available on the callas web site[1]. This conversion plugin is especially useful when dealing with blocks and PDF/X, since some versions of PDF/X require PDF 1.3 (see »Using Blocks with PDF/X«, page 232, and Section 9.4.2, »Generating PDF/X-conforming Output«, page 202).

[1] *See www.callassoftware.com*

9.2 Encrypted PDF

9.2.1 Strengths and Weaknesses of PDF Security

PDF supports various security features which aid in protecting document contents. They are based on Acrobat's standard encryption handler which uses symmetric encryption. Both Acrobat Reader and the full Acrobat product support the following security features:

- Permissions restrict certain actions for the PDF document, such as printing or extracting text.
- The user password is required to open the file.
- The master password is required to change any security settings, i.e. permissions, user or master password. Files with user and master passwords can be opened for reading or printing with either password.
- 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.

Cracking protected PDF documents. The length of the encryption keys used for protecting documents depends on the PDF compatibility level:

- For PDF versions up to and including 1.3 (i.e., Acrobat 4) the key length is 40 bits.
- For PDF version 1.4 and above the key length is 128 bits. This requires Acrobat 5 or above. For PDF 1.5 the key length will also be 128 bits, but a slightly different encryption method will be used, which requires Acrobat 6.
- For PDF 1.6 supports AES (Advanced Encryption Standard) with 128-bit keys. This requires Acrobat 7.

It is widely known that a key length of 40 bits for symmetrical encryption (as used in PDF) is not secure. Actually, using commercially available cracking software it is possible to disable 40-bit PDF security settings with a brute-force attack within days or weeks, depending on the length and quality of the password. For maximum security we recommend the following:

- Use 128-bit encryption (i.e., PDF 1.4 compatibility setting) if at all possible. This requires Acrobat 5 or above for all users of the document.
- Use AES encryption unless your users work with older versions than Acrobat 7.
- Passwords should be at least six characters long and should contain non-alphabetic characters. Passwords should definitely not resemble your spouse's or pet's name, your birthday etc. in order to prevent so-called dictionary attacks or password guessing. It is important to mention that even with 128-bit encryption short passwords can be cracked within minutes.

Access permissions. Setting some access restriction, such as *printing prohibited* will disable the respective function in Acrobat. However, this not necessarily holds true for third-party PDF viewers or other software. It is up to the developer of PDF tools whether or not access permissions will be honored. Indeed, several PDF tools are known to ignore permission settings altogether; commercially available PDF cracking tools can be used to disable any access restrictions. This has nothing to do with cracking the encryption; there is simply no way that a PDF file can make sure it won't be printed while it still remains viewable. This is actually documented in Adobe's own PDF reference:

There is nothing inherent in PDF encryption that enforces the document permissions specified in the encryption dictionary. It is up to the implementors of PDF viewers to respect the intent of the document creator by restricting user access to an encrypted PDF file according to the permissions contained in the file.

9.2.2 Protecting Documents with PDFlib

Encryption algorithm and key length. When creating protected documents PDFlib will choose 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. This requires Acrobat 5 or above.
- 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.

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.

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- or 128-bit keys, on the user level passwords of up to 32 characters are used. The internal key which is used to encrypt the PDF document is derived from the user-supplied password via by applying some complicated calculations. If the password is weak, the resulting protection will be weak as well, regardless of the key length. Even 128-bit keys and AES encryption are not very secure if short passwords are used.

Non-ASCII characters in passwords. Attention must be paid when characters outside the range 0x20-0x7E are used in passwords, i.e. characters which are not in the traditional ASCII character set. As an example, let's take a look at the use of the character Ä within a password. On the Mac this character has code 0x80, while on Windows it is encoded as 0xC4. Since users expect the file to be opened when using the password Ä on either platform, Acrobat converts the supplied password to an internal encoding (called PDFDocEncoding) before applying the password. Characters which are not available in this encoding will be mapped to the *space* character. PDFDocEncoding contains all characters of the Mac and Windows platforms, but requires several characters to be converted. In the example above, when the user encrypts the file with password Ä on the Mac, PDI would be unable to decrypt the file if the code for Ä would be used directly. PDI therefore applies the same password conversion as Acrobat in order to make sure that files encrypted with Mac or Windows versions of Acrobat can successfully be decrypted. Upon decryption PDI will automatically detect the required conversion:

- WinAnsi to PDFDocEncoding conversion if the document was encrypted with Acrobat on Windows or with PDFlib PLOP 2.1 or above;
- MacRoman to PDFDocEncoding conversion if the document was encrypted with Acrobat on the Mac;
- No conversion if the document was encrypted with some other software, including PDFlib PLOP 2.0 (but not any newer versions).

When encrypting files, PDFlib will act like Acrobat on Windows and interpret the supplied passwords in WinAnsi encoding, i.e., it will apply a WinAnsi to PDFDocEncoding conversion to the supplied user and master passwords; on EBCDIC platforms it will apply EBCDIC to WinAnsi conversion prior to that.

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=abc123 permissions={noprint nocopy}");
```

Table 9.4 lists all supported access restriction keywords. As detailed in the table, some keywords require PDF 1.4 or higher compatibility. They will be rejected if the PDF output version is too low.

Table 9.4 Access restriction keywords for the permissions option in PDF_begin_document( )

keyword	***explanation***
noprint	*Acrobat will prevent printing the file.*
nomodify	*Acrobat will prevent users from adding form fields or making any other changes.*
nocopy	*Acrobat will prevent copying and extracting text or graphics, and will disable the accessibility interface*
noannots	*Acrobat will prevent adding or changing comments or form fields.*
noforms	*(PDF 1.4) Acrobat will prevent form field filling, even if noannots hasn't been specified.*
noaccessible	*(PDF 1.4) Acrobat will prevent extracting text or graphics for accessibility purposes (such as a screenreader program)*
noassemble	*(PDF 1.4) Acrobat will prevent inserting, deleting, or rotating pages and creating bookmarks and thumbnails, even if nomodify hasn't been specified.*

Table 9.4 Access restriction keywords for the permissions option in PDF_begin_document( )

keyword	explanation
nohiresprint	(PDF 1.4) Acrobat will prevent high-resolution printing. If noprint hasn't been specified printing is restricted to the »print as image« feature which prints a low-resolution rendition of the page.
plainmetadata	(PDF 1.5) Keep document metadata unencrypted even for encrypted documents. This relates only to XMP metadata, but not to document info fields.

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.

9.3 Web-Optimized (Linearized) PDF

PDFlib can apply a process called linearization to PDF documents (linearized PDF is called *Optimized* in Acrobat 4, and *Fast Web View* in Acrobat 5 and above). Linearization reorganizes the objects within a PDF file and adds supplemental information which can be used for faster access.

While non-linearized PDFs must be fully transferred to the client, a Web server can transfer linearized PDF documents one page at a time using a process called byteserving. It allows Acrobat (running as a browser plugin) to retrieve individual parts of a PDF document separately. The result is that the first page of the document will be presented to the user without having to wait for the full document to download from the server. This provides enhanced user experience.

Note that the Web server streams PDF data to the browser, not PDFlib. Instead, PDFlib prepares the PDF files for byteserving. All of the following requirements must be met in order to take advantage of byteserving PDFs:

- The PDF document must be linearized, which can be achieved with the *linearize* option in *PDF_begin_document( )*. In Acrobat you can check whether a file is linearized by looking at its document properties (»Fast Web View: yes«).
- The Web server must support byteserving. The underlying byterange protocol is part of HTTP 1.1 and therefore implemented in all current Web servers. In particular, the following Web servers support byteserving:

  ```
  Microsoft Internet Information Server (IIS) 3.0 and above
  Apache 1.2.1 and above; however, Apache 1.3.14 (but not other versions) has a bug
  which prevents byteserving
  ```

- The user must use Acrobat as a Browser plugin, and have page-at-a-time download enabled in Acrobat (Acrobat 6/7: *Edit, Preferences, [General...,] Internet, Allow fast web view;* Acrobat 5: *Edit, Preferences, General..., Options, Allow Fast Web view)*. Note that this is enabled by default.

The larger a PDF file (measured in pages or MB), the more it will benefit from linearization when delivered over the Web.

Note Linearizing a PDF document generally slightly increases its file size due to the additional linearization information.

Temporary storage requirements for linearization. PDFlib must create the full document before it can be linearized; the linearization process will be applied in a separate step after the document has been created. For this reason PDFlib has additional storage requirements for linearization. Temporary storage will be required which has roughly the same size as the generated document (without linearization). Subject to the *inmemory* option in *PDF_begin_document( )* PDFlib will place the linearization data either in memory or on a temporary disk file.

9.4 PDF/X for Print Production

9.4.1 The PDF/X Family of Standards

The PDF/X formats specified in the ISO 15930 standards family strive to provide a consistent and robust subset of PDF which can be used to deliver data suitable for commercial printing[1]. PDFlib can generate output and process input conforming to the following flavors of PDF/X:

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) are 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-2:2003 as defined in ISO 15930-5. This standard is targeted at »partial exchange« which requires more discussion between supplier and receiver of a file. PDF documents according to this standard can reference external entities (point to other PDF pages external to the current document). PDF/X-2:2003 is based on PDF 1.4. As a superset of PDF/X-3:2003 it supports device independent colors.

PDF/X-3:2002 as defined in ISO 15930-3. This standard is based on PDF 1.3, and supports modern workflows based on device-independent color in addition to grayscale, CMYK, and spot colors. It is especially popular in European countries. Output devices can be monochrome, RGB, or CMYK.

PDF/X-3:2003 as defined in ISO 15930-6. This standard is the successor to PDF/X-3:2002. It is based on PDF 1.4, with some features (e.g. transparency) prohibited.

When one of the PDF/X standards is referenced below without any standardization year, all versions of the respective standard are meant. For example, *PDF/X-3* means *PDF/X-3:2002* and *PDF-X/3:2003*.

9.4.2 Generating PDF/X-conforming Output

Creating PDF/X-conforming output with PDFlib is achieved by the following means:

- PDFlib will automatically take care of several formal settings for PDF/X, such as PDF version number and PDF/X conformance keys.
- The PDFlib client must explicitly use certain function calls or parameter settings as detailed in Table 9.5.
- The PDFlib client must refrain from using certain function calls and parameter settings as detailed in Table 9.6.

[1] It is highly recommended to read the PDF/X FAQ at www.globalgraphics.com/products/pdfx/index.html

- Additional rules apply when importing pages from existing PDF/X-conforming documents (see Section 9.4.3, »Importing PDF/X Documents with PDI«, page 205).

Required operations. Table 9.5 lists all operations required to generate PDF/X-compatible output. The items apply to all PDF/X conformance levels unless otherwise noted. Not calling one of the required functions while in PDF/X mode will trigger an exception.

Table 9.5 Operations which must be applied for PDF/X compatibility

item	***PDFlib function and parameter requirements for PDF/X compatibility***
conformance level	*The* `pdfx` *option in PDF_begin_document( ) must be set to the required PDF/X conformance level.*
output condition (output intent)	*PDF_load_iccprofile( ) with* `usage=outputintent` *or PDF_process_pdi( ) with* `action=copyoutputintent` *(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: 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 sizes	*The page boxes, which are settable via the CropBox, BleedBox, TrimBox, and ArtBox parameters, must satisfy all of the following requirements:* ► *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.*
grayscale color	*PDF/X-3: the* `defaultgray` *option in PDF_begin_page_ext( ) must be set if grayscale images are used or PDF_setcolor( ) is used with a gray color space, and the PDF/X output condition is not a CMYK or grayscale device.*
RGB color	*PDF/X-3: the* `defaultrgb` *option in PDF_begin_page_ext( ) must be set if RGB images are used or PDF_setcolor( ) is used with an RGB color space, and the PDF/X output condition is not an RGB device.*
CMYK color	*PDF/X-3: the* `defaultcmyk` *option in PDF_begin_page_ext( ) must be set if CMYK images are used or PDF_setcolor( ) is used with a CMYK color space, and the PDF/X output condition is not a CMYK device.*
document info keys	*The* `Creator` *and* `Title` *info keys must be set with PDF_set_info( ).*

Prohibited operations. Table 9.6 lists all operations which are prohibited when generating PDF/X-compatible output. The items apply to all PDF/X conformance levels unless otherwise noted. Calling one of the prohibited functions while in PDF/X mode will trigger an exception. Similarly, if an imported PDF page doesn't match the current PDF/X conformance level, the corresponding PDI call will fail

Table 9.6 Operations which must be avoided to achieve PDF/X compatibility

item	***PDFlib functions and parameters to be avoided for PDF/X compatibility***
grayscale color	*PDF/X-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* `defaultcmyk` *option in PDF_begin_page_ext( ) must be avoided.*

Table 9.6 Operations which must be avoided to achieve PDF/X compatibility

item	PDFlib functions and parameters to be avoided for PDF/X compatibility
ICC-based color	PDF/X-1a: the `iccbasedgray/rgb/cmyk` color space in PDF_setcolor( ) and the `setcolor:iccprofilegray/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( ) and related deprecated functions.
actions and JavaScript	All actions including JavaScript must be avoided: PDF_create_action( ), and related deprecated functions
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. The `OPI-1.3` and `OPI-2.0` options in PDF_load_image( ) must be avoided.
transparency	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. The `opacityfill` and `opacitystroke` options for PDF_create_gstate( ) must be avoided unless they have a value of 1.
viewer preferences / view and print areas	When the `viewarea`, `viewclip`, `printarea`, and `printclip` keys are used for PDF_set_parameter( ) values other than media or bleed are not allowed.
document info keys	Trapped info key values other than `True` or `False` for PDF_set_info( ) must be avoided.
security	PDF/X-1a and PDF/X-3: `userpassword`, `masterpassword`, and `permissions` options in PDF_begin_document( ) must be avoided.
PDF version / compatibility	PDF/X-1a:2001 and PDF/X-3:2002 are based on PDF 1.3. Operations that require PDF 1.4 or above (such as transparency or soft masks) must be avoided. PDF/X-1a:2003, PDF/X-2:2003, and PDF/X-3:2003 are based on PDF 1.4. Operations that require PDF 1.5 (such as layers) must be avoided.
PDF import (PDI)	Imported documents must conform to a compatible PDF/X level according to Table 9.8, and must have been prepared according to the same output intent.

Standard output conditions. The output condition defines the intended target device, which is mainly useful for reliable proofing. The output intent can either be specified by an ICC profile or by supplying the name of a standard output intent. The standard output intents are known internally to PDFlib (see PDFlib Reference for a complete list of the names and the corresponding printing conditions). Standard output intents can be referenced as follows:

```
p.load_iccprofile("CGATS TR 001", "usage=outputintent");
```

When creating PDF/X-3 output and using any of HKS, Pantone, ICC-based, or Lab colors the use of standard output intents is not allowed, but an ICC profile of the output device must be embedded instead.

Additional standard output intents can be defined using the *StandardOutputIntent* resource category (see Section 3.1.3, »Resource Configuration and File Searching«, page 46). It is the user's responsibility to add only those names as standard output intents which can be recognized by PDF/X-processing software.

Selecting 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 or publisher 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 9.7 as a starting point (taken from the PDF/X FAQ).

Table 9.7 Suitable PDF/X output intents for common printing situations

	Europe	***North America***
Magazine ads	*FOGRA28*	*CGATS TR 001 (SWOP)*
Newsprint ads	*IFRA26*	*IFRA30*
Sheet-fed offset	*Dependent on paper stock:* *Types 1 & 2 (coated): FOGRA27* *Type 3 (LWC): FOGRA28* *Type 4 (uncoated): FOGRA29*	*Dependent on paper stock:* *Grades 1 and 2 (premium coated): FOGRA27* *Grade 5: CGATS TR 001 (SWOP)* *Uncoated: FOGRA29*
Web-fed offset	*Dependent on paper stock:* *Type 1 & 2 (coated): FOGRA28* *Type 4 (uncoated, white): FOGRA29* *Type 5 (uncoated, yellowish): FOGRA30*	*Dependent on paper stock:* *Grade 5: CGATS TR 001 (SWOP)* *Uncoated (white): FOGRA29* *Uncoated (yellowish): FOGRA30*

9.4.3 Importing PDF/X Documents with PDI

Special rules apply when pages from an existing PDF document will be imported into a PDF/X-conforming output document (see Section 6.2, »Importing PDF Pages with PDI (PDF Import Library)«, page 130, for details on the PDF import library PDI). All imported documents must conform to an acceptable PDF/X conformance level according to Table 9.8. As a general rule, input documents conforming to the same PDF/X conformance level as the generated output document, or to an older version of the same level, are acceptable. In addition, certain other combinations are acceptable. If a certain PDF/X conformance level is configured in PDFlib and the imported documents adhere to one of the acceptable levels, the generated output is guaranteed to comply with the selected PDF/X conformance level. Imported documents which do not adhere to one of the acceptable PDF/X levels will be rejected.

Table 9.8 Acceptable PDF/X input levels for various PDF/X output levels; other combinations must be avoided.

	PDF/X level of the imported document				
PDF/X output level	***PDF/X-1a:2001***	***PDF/X-1a:2003***	***PDF/X-2:2003***	***PDF/X-3:2002***	***PDF/X-3:2003***
PDF/X-1a:2001	*allowed*				
PDF/X-1a:2003	*allowed*	*allowed*			
PDF/X-2:2003	*allowed*	*allowed*	*allowed*	*allowed*	*allowed*
PDF/X-3:2002	*allowed*			*allowed*	
PDF/X-3:2003	*allowed*	*allowed*		*allowed*	*allowed*

If multiple PDF/X documents are imported, they must all have been prepared for the same output condition. While PDFlib can correct certain items, it is not intended to work as a full PDF/X validator or to enforce full PDF/X compatibility for imported 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:

```
pdfalevel = p.pcos_get_string(doc, "pdfx");
```

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 the *usage* option set to *outputintent*.

9.5 PDF/A for Archiving

9.5.1 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-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 9.6, »Tagged PDF«, page 214).

PDFlib’s PDF/A implementation is based on the ISO 19005-1 standard plus the 2006 corrigendum. When PDF/A (without any conformance level) is mentioned below, both conformance levels are meant.

PDF/A Competence Center. PDFlib GmbH is a founding member of the PDF/A Competence Center. 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 conducts seminars and conference on the subject. For more information refer to the PDF/A Competence Center web site[1].

9.5.2 Generating PDF/A-conforming Output

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 9.9.
- The PDFlib client program must refrain from using certain function calls and option settings as detailed in Table 9.10.

[1] *See www.pdfa.org*

- Additional rules apply when importing pages from existing PDF/A-conforming documents (see Section 9.5.3, »Importing PDF/A Documents with PDI«, page 210).

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 9.9 lists all operations required to generate PDF/A-compatible 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 9.9 Operations which must be applied for PDF/A-1 level A and B compatibility

item	***PDFlib function and parameter requirements for PDF/A compatibility***
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=copyoutputintent *(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. Unlike PDF/X, standard output conditions are not sufficient; an output device ICC profile always must be embedded (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.*
grayscale color	*A gray, RGB, or CMYK ICC profile must be set as PDF/A output condition if grayscale color is used in the document.*
RGB color	*An RGB ICC profile must be set as PDF/A output condition if RGB color is used in the document.*
CMYK color	*A CMYK ICC profile must be set as PDF/A output condition if CMYK color is used in the document.*
document info	*The* Creator *and* Title *keys must be set with PDF_set_info( ), or the corresponding elements must be supplied as XMP to the* metadata *option of PDF_begin/end_document( ). If PDF_set_info( ) is used, PDFlib will automatically create the required XMP entries.*

Prohibited and restricted operations. Table 9.10 lists all operations which are prohibited when generating PDF/A-compatible 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 is not compatible to the current PDF/A output level, the corresponding PDI call will fail.

Table 9.10 Operations which must be avoided or are restricted to achieve PDF/A compatibility

item	***PDFlib functions and parameters to be avoided for PDF/A compatibility***
annotations	*PDF_create_annotation( ): annotations with* type=FileAttachment *must be avoided; 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.*
form fields	*PDF_create_field( ) and PDF_create_fieldgroup( ) for creating form fields must be avoided*

Table 9.10 Operations which must be avoided or are restricted to achieve PDF/A compatibility

item	PDFlib functions and parameters to be avoided for PDF/A compatibility
actions and JavaScript	*PDF_create_action( ): actions with* `type=Launch`, `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 PDF_load_image( ) must be avoided.*
templates	*The* `OPI-1.3` *and* `OPI-2.0` *options in PDF_begin_template_ext( ) must be avoided.*
transparency	*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.* *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`. *The* `opacity` *option in PDF_create_annotation( ) must be avoided.*
security	*The* `userpassword`, `masterpassword`, *and* `permissions` *options in PDF_begin_document( ) 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 Section 9.5.3, »Importing PDF/A Documents with PDI«, page 210)*

Additional requirements and restrictions for PDF/A-1a. Table 9.11 lists additional requirements which must be met when creating PDF/A-1a: all requirements for creating Tagged PDF output as discussed in Section 9.6, »Tagged PDF«, page 214, must be met. Table 9.11 lists additional restrictions on operations which must be avoided when generating PDF/A-1a.

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, but violates the goal of faithful semantic reproduction, and therefore the spirit of PDF/A-1a.

Table 9.11 Additional requirements for PDF/A-1a compatibility

item	PDFlib function and parameter requirements for PDF/A compatibility
Tagged PDF	*All requirements for Tagged PDF must be met (see Section 9.6, »Tagged PDF«, page 214).* *The following are strongly recommended:* ► *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( ): a non-empty string must be supplied for the* `contents` *option*

Table 9.12 Additional restrictions for PDF/A-1a compatibility

item	PDFlib functions and parameters to be avoided for PDF/A compatibility
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 9.13), 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=copyoutputintent*.

Creating PDF/A and PDF/X at the same time. A PDF/A document can at the same time conform to PDF/X-1a:2003 or PDF/X-3:2003. In order to achieve 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-3:2003 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.

9.5.3 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 6.2, »Importing PDF Pages with PDI (PDF Import Library)«, page 130, 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 9.13.

Note PDFlib does not validate PDF documents for PDF/A compliance, nor can it convert valid PDF/A from arbitrary input PDFs. If you have a need for these features we recommend the product pdfaPilot from callas software GmbH[1].

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. Imported documents which are incompatible to the current PDF/A level will be rejected.

If multiple PDF/A documents are imported, they must all have been prepared for a compatible output condition according to Table 9.14. The output intents in all imported doc-

[1] *See www.callassoftware.com*

Table 9.13 Compatible PDF/A input levels for various PDF/A output levels

	PDF/A level of the imported document	
PDF/A output level	***PDF/A-1a:2005***	***PDF/A-1b:2005***
PDF/A-1a:2005	–	*allowed*
PDF/A-1b:2005	–	*allowed*

uments must be identical or compatible; it is the user's responsibility to make sure that this condition is met.

Table 9.14 Output intent compatibility when importing PDF/A documents

		output intent of imported document		
output intent of generated 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 compatibility 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 an ISO 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 should first use pCOS to check for the existence of an output intent before attempting to copy it:

```
res = p.pcos_get_string(doc, "type:/Root/OutputIntents");
if (res.equals("array"))
{
        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. 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( )*.

9.5.4 Color Strategies for creating PDF/A

The PDF/A requirements related to color handling may be confusing. The summary of color strategies in Table 9.15 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, and because *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
- Hypertext elements may specify border colors

Table 9.15 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 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-compatible 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);
```

In addition to the color spaces listed in Table 9.15, 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 color mode.

9.5.5 PDF/A Validation

Acrobat 7. The Preflight tool in Acrobat 7.07 includes a validation profile for PDF/A-1b (Acrobat 7.0 supports only an earlier draft version of PDF/A, but not the final ISO standard, and should therefore not be used for validation). However, there are a few items where it differs from the PDF/A standard, and issues inappropriate warnings. While we implemented workarounds for some problems, the inappropriate warnings in the following list can be issued for PDFlib-generated output. We discussed these items with the developers of the Preflight plugin and verified that the output created by PDFlib fully conforms to the PDF/A standard despite those warnings:

- All PDFlib-generated documents raise a warning »PDF/A label missing« because it expects (and Acrobat 7.07 creates) a wrong PDF/A namespace URI in the XMP metadata containing the PDF/A signature.
- Combined PDF/A and PDF/X files raise a warning »OutputIntent for PDF/X missing« even when an output intent is present.

Acrobat 8. The Preflight tool in Acrobat 8 includes validation profiles for PDF/A-1a and PDF/A-1b, and validates according to the final ISO standard (including the 2006 corrigendum). We regard Acrobat 8 Preflight as reference tool for PDF/A validation. According to our testing, PDFlib's PDF/A output is fully compliant to PDF/A according to Acrobat 8 Preflight.

Other PDF/A validation tools. Several third-party products are available for PDF/A validation. We identified a number of problems with some of those tools, and are actively working with the respective software developers in order to obtain a common understanding of the PDF/A standard. Please contact us if you have questions regarding validating PDFlib's PDF/A output with specific tools.

9.6 Tagged PDF

Tagged PDF is a certain kind of enhanced PDF which enables additional features in PDF viewers, such as accessibility support, text reflow, reliable text extraction and conversion to other document formats such as RTF or XML.

PDFlib supports Tagged PDF generation. However, Tagged PDF can only be created if the client provides information about the document's internal structure, and obeys certain rules when generating PDF output.

Note *PDFlib doesn't support custom structure element types (i.e. only standard structure types as defined by PDF can be used), role maps, and structure element attributes.*

9.6.1 Generating Tagged PDF with PDFlib

Required operations. Table 9.16 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 9.16 Operations which must be applied for generating Tagged PDF

item	*PDFlib function and parameter requirements for Tagged PDF compatibility*
Tagged PDF output	*The* `tagged` *option in PDF_begin_document( ) must be set to* `true`.
document language	*The* `lang` *option in PDF_begin_document( ) must be set to specify the natural language of the document. It must initially be set for the document as a whole, but can later be overridden for individual items on an arbitrary structure level.*
structure information	*Structure information and artifacts must be identified as such. All content-generating API functions should be enclosed by PDF_begin_item( ) / PDF_end_item( ) pairs.*

Unicode-compatible text output. When generating Tagged PDF, all text output must use fonts which are Unicode-compatible as detailed in Section 5.4.4, »Unicode-compatible Fonts«, page 111. This means that all used fonts must provide a mapping to Unicode. Non Unicode-compatible fonts are only allowed if alternate text is provided for the content via the *ActualText* or *Alt* options in *PDF_begin_item( )*. PDFlib will throw an exception if text without proper Unicode mapping is used while generating Tagged PDF.

Note *In some cases PDFlib will not be able to detect problems with wrongly encoded fonts, for example symbol fonts encoded as text fonts. Also, due to historical problems PostScript fonts with certain typographical variations (e.g., expert fonts) are likely to result in inaccessible output.*

Page content ordering. The ordering of text, graphics, and image operators which define the contents of the page is referred to as the content stream ordering; the content ordering defined by the logical structure tree is referred to as logical ordering. Tagged PDF generation requires that the client obeys certain rules regarding content ordering.

The natural and recommended method is to sequentially generate all constituent parts of a structure element, and then move on to the next element. In technical terms, the structure tree should be created during a single depth-first traversal.

A different method which should be avoided is to output parts of the first element, switch to parts of the next element, return to the first, etc. In this method the structure tree is created in multiple traversals, where each traversal generates only parts of an element.

Importing Pages with PDI. Pages from Tagged PDF documents or other PDF documents containing structure information cannot be imported in Tagged PDF mode since the imported document structure would interfere with the generated structure.

Pages from unstructured documents can be imported, however. Note that they will be treated »as is« by Acrobat's accessibility features unless they are tagged with appropriate *ActualText*.

Artifacts. Graphic or text objects which are not part of the author's original content are called artifacts. Artifacts should be identified as such using the *Artifact* pseudo tag, and classified according to one of the following categories:

- *Pagination:* features such as running heads and page numbers
- *Layout:* typographic or design elements such as rules and table shadings
- *Page:* production aids, such as trim marks and color bars.

Although artifact identification is not strictly required, it is strongly recommended to aid text reflow and accessibility.

Inline items. PDF defines block-level structure elements (BLSE) and inline-level structure elements (ILSE) (see function 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 9.17 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 9.18 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 9.18 Operations which are recommended for generating Tagged PDF

item	Optional PDFlib function and parameter for Tagged PDF compatibility
hyphenation	*Word breaks (separating words in two parts at the end of a line) should be presented using a soft hyphen character (U+00A0) as opposed to a hard hyphen (U+002D)*
word boundaries	*Words should be separated by space characters (U+0020) even if this would not strictly be required for positioning. The* autospace *parameter can be used for automatically generating space characters after each call to one of the show functions.*
artifacts	*In order to distinguish real content from page artifacts, artifacts should be identified as such using PDF_begin_item( ) with* tag=Artifact.
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).*

Prohibited operations. Table 9.19 lists all operations which are prohibited when generating Tagged PDF output. Calling one of the prohibited functions while in Tagged PDF mode will trigger an exception.

Table 9.19 Operations which must be avoided when generating Tagged PDF

item	PDFlib functions and parameters to be avoided for Tagged PDF compatibility
non-Unicode compatible fonts	*Fonts which are not Unicode-compatible according to Section 5.4.4, »Unicode-compatible Fonts«, page 111, must be avoided.*
PDF import	*Pages from PDF documents which contain structure information (in particular: Tagged PDF documents) must not be imported.*

9.6.2 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 lang=en") == -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 7.2, »Multi-Line Textflows«, page 142) 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.

9.6.3 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 9.1. 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);
        ...
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);
```

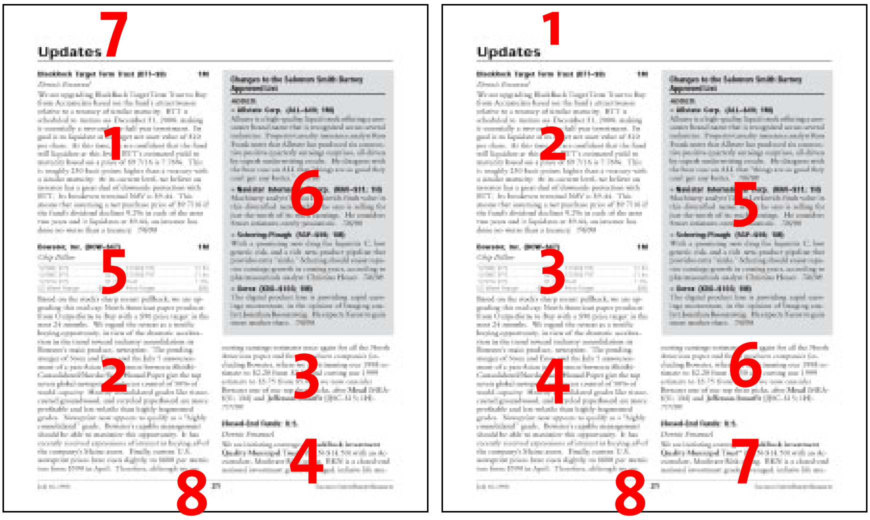


Fig. 9.1
Creating a complex page layout in logical structure order (left) and in visual order (right). The right variant uses item activation for the first section before continuing fragments 4 and 6.

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.

9.6.4 Using Tagged PDF in Acrobat

This section mentions observations which we made while testing Tagged PDF output in Adobe Acrobat. Unless mentioned otherwise the observations below apply to Acrobat 6 and 7. They are mostly related to bugs or inconsistent behavior in Acrobat. A workaround is provided in cases where we found one.

Acrobat's Reflow Feature. Acrobat allows Tagged PDF documents to reflow, i.e. to adjust the page contents to the current window size. While testing Tagged PDF we made several observations regarding the reflow feature in Acrobat:

- The order of content on the page should follow the desired reflow order.
- Symbol (non-Unicode fonts) can cause Acrobat's reflow feature to crash. For this reason it is recommended to put the text in a *Figure* element.
- BLSEs may contain both structure children and direct content elements. In order for the reflow feature (as well as Accessibility checker and Read Aloud) to work it is recommended to put the direct elements before the first child elements.

- The *BBox* option should be provided for tables and illustrations. The *BBox* should be exact; however, for tables only the lower left corner has to be set exactly. As an alternative to supplying a *BBox* entry, graphics could also be created within a BLSE tag, such as *P, H,* etc. However, vector graphics will not be displayed when Reflow is active. If the client does not provide the *BBox* option (and relies on automatic *BBox* generation instead) all table graphics, such as cell borders, should be drawn outside the table element.
- Table elements should only contain table-related elements (*TR, TD, TH, THead, TBody,* etc.) as child elements, but not any others. For example, using a *Caption* element within a table could result in reflow problems, although it would be correct Tagged PDF.
- Content covered by the *Private* tag will not be exported to other formats. However, they are subject to reflow and Read Aloud, and illustrations within the *Private* tag must therefore have alternate text.
- Imported images should be covered by a *Figure* element; imported PDF pages should be covered by a *Form* item. Item type *Formula* should be avoided in order to avoid problems with reflow.
- Reflow seems to have problems with PDF documents generated with the *topdown* option.
- Structure items with mixed types of children (i.e., both page content sequences and non-inline structure elements) should be avoided since otherwise Reflow could fail.
- If an activated item contains only content, but no structure children, Reflow could fail, especially if the item is activated on another page. This problem can be avoided by wrapping the activated item with a non-inline *Span* tag.
- Acrobat 7 cannot reflow pages containing form fields; it will display a warning in this case. Acrobat 6 can reflow pages with form fields, but will omit the form fields. Acrobat 7, on the other hand, is able to reflow annotations.

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:

- In order to make form fields accessible, use the *tooltip* option of *PDF_create_field( )* and *PDF_create_fieldgroup( )*.
- If the *Lbl* tag is set within the *TOCI* tag (as actually described in the PDF reference) the Accessibility Checker will warn that the *Lbl* tag is not set within an *LI* tag.
- Acrobat 6: Elements containing an imported image should use the *Alt* property. The *ActualText* property could cause the accessibility checker to crash. Another reason to prefer *Alt* over *ActualText* is that the Read Aloud feature will catch the real text.
- Acrobat 6: If a *Form* tag covering an imported PDF page is the very first item on the page it can cause problems with the accessibility checker.

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.

- 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.
- The content of a *NonStruct* tag will not be exported to HTML 4.01 CSS 1.0 (but it will be used for HTML 3.2 export).

- Alternate text should be supplied for ILSEs (such as *Code*, *Quote*, or *Reference)*. If the *Alt* option is used, Read Aloud will read the provided text, but the real content will be exported to other formats. If the *ActualText* option is used, the provided text will be used both for reading and exporting.
- Acrobat 6: Elements containing an imported image should use the *Alt* property instead of *ActualText* so that the Export feature will catch the real text.

Acrobat's Read Aloud Feature. Tagged PDF will enhance Acrobat's capability to read text aloud.

- When supplying *Alt* or *ActualText* it is useful to include a space character at the beginning. This allows the Read Aloud feature to distinguish the text from the preceding sentence. For the same reason, including a '.' character at the end may also be useful. Otherwise Read Aloud will try to read the last word of the preceding sentence in combination with the first word of the alternate text.

10 Variable Data and Blocks

PDFlib supports a template-driven PDF workflow for variable data processing. Using the concept of blocks, imported pages can be populated with variable amounts of single- or multi-line text, images, or PDF graphics which can be pulled from an external source. This can be used to easily implement applications which require customized PDF documents, for example:

- mail merge
- flexible direct mailings
- transactional and statement processing
- business card personalization

Note *Block processing requires the PDFlib Personalization Server (PPS). Although PPS is contained in all commercial PDFlib packages, you must purchase a license key for PPS; a PDFlib or PDFlib+PDI license key is not sufficient. The PDFlib Block plugin for Adobe Acrobat is required for creating blocks in PDF templates interactively.*

10.1 Installing the PDFlib Block Plugin

The Block plugin and its sibling, the PDF form field conversion plugin, work with Acrobat 5, Acrobat 6/7 Standard and Acrobat 6/7 Professional on Windows and Mac, and Acrobat 8 Standard and Professional on Windows. The plugins don't work with Acrobat Elements or any version of Acrobat Reader/Adobe Reader.

Note *The plugins contain multiple language versions of the user interface, and will automatically use the same interface language as the Acrobat application if possible. If the plugin does not support the native Acrobat language the English user interface will be used instead.*

Installing the PDFlib Block plugins for Acrobat 5/6/7/8 on Windows. To install the PDFlib Block plugin and the PDF form field conversion plugin in Acrobat 5, 6, 7, or 8 the plugin files must be copied to 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 7.0\Acrobat\plug_ins\PDFlib Block Plugin
```

Installing the PDFlib Block plugins for Acrobat 6/7 on the Mac. With Acrobat 6/7 the plugin folder will not be visible in the Finder. Instead of dragging the plugin files to the plugin folder use the following steps (make sure that Acrobat is not running):

- Extract the plugin files to a folder by double-clicking the disk image.
- Locate the Acrobat application icon in the Finder. It is usually located in a folder which has a name similar to the following:

  ```
  /Applications/Adobe Acrobat 7.0 Professional
  ```

- Single-click on the Acrobat application icon and select *File, Get Info*.
- In the window that pops up click the triangle next to *Plug-ins*.
- Click *Add...* and select the *PDFlib Block Plugin* folder (for Acrobat 7) or the *PDFlib Block Plugin Acro 5-6* folder (for Acrobat 6) from the folder which has been created in the

first step. Note that after installation this folder will not immediately show up in the list of plugins, but only when you open the info window next time.

Installing the PDFlib Block plugins for Acrobat 5 on the Mac. To install the plugins for Acrobat 5, start by double-clicking the disk image. Drag the folder *PDFlib Block Plugin Acro 5-6* to the Acrobat 5 plugin folder. A typical plugin folder name is as follows:

```
/Applications/Adobe Acrobat 5.0/Plug-Ins
```

Trouble-shooting. If the PDFlib Block plugin doesn't seem to work check the following:

- Make sure that in Edit, Preferences, [General...], Startup (Acrobat 6/7) or Options (Acrobat 5) 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 PDF'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.

10.2 Overview of the PDFlib Block Concept

10.2.1 Complete Separation of Document Design and Program Code

PDFlib data blocks make it easy to place variable text, images, or graphics on imported pages. In contrast to simple PDF pages, pages containing data blocks intrinsically carry information about the required processing which will be performed later on the server side. The PDFlib block concept completely separates the following tasks:

- A designer creates the page layout, and specifies the location of variable text and image elements along with relevant properties such as font size, color, or image scaling. After creating the layout as a PDF document, the designer uses the PDFlib Block plugin for Acrobat to specify variable data blocks and their associated properties.
- A programmer writes code to connect the information contained in PDFlib blocks on imported PDF pages with dynamic information, e.g., database fields. The programmer doesn't need to know any details about a block (whether it contains a name or a ZIP code, the exact location on the page, its formatting, etc.) and is therefore independent from any layout changes. PDFlib will take care of all block-related details based on the block properties found in the file.

In other words, the code written by the programmer is »data-blind« – it is generic and does not depend on the particulars of any block. For example, the designer may decide to use the first name of the addressee in a mailing instead of the last name. The generic block handling code doesn't need to be changed, and will generate correct output once the designer changed the block properties with the Acrobat plugin to use the first name instead of the last name.

Example: adding variable text to a template. Adding dynamic text to a PDF template is a very common task. The following code fragment will open a page in an input PDF document (the template), place it on the output page, and fill some variable text into a text block called *firstname:*

```
doc = 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, "");
p.fit_pdi_page(page, 0.0, 0.0, "");
p.fill_textblock(page, "firstname", "Serge", "encoding=winansi");
p.close_pdi_page(page);
p.end_page_ext("");
p.close_pdi_document(doc);
```

Controlling the display order of imported page and 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 with the *blind* option of *PDF_fit_pdi_page( ):*

```
/* 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");
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, "");
```

10.2.2 Block Properties

The behavior of blocks can be controlled with block properties. The properties are assigned to a block with the PDFlib Block plugin for Acrobat.

Standard block properties. PDFlib blocks are defined as rectangles on the page which are assigned a name, a type, and an open set of properties which will later be processed on the server side. The name is an arbitrary string which identifies the block, such as *firstname*, *lastname*, or *zipcode*. PDFlib supports the following kinds of blocks:

- Type *Text* means that the block will hold one or more lines of textual data. Multi-line text will be formatted with the Textflow feature. Textflow blocks can be linked so that one block holds the overflow text of the previous block (see Section 10.2.3, »Linking multiple Textflow Blocks«, page 227).
- Type *Image* means that the block will hold a raster image. This is similar to importing a TIFF or JPEG file in a DTP application.
- Type *PDF* means that the block will hold arbitrary PDF graphics imported from a page in another PDF document. This is similar to importing an EPS graphic in a DTP application.

A block may carry a number of standard properties depending on its type. For example, a text block may specify the font and size of the text, an image or PDF block may specify the scaling factor or rotation. For each type of block the PDFlib API offers a dedicated function for processing the block. These functions search an imported PDF page for a block by its name, analyze its properties, and place some client-supplied data (text, raster image, or PDF page) on the new page according to the corresponding block properties.

Custom block properties. Standard block properties make it possible to quickly implement variable data processing applications, but these are limited to the set of properties which are internally known to PDFlib and can automatically be processed. In order to provide more flexibility, the designer may also assign custom properties to a block. These can be used to extend the block concept in order to match the requirements of the most demanding variable data processing applications.

There are no rules for custom properties since PDFlib will not process custom properties in any way, except making them available to the client. The client code can examine the custom properties and act in whatever way it deems appropriate. Based on some custom property of a block the code may make layout-related or data-gathering decisions. For example, a custom property for a scientific application could specify the number of digits for numerical output, or a database field name may be defined as a custom block property for retrieving the data corresponding to this block.

Overriding block properties. In certain situations the programmer would like to use only some of the properties provided in a block definition, but override other properties with custom values. This can be useful in various situations:

- The scaling factor for an image or PDF page will be calculated instead of taken from the block definition.
- Change the block coordinates programmatically, for example when generating an invoice with a variable number of data items.
- Individual spot color names could be supplied in order to match the requirements of individual customers in a print shop application.

Property overrides can be achieved by supplying property names and the corresponding values in the option list of all *PDF_fill_*block( )* functions as follows:

```
p.fill_textblock(page, "firstname", "Serge", "fontsize=12");
```

This will override the block's internal *fontsize* property with the supplied value 12. Almost all names of general properties can be used as options, as well as those specific to a particular block type. For example, the *underline* option is only allowed for *PDF_fill_textblock( )*, while the *scale* option is allowed for both *PDF_fill_imageblock( )* and *PDF_fill_pdfblock( )* since *scale* is a valid property for both image and PDF blocks.

Property overrides apply only to the respective function calls; they will not be stored in the block definition.

Coordinate systems. The coordinates describing a block reference the PDF default coordinate system. When the page containing the block is placed on the output page, several positioning and scaling options may be supplied to *PDF_fit_pdi_page( )*. These parameters are taken into account when the block is being processed. This makes it possible to place a template page on the output page multiply, every time filling its blocks with data. For example, a business card template may be placed four times on an imposition sheet. The block functions will take care of the coordinate system transformations, and correctly place the text for all blocks in all invocations of the page. The only requirement is that the client must place the page and then process all blocks on the placed page. Then the page can be placed again at a different location on the output page, followed by more block processing operations referring to the new position, and so on.

Note The Block plugin will display the block coordinates differently from what is stored in the PDF file. The plugin uses Acrobat's convention which has the coordinate origin in the upper left corner of the page, while the internal coordinates (those stored in the block) use PDF's convention of having the origin at the lower left corner of the page.

10.2.3 Linking multiple Textflow Blocks

Textflow blocks can be linked so that one block holds the overflow text from 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 text which is still available after filling 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.

With linked Textflow blocks the overflow text which remains after filling the first block can be filled into the next block. The remainder of the 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 blocks a value of -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 user.
- In the next call 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 skeleton 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;
        }
}
```

10.2.4 Why not use PDF Form Fields?

Experienced Acrobat users may ask why we implemented a new block concept for PDFlib, instead of relying on the established form field scheme available in PDF. The primary distinction is that PDF form fields are optimized for interactive filling, while PDFlib blocks are targeted at automated filling. Applications which need both interactive and automated filling can easily achieve this by using a feature which automatically converts form fields to blocks (see Section 10.3.4, »Converting PDF Form Fields to PDFlib Blocks«, page 235).

Although there are many parallels between both concepts, PDFlib blocks offer several advantages over PDF form fields as detailed in Table 10.1.

Table 10.1 Comparison of PDF form fields and PDFlib blocks

feature	***PDF form fields***	***PDFlib blocks***
design objective	*for interactive use*	*for automated filling*
typographic features (beyond choice of font and font size)	–	*kerning, word and character spacing, underline/ overline/strikeout*
font control	*font embedding*	*font embedding and subsetting, encoding*
text formatting controls	*left-, center-, right-aligned*	*left-, center-, right-aligned, justified; various formatting algorithms and controls; inline options can be used to control the appearance of text*
change font or other text attributes within text	–	*yes*
merged result is integral part of PDF page description	–	*yes*
users can edit merged field contents	*yes*	*no*
extensible set of properties	–	*yes (custom block properties)*
use image files for filling	–	*BMP, CCITT, GIF, PNG, JPEG, JPEG 2000, TIFF*
color support	*RGB*	*grayscale, RGB, CMYK, Lab, spot color (HKS and Pantone spot colors integrated in the Block plugin)*
PDF/X and PDF/A compatible	*PDF/X: no; PDF/A: restricted*	*yes (both template with blocks and merged results)*
graphics and text properties can be overridden upon filling	–	*yes*
Text blocks can be linked	–	*yes*

10.3 Creating PDFlib Blocks

10.3.1 Creating Blocks interactively with the PDFlib Block Plugin

Activating the PDFlib Block tool. The PDFlib Block plugin for creating PDFlib blocks is similar to the form tool in Acrobat. All blocks on the page will be visible when the block tool is active. When another Acrobat tool is selected the blocks will be hidden, but they are still present. You can activate the block tool in several ways:

- by clicking the block icon [≡] in Acrobat's *Advanced Editing* toolbar (in Acrobat 5: *Editing* toolbar);
- via the menu item *PDFlib Blocks, PDFlib Block Tool;*
- by using the keyboard shortcut *P*; make sure to enable *Edit, Preferences, [General...], General, Use single key accelerators to access tools*, which is disabled by default (not required in Acrobat 5)

Creating and modifying blocks. Once you activated the block tool you can simply drag the cross-hair pointer to create a block at the desired position on the page and the desired size. Blocks will always be rectangular with edges parallel to the page edges. When you create a new block the block properties dialog appears where you can edit various properties of the block (see Section 10.3.2, »Editing Block Properties«, page 232). The block tool will automatically create a block name which can be changed in the properties dialog. Block names must be unique within a page. You can change the block type in the top area to one of Text, Image, or PDF. The *General* and *Custom* tabs will always be available, while only one of the *Text, Image,* and *PDF* tabs will be active at a time depending on the chosen block type. The *Textflow* tab will only be present for blocks of type text if the *textflow* property is *true*. Another tab labelled *Tabs* (tabulator positions) will only be available if the *hortabmethod* property in the *Textflow* tab has been set to *ruler*.

Note After you added blocks or made changes to existing blocks in a PDF, use Acrobat's »Save as...« Command (as opposed to »Save«) to achieve smaller file sizes.

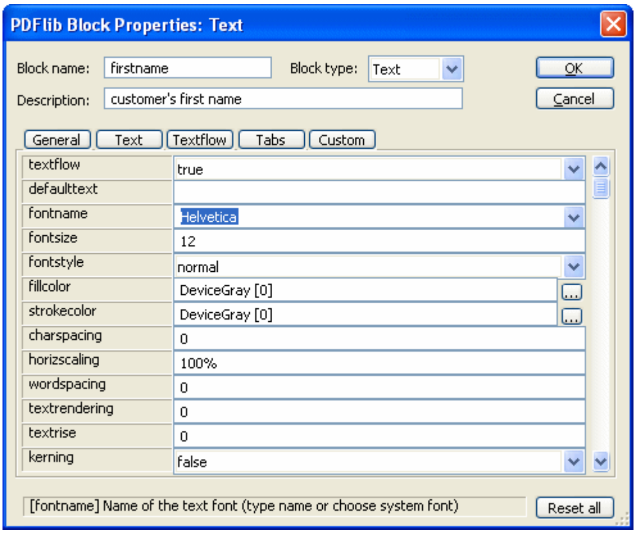
Note When using the Acrobat plugin Enfocus PitStop to edit documents which contain PDFlib blocks you may see the message »This document contains PieceInfo from PDFlib. Press OK to continue editing or Cancel to abort.« This message can be ignored; it is safe to click OK in this situation.

Selecting blocks. Several block operations, such as copying or moving, work with selected blocks. You can select one or more blocks with the block tool as follows:

- To select a single block simply click on it with the mouse.
- Hold down the Shift key while clicking on another block to extend the selection.
- Press Ctrl-A (on Windows) or Cmd-A (on the Mac) or *Edit, Select All* to select all blocks on a page.

The context menu. When one or more blocks are selected you can open the context menu to quickly access block-related functions (which are also available in the PDFlib Blocks menu). To open the context menu, click on the selected block(s) with the right mouse button on Windows, or Ctrl-click the block(s) on the Mac.

For example, to delete a block, select it with the block tool and press the *Delete* key, or use *Edit, Delete* in the context menu.



Block name: firstname

Block type: Text

Description: customer's first name

textflow	true
defaulttext	
fontname	Helvetica
fontsize	12
fontstyle	normal
fillcolor	DeviceGray [0]
strokecolor	DeviceGray [0]
charspacing	0
horizscaling	100%
wordspacing	0
textrendering	0
textrise	0
kerning	false

[fontname] Name of the text font (type name or choose system font)

Fig. 10.1
Editing block properties: the Textflow panel is only visible if `textflow=true;` *the Tabs panel is only visible if* `hortabmethod=ruler`

Fine-tuning block size and position. Using the block tool you can move one or more selected blocks to a different position. Hold down the Shift key while dragging a block to restrain the positioning to horizontal and vertical movements. This may be useful for exactly aligning blocks. When the pointer is located near a block corner, the pointer will change to an arrow and you can resize the block. To adjust the position or size of multiple blocks, select two or more blocks and use the *Align, Center, Distribute,* or *Size* commands from the *PDFlib Blocks* menu or the context menu. The position of one or more blocks can also be changed by using the arrow keys.

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 proceed as follows:

- In Acrobat 6/7 go to *Edit, Preferences, [General...], Units & Guides* [or *Unit, Page Units* in Acrobat 7 Standard] and choose one of Points, Inches, Millimeters, Picas, Centimeters. You can also go to *View, Navigation Tabs, Info* and select a unit from the *Options* menu.
- In Acrobat 5 go to *Edit, Preferences, General..., Display, Page Units* and choose one of Points, Inches, Millimeters. You can also go to *Window, Info* and select a unit from the *Info* menu.

Note that the chosen unit will only affect the *Rect* property, but not any other numerical properties.

Creating blocks by selecting an image or graphic. As an alternative to manually dragging block rectangles you can use existing page contents to define the block size. First, make sure that the menu item *PDFlib Blocks, Click Object to define Block* is enabled. Now you can use the block tool to click on an image on the page in order to create a block with the size of the image. You can also click on other graphical objects, and the block tool will try to select the surrounding graphic (e.g., a logo). The *Click Object* feature is intended as an aid for defining blocks. If you want to reposition or resize the block you can do so afterwards without any restriction. The block will not be locked to the image or graphics object which was used as a positioning and sizing aid.

The *Click Object* feature will try to recognize which vector graphics and images form a logical element on the page. When some page content is clicked, its bounding box (the surrounding rectangle) will be selected unless the object is white or very large. In the next step other objects which are partially contained in the detected rectangle will be added to the selected area, and so on. The final area will be used as the basis for the generated block rectangle. The end result is that the *Click Object* feature will try to select complete graphics, and not only individual lines.

The *Click Object* feature isn't perfect: it will not always select what you want, depending on the nature of the page content. Keep in mind that this feature is only intended as a positioning aid for quickly creating block rectangles.

Automatically detecting font properties. The PDFlib Block plugin can analyze the underlying font which is present at the location where a block is positioned, and can automatically fill in the corresponding properties of the block:

```
fontname, fontsize, fillcolor, charspacing, horizscaling, wordspacing,
textrendering, textrise
```

Since automatic detection of font properties can result in undesired behavior when the background shall be ignored, it can be activated or deactivated using *PDFlib Blocks, Detect underlying font and color*. By default this feature is turned off.

Locking blocks. Blocks can be locked to protect them against accidentally moving, resizing, or deleting them. With the block tool active, select the block and choose *Lock* from its context menu. While a block is locked you cannot move, resize, or delete it, nor display its properties dialog.

Using Blocks with PDF/X. Unlike PDF form fields, PDFlib blocks are PDF/X compatible. Both the input document containing blocks as well as the generated output PDF can be made PDF/X conforming. However, in preparing block files for a PDF/X workflow you may run into the following problem:

- PDF/X-1:2001, PDF/X-1a:2001, and PDF/X-3:2002 are based on Acrobat 4/PDF 1.3, and do not support Acrobat 5 files;
- The PDFlib Block plugin requires Acrobat 5 or above.

You can solve this problem by using Acrobat to convert the files to the required PDF version. See »Changing the PDF version of a document«, page 196, for details.

10.3.2 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 10.1). As detailed in Section 10.4, »Standard Properties for Automated Processing«, page 238, there are several types of properties:

- Name, type, description, and the properties in the *General* tab apply to all blocks.
- Properties in the *Text, Image,* and *PDF* tabs apply only to the respective block type. Only the tab corresponding to the block's type will be active, while the other tabs are inactive.

- If a block of type *Text* has the *textflow* property set to *true,* another tab called *Textflow* will appear with Textflow-related settings.
- If a block of type Text has the *textflow* property set to *true,* and the *hortabmethod* property in the *Textflow* tab is set to *ruler*, still another panel called *Tabs* will appear where you can edit tabulator settings.
- Properties in the *Custom* tab can be defined by the user, and apply to any block type.

To change a property's value enter the desired number or string in the property's input area (e.g. *linewidth),* choose a value from a drop-down list (e.g. *fontname, fitmethod),* or select a color value or file name by clicking the »...« button at the right-hand side of the dialog (e.g. *backgroundcolor)*. For the *fontname* property you can either choose from the list of fonts installed on the system, 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 the PDFlib Personalization Server.

When you are done editing properties, click OK to close the properties dialog. The properties just defined will be stored in the PDF file as part of the block definition.

Stacked blocks. Overlapping blocks can be difficult to select since clicking an area with the mouse will always select the topmost block. In such a situation the *Choose Block* entry in the context menu can be used to select one of the blocks by name. As soon as a block has been selected the next action (e.g. double-click) within its area will not affect other blocks, but only the selected one. This way block properties can easily be edited even for blocks which are partially or completely covered by other blocks.

Using and restoring default properties. In order to save some amount of typing and clicking, the block tool will remember the property values which have been entered into the previous block's properties dialog. These values will be reused when you create a new block. Of course you can override these values with different ones at any time.

Pressing the *Reset All* button in the properties dialog will reset most block properties to their respective defaults. However, the following items will remain unmodified:

- the *Name, Type, Rect,* and *Description* properties
- all custom properties.

Shared properties. By holding the Shift key and using the block tool to select multiple blocks you can select an arbitrary number of blocks on a page. Double-clicking one of the selected blocks or pressing the *Enter* key will display the properties dialog which now applies to all selected blocks. However, since not all properties can be shared among multiple blocks, only a subset of all properties will be available for editing. Section 10.4, »Standard Properties for Automated Processing«, page 238, details which properties can be shared among multiple blocks. Custom properties cannot be shared.

10.3.3 Copying Blocks between Pages and Documents

The Block plugin offers several methods for moving and copying blocks within the current page, the current document, or other documents:

- move or copy blocks by dragging them with the mouse, or pasting blocks to another page or open document
- duplicate blocks on one or more pages of the same document
- export blocks to a new file (with empty pages) or to an existing document (apply the blocks to existing pages)

- import blocks from other documents

In order to update the page contents while maintaining block definitions you can replace the underlying page(s) while keeping the blocks. Use *Document, Replace Pages...* (Acrobat 5 and 7) or *Document, Pages, Replace* (Acrobat 6).

Moving and copying blocks. You can relocate blocks or create copies of blocks by selecting one or more blocks and dragging them to a new location while pressing the Ctrl key (on Windows) or Alt key (on the Mac). The mouse cursor will change while the key is pressed. A copied block will have the same properties as the original block, with the exception of its name and position which will automatically be changed.

You can also use copy/paste to copy blocks to another location on the same page, to another page in the same document, or to another document which is currently open in Acrobat:

- Activate the block tool and select the blocks you want to copy.
- Use Ctrl-C (on Windows) or Cmd-C (on the Mac) or *Edit, Copy* to copy the selected blocks to the clipboard.
- Use Ctrl-V (on Windows) or Cmd-V (on the Mac) or *Edit, Paste* to paste the blocks which are currently in the clipboard.

Duplicating blocks on other pages. You can create duplicates of one or more blocks on an arbitrary number of pages in the current document simultaneously:

- Activate the block tool and select the blocks you want to duplicate.
- Choose *Import and Export, Duplicate...* from the *PDFlib Blocks* menu or the context menu.
- Choose which blocks to duplicate (selected blocks or all on the page) and the range of target pages where you want duplicates of the blocks.

Exporting and importing blocks. Using the export/import feature for blocks it is possible to share the block definitions on a single page or all blocks in a document among multiple PDF files. This is useful for updating the page contents while maintaining existing block definitions. To export block definitions to a separate file proceed as follows:

- Activate the block tool and Select the blocks you want to export.
- Choose *Import and Export, Export...* from the *PDFlib Blocks* menu or the context menu. Enter the page range and a file name for the file containing the block definitions.

You can import block definitions via *PDFlib Blocks, Import and Export, Import...* . Upon importing blocks you can choose whether to apply the imported blocks to all pages in the document, or only to a page range. If more than one page is selected the block definitions will be copied unmodified to the pages. If there are more pages in the target range than in the imported block definition file you can use the *Repeate Template* checkbox. If it is enabled the sequence of blocks in the imported file will be repeated in the current document until the end of the document is reached.

Copying blocks to another document upon export. When exporting blocks you can immediately apply them to the pages in another document, thereby propagating the blocks from one document to another. In order to do so choose an existing document to export the blocks to. If you activate the checkbox *Delete existing blocks* all blocks which may be present in the target document will be deleted before copying the new blocks into the document.

10.3.4 Converting PDF Form Fields to PDFlib Blocks

As an alternative to creating PDFlib blocks manually you can automatically convert PDF form fields to blocks. This is especially convenient if you have complicated PDF forms which you want to fill automatically with the PDFlib Personalization Server, or need to convert a large number of existing PDF forms for automated filling. In order to convert all form fields on a page to PDFlib blocks choose *PDFlib Blocks, Convert Form Fields, Current Page*. To convert all form fields in a document choose *All Pages* instead. Finally, you can convert only selected form fields (choose Acrobat's Form Tool or the Select Object Tool to select form fields) with *Selected Form Fields*.

Form field conversion details. Automatic form field conversion will convert form fields of the types selected in the *PDFlib Blocks, Convert Form Fields, Conversion Options...* dialog to blocks of type *Text*. By default all form field types will be converted. Attributes of the converted fields will be transformed to the corresponding block properties according to Table 10.2.

Table 10.2 Conversion of PDF form fields to PDFlib blocks

PDF form field attribute...	***...will be converted to the PDFlib block property***
all fields	
Position	*General, Rect*
Name	*General, Name*
Tooltip	*General, Description*
Appearance, Text, Font	*Text, fontname*
Appearance, Text, Font Size	*Text, fontsize; »auto« font size will be converted to a fixed font size of 2/3 of the block height, and the* `fitmethod` *will be set to* `auto`*. For multi-line fields/blocks this combination will automatically result in a suitable font size which may be smaller than the initial value of 2/3 of the block height.*
Appearance, Text, Text Color	*Text, strokecolor; Text, fillcolor*
Appearance, Border, Border Color	*General, bordercolor*
Appearance, Border, Fill Color	*General, backgroundcolor*
Appearance, Border, Line Thickness	*General, linewidth: Thin=1, Medium=2, Thick=3*
General, Common Properties, Form Field	*General, Status: Visible=active, Hidden=ignore, Visible but doesn't print=ignore, Hidden but printable=active*
General, Common Properties, Orientation	*General, orientate: 0=north, 90=west, 180=south, 270=east*
text fields	
Options, Default Value	*Text, defaulttext*
Options, Alignment	*General, position: Left={left center}, Center={center center}, Right={right center}*
Options, Multi-line	*Text, textflow: checked=true, unchecked=false*
radio buttons and check boxes	
If »Check box/Button is checked by default« is selected: Options, Check Box Style or Options, Button Style	*Text, defaulttext: Check=4, Circle=l, Cross=8, Diamond=u, Square=n, Star=H (these characters represent the respective symbols in the ZapfDingbats font)*

Table 10.2 Conversion of PDF form fields to PDFlib blocks

PDF form field attribute...	*...will be converted to the PDFlib block property*
list boxes and combo boxes	
Options, Selected (default) item	*Text, defaulttext*
buttons	
Options, Icon and Label, Label	*Text, defaulttext*

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 236).

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 in 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 will record details about the original form field as custom properties when creating the corresponding block. Table 10.3 details the custom properties which can be used to reliably identify the blocks; all properties have type *string*.

Table 10.3 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 block tool will internally maintain the relationship of form fields and blocks. When the conversion process is activated again, bound blocks will be updated to reflect the attributes of the corresponding PDF form fields. Bound blocks are useful to avoid duplicate work: when a form is updated for interactive use, the corresponding blocks can automatically be updated, too.

If you do not want to keep the converted form fields after blocks have been generated you can choose the option *Delete converted Form Fields* in the *PDFlib Blocks, Convert*

Form Fields, Conversion Options... dialog. This option will permanently remove the form fields after the conversion process. Any actions (e.g., JavaScript) associated with the affected fields will also be removed from the document.

Batch conversion. If you have many PDF documents with form fields that you want to convert to PDFlib blocks you can automatically process an arbitrary number of documents using the batch conversion feature. The batch processing dialog is available via *PDFlib Blocks, Convert Form Fields, Batch conversion...:*

- The input files can be selected individually; alternatively the full contents of a folder can be processed.
- The output files can be written to the same folder where the input files are, or to a different folder. The output files can receive a prefix to their name in order to distinguish them from the input files.
- When processing a large number of documents it is recommended to specify a log file. After the conversion it will contain a full list of processed files as well as details regarding the result of each conversion along with possible error messages.

During the conversion process the converted PDF documents will be visible in Acrobat, but you cannot use any of Acrobat's menu functions or tools.

10.4 Standard Properties for Automated Processing

PDFlib supports general properties which can be assigned to any type of block. In addition there are properties which are specific to the block types *Text*, *Image*, and *PDF*. Some properties are *shared*, which means that they can be assigned to multiple blocks at once using the Block plugin.

Properties support the same data types as option lists except handles and action lists.

Many block properties have the same name as options for *PDF_fit_image( )* (e.g., *fitmethod)* and other functions, or as PDFlib parameters (e.g., *charspacing)*. In these cases the behavior is exactly the same as the one documented for the respective option or parameter.

Property processing in PDFlib. The PDFlib Block functions *PDF_fill_*block( )* will process block properties in the following order:

- If the *backgroundcolor* property is present and contains a color space keyword different from *None,* the block rectangle will be filled with the specified color.
- All other properties except *bordercolor* and *linewidth* will be processed.
- If the *bordercolor* property is present and contains a color space keyword different from *None,* the block rectangle will be stroked with the specified color and linewidth.
- Text 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.

There will be no clipping; if you want to make sure that the block contents do not exceed the block rectangle avoid *fitmethod nofit*.

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 PDFlib (see Section 3.3.2, »Spot Colors«, page 58) 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 PDFlib application using *PDF_makespotcolor( )*. Otherwise the block functions will fail.

10.4.1 General Properties

General properties apply to all kinds of blocks *(Text, Image, PDF)*. They are required for block administration, describe the appearance of the block rectangle itself, and manage how the contents will be placed within the block. Required entries will automatically be generated by the PDFlib Block Plugin. Table 10.4 lists the general properties.

Table 10.4 General block properties

keyword	***possible values and explanation***
Block administration	
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.*
Description	*(String) Human-readable description of the block's function, coded in PDFDocEncoding or Unicode (in the latter case starting with a BOM). This property is for user information only, and will be ignored when processing the block.*

Table 10.4 General block properties

keyword	***possible values and explanation***
Locked	*(Boolean; shareable) If true, the block and its properties can not be edited with the Block plugin. This property will be ignored when processing the block. Default: false.*
Rect	*(Rectangle; required) The block coordinates. The origin of the coordinate system is in the lower left corner of the page. However, the Block plugin will display the coordinates in Acrobat's notation, i.e., with the origin in the upper left corner of the page. 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. Default:* `active`. ***active*** *The block will be fully processed according to its properties.* ***ignore*** *The block will be ignored.* ***static*** *No variable contents will be placed; instead, the block's default text, image, or PDF contents will be used if available.*
Subtype	*(Keyword; required) Depending on the block type, one of* `Text`, `Image`, *or* `PDF`
Type	*(Keyword; required) Always* `Block`
Block appearance	
background-color	*(Color; shareable) If this property is present and contains a color space keyword different from None, a rectangle will be drawn and filled with the supplied color. This may be useful to cover existing page contents. Default:* `None`
bordercolor	*(Color; shareable) If this property is present and contains a color space keyword different from* `None`, *a rectangle will be drawn and stroked with the supplied color. Default:* `None`
linewidth	*(Float; shareable; must be greater than 0) Stroke width of the line used to draw the block rectangle; only used if bordercolor is set. Default: 1*
Content placing	
fitmethod	*(Keyword; shareable) Strategy to use if the supplied content doesn't fit into the box. Possible values are* `auto`, `nofit`, `clip`, `meet`[1], `slice`[1], *and* `entire`[1]. *For simple text 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.*
orientate	*(Keyword; shareable) Specifies the desired orientation of the content when it is placed. Possible values are* `north`, `east`, `south`, `west`. *Default:* `north`
position[1]	*(Float list; shareable) One or two values specifying the position of the reference point within the content. The position is specified as a percentage within the block. Default: {0 0}, i.e. the lower left corner*
rotate	*(Float; shareable) Rotation angle in degrees by which the block will be rotated counter-clockwise before processing begins. The reference point is center of the rotation. Default: 0*

[1] *This keyword or property is not supported for Textflow blocks (text blocks with* `textflow=true`*).*

10.4.2 Text Properties

Text-related properties apply to blocks of type *Text* (in addition to general properties). All text-related properties can be shared.

Properties for all text blocks. Text blocks can contain a single line or multiple lines, depending on the *textflow* property. Table 10.5 lists the text-related properties which apply to both types.

Table 10.5 Text block properties

keyword	***possible values and explanation***
alignchar	*(Unichar or keyword) If the specified character is found in the text, its lower left corner will be aligned at the reference point. 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 0 and the keyword* `none` *suppress alignment characters. The specified* `fitmethod` *will be applied, although the text cannot be placed within the fitbox because of the forced positioning of* `alignchar`. *Default:* `none`
charspacing	*(Float or percentage) Character spacing. Percentages are based on fontsize. Default: 0*
defaulttext	*(String) Text which will be used if no substitution text is supplied by the client*[1]
escape-sequence	*(Boolean) If* `true`, *enable substitution of escape sequences in content strings, hypertext strings, and name strings. Default: the global* `escapesequence` *parameter*
fillcolor	*(Color) Fill color of the text. Default: gray 0 (=black)*
fontname[2]	*(String) Name of the font as required by PDF_load_font( ). The PDFlib Block plugin will present a list of system-installed fonts. However, these font names may not be portable between Mac, Windows, and Unix systems.* *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`
margin	*(Float list) One or two float values describing additional horizontal and vertical extensions of the text box. Default: 0*
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`
stamp	*(Keyword; will be ignored if* `boxsize` *is not specified) This option can be used to create a diagonal stamp within the box specified in the* `boxsize` *option. 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.*
strikeout	*(Boolean) Strikeout mode. Default:* `false`

Table 10.5 Text block properties

keyword	***possible values and explanation***
strokecolor	*(Color) Stroke color of the text. Default: gray 0 (=black)*
textflow	*(Boolean) Controls single- or multiline processing (Default:* `false`): ***false*** *Text can span a single line and will be processed with PDF_fit_text( ).* ***true*** *Text can span multiple lines and will be processed with PDF_fit_textflow( ). The general property position will be ignored. In addition to the standard text properties all Textflow-related properties can be specified (see Table 10.6).*
textrendering	*(Integer) Text rendering mode. Default: 0*
textrise	*(Float pr 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. The text will be interpreted in winansi encoding or Unicode.
2. This property is required in a text block; it will automatically be enforced by the PDFlib Block plugin.

Properties for Textflow blocks. Textflow-related properties apply to blocks of type *Text* where the *textflow* property is *true*. The text-related properties will be used to construct the initial option list for processing the Textflow (corresponding to the *optlist* parameter of *PDF_create_textflow( )*). Inline option lists can not be specified with the plugin, but they can be supplied on the server as part of the text contents when filling the block with *PDF_fill_textblock( )*. All Textflow-related properties can be shared. Table 10.6 lists the Textflow-related properties.

Table 10.6 Textflow block properties

keyword	***possible values and explanation***
property for text semantics	
tabalignchar	*(Integer) Unicode value of the character at which decimal tabs will be aligned. Default: the '.' character (U+002E)*
properties for controlling the text layout	
alignment	*(Keyword) Specifies formatting for lines in a paragraph. Default:* `left`. ***left*** *left-aligned, starting at* `leftindent` ***center*** *centered between* `leftindent` *and* `rightindent` ***right*** *right-aligned, ending at* `rightindent` ***justify*** *left- and right-aligned*

Table 10.6 Textflow block properties

keyword	possible values and explanation
firstlinedist	(Float, percentage, or keyword) The distance between the top of the fitbox 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 `fixedleading=true`, and the maximum of all font sizes in the line otherwise), or as a keyword. Default: `leading`. **leading** The leading value determined for the first line; typical diacritical characters such as À will touch the top of the fitbox. **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. **capheight** The capheight value determined for the first line; typical capital uppercase characters such as H will touch the top of the fitbox. **xheight** The xheight value determined for the first line; typical lowercase characters such as x will touch the top of the fitbox. If `fixedleading=false` the maximum of all `leading`, `ascender`, `xheight`, or `capheight` values found in the first line will be used.
fixedleading	(Boolean) If true, the first leading value found in each line will be used. Otherwise the maximum of all leading values in the line will be used. Default: `false`
hortabsize	(Float or percentage) Width of a horizontal tab[1]. The interpretation depends on the `hortabmethod` option. Default: 7.5%
hortabmethod	(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: `relative`. **relative** The position will be advanced by the amount specified in `hortabsize`. **typewriter** The position will be advanced to the next multiple of `hortabsize`. **ruler** The position will be advanced to the n-th tab value in the ruler option, where n is the number of tabs found in the line so far. If n is larger than the number of tab positions the relative method will be applied.
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.
lastlinedist	(Float, percentage, or keyword) Will be ignored for `fitmethod=nofit`) 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 `fixedleading= true`, and the maximum of all font sizes in the line otherwise), or as a keyword. Default: 0, i.e. the bottom of the `fitbox` will be used as baseline, and typical descenders will extend below the fitbox. **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. If `fixedleading=false` the maximum of all descender values found in the last line 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%
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
rotate	(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: 0

Table 10.6 Textflow block properties

keyword	**possible values and explanation**
ruler[2]	*(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`
tabalignment	*(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.*
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`.
properties for controlling the line-breaking algorithm	
adjustmethod	*(Keyword) Method used to adjust a line when a text portion doesn't fit into a line after compressing or expanding the distance between words subject to the limits specified by the* `minspacing` *and* `maxspacing` *options. Default:* `auto`. **auto** *The following methods are applied in order:* `shrink`, `spread`, `nofit`, `split`. **clip** *Same as* `nofit`, *except that the long part at the right edge of the fit box (taking into account the* `rightindent` *option) will be clipped.* **nofit** *The last word will be moved to the next line provided the remaining (short) line will not be shorter than the percentage specified in the* `nofitlimit` *option. Even justified paragraphs may look slightly ragged.* **shrink** *If a word doesn't fit in the line the text will be compressed subject to* `shrinklimit`. *If it still doesn't fit the nofit method will be applied.* **split** *The last word will not be moved to the next line, but will forcefully be hyphenated. For text fonts a hyphen character will be inserted, but not for symbol fonts.* **spread** *The last word will be moved to the next line and the remaining (short) line will be justified by increasing the distance between characters in a word, subject to spreadlimit. If justification still cannot be achieved the* `nofit` *method will be applied.*
linespreadlimit	*(Float or percentage; only for* `verticalalign=justify`) *Maximum amount in user coordinates or as percentage of the leading for increasing the leading for vertical justification. Default: 200%*
maxlines	*(Integer or keyword) The maximum number of lines in the fitbox, or the keyword auto which means that as many lines as possible will be placed in the fitbox. When the maximum number of lines has been placed PDF_fit_textflow( ) will return the string* `_boxfull`.
maxspacing **minspacing**	*(Float or percentage) The maximum or minimum distance between words (in user coordinates, or as a percentage of the width of the space character). The calculated word spacing is limited by the provided values (but the* `wordspacing` *option will still be added). Defaults:* `minspacing=50%`, `maxspacing=500%`
nofitlimit	*(Float or percentage) Lower limit for the length of a line with the nofit method (in user coordinates or as a percentage of the width of the fitbox). Default: 75%.*

Table 10.6 Textflow block properties

keyword	possible values and explanation
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*

1. In user coordinates, or as a percentage of the width of the fit box
2. Rulers can be edited in the »Tabs« section of the Block properties dialog.

10.4.3 Image Properties

Image-related properties apply to blocks of type *Image* (in addition to general properties). All image-related properties can be shared. Table 10.7 lists image-related properties.

Table 10.7 Image block properties

keyword	***possible values and explanation***
defaultimage	*(String) Path name of an image which will be used if no substitution image is supplied by the client. It is recommended to use file names without absolute paths, and use the SearchPath feature in the PPS client application. This will make block processing independent from platform and file system details.*
dpi	*(Float list) One or two values specifying the desired image resolution in pixels per inch in horizontal and vertical direction. With the value 0 the image's internal resolution will be used if available, or 72 dpi otherwise. This property will be ignored if the* `fitmethod` *property has been supplied with one of the keywords auto, meet, slice, or entire. Default: 0*
scale	*(Float list) One or two values specifying the desired scaling factor(s) in horizontal and vertical direction. This option will be ignored if the* `fitmethod` *property has been supplied with one of the keywords* `auto`, `meet`, `slice`, *or* `entire`. *Default: 1*

10.4.4 PDF Properties

PDF-related properties apply to blocks of type *PDF* (in addition to general properties). All PDF-related properties can be shared. Table 10.8 lists PDF-related properties.

Table 10.8 PDF block properties

keyword	***possible values and explanation***
defaultpdf	*(String) Path name of a PDF document which will be used if no substitution PDF is supplied by the client. It is recommended to use file names without absolute paths, and use the SearchPath feature in the PPS client application. This will make block processing independent from platform and file system details.*
defaultpdfpage	*(Integer) Page number of the page in the default PDF document. Default: 1*
scale	*(Float list) One or two values specifying the desired scaling factor(s) in horizontal and vertical direction. This option will be ignored if the* `fitmethod` *property has been supplied with one of the keywords* `auto`, `meet`, `slice`, *or* `entire`. *Default: 1*
pdiusebox	*(Keyword; possible values:* `media`, `crop`, `bleed`, `trim`, `art`*) Use the placed page's MediaBox, CropBox, BleedBox, TrimBox, or ArtBox for determining its size. Default:* `crop`

10.4.5 Custom Properties

Custom properties apply to blocks of any type of block (in addition to general and type-specific properties). Custom properties are optional, and can not be shared. Table 10.9 lists the naming rules for custom properties.

Table 10.9 Custom block properties

keyword	***possible values and explanation***
any name not containing the three characters [ ] /	*(String, name, float, or float list) The interpretation of the values corresponding to custom properties is completely up to the client application.*

10.5 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.

Finding the numbers and names of blocks. The client code must not even know the names or numbers of the blocks on an imported page since these can also be queried. The following statement returns the number of blocks on page with number *pagenum*:

```
blockcount = (int) p.pcos_get_number(doc,
        "length:pages[" + pagenum + "]/PieceInfo/PDFlib/Private/Blocks");
```

The following statement returns the name of block number *pagenum* on page *pagenum* (block and page counting start at 0):

```
blockname = p.pcos_get_string(doc,
        "pages[" + pagenum + "]/PieceInfo/PDFlib/Private/Blocks[" + blocknum + "]/Name");
```

The returned block name can subsequently be used to query the block's properties or populate 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 (remember that the block count will be 1 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[...]/PieceInfo/PDFlib/Private/Blocks[<number>]
pages[...]/PieceInfo/PDFlib/Private/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 + "]/PieceInfo/PDFlib/Private/Blocks/foo/Rect[0]");
lly = p.pcos_get_number(doc,
        "pages[" + pagenum + "]/PieceInfo/PDFlib/Private/Blocks/foo/Rect[1]");
urxx = p.pcos_get_number(doc,
        "pages[" + pagenum + "]/PieceInfo/PDFlib/Private/Blocks/foo/Rect[2]");
ury = p.pcos_get_number(doc,
        "pages[" + pagenum + "]/PieceInfo/PDFlib/Private/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 + "]/PieceInfo/PDFlib/Private/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 + "]/PieceInfo/PDFlib/Private/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 PDFlib property names (as defined in Section 10.4, »Standard Properties for Automated Processing«, page 238) will never conflict with custom property names.

10.6 PDFlib Block Specification

The PDFlib Block syntax is fully compliant with the PDF Reference, which specifies an extension mechanism that allows applications to store private data attached to the data structures comprising a PDF page. A detailed description of the PDFlib block syntax is provided here for the benefit of users who wish to create PDFlib blocks by other means than the PDFlib block plugin. Plugin users can safely skip this section.

10.6.1 PDF Object Structure for PDFlib Blocks

The page dictionary contains a */PieceInfo* entry, which has another dictionary as value. This dictionary contains the key */PDFlib* with an application data dictionary as value. The application data dictionary contains two standard keys listed in Table 10.10.

Table 10.10 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 10.11)*

A Block list is a dictionary containing general information about block processing, plus a list of all blocks on the page. Table 10.11 lists the keys in a block list dictionary.

Table 10.11 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 7 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 10.13). 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 PDFlib 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 action lists. Table 10.12 details how these types are mapped to PDF data types.

Table 10.12 Data types for block properties

block type	*PDF type and remarks*
boolean	*(Boolean)*
string	*(String)*
keyword	*(Name) It is an error to provide keywords outside the list of keywords supported by a particular property.*

Table 10.12 Data types for block properties

block type	***PDF type and remarks***
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.*
color	*(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:* ***/DeviceGray*** *The second element is a single gray value.* ***/DeviceRGB*** *The second element is an array of three RGB values.* ***/DeviceCMYK*** *The second element is an array of four CMYK values.* ***[/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 PDFlib, or which has been defined by the application at runtime.* ***[/Lab]*** *The first element is an array containing the keyword* `/Lab`*. The second element is an array of three Lab values.* *To specify the absence of color the respective property must be omitted.*
unichar	*(Text string) Unicode strings in* `utf16be` *format, starting with the U+FEFF BOM*

Block dictionary keys. Block dictionaries may contain the keys in Table 10.13. Only keys from one of the Text, Image or PDF groups may be present depending on the */Subtype* key in the General group (see Table 10.4).

Table 10.13 Entries in a block dictionary

key	***value***
general properties	*(Some keys are required) General properties according to Table 10.4*
text properties	*(Optional) Text and Textflow properties according to Table 10.5 and Table 10.6*
image properties	*(Optional) Image properties according to Table 10.7*
PDF properties	*(Optional) PDF properties according to Table 10.8*
Custom	*(Dictionary; optional) A dictionary containing key/value pairs for custom properties according to Table 10.9.*
Internal	*(Dictionary; optional) This key is reserved for private use, and applications should not depend on its presence or specific behavior. Currently it is used for maintaining the relationship between converted form fields and corresponding blocks.*

Example. The following fragment shows the PDF code for two blocks, a text block called *job_title* and an image block called *logo*. The text block contains a custom property called *format:*

```
<<
        /Contents 12 0 R
        /Type /Page
        /Parent 1 0 R
        /MediaBox [ 0 0 595 842 ]
        /PieceInfo << /PDFlib 13 0 R >>
>>

13 0 obj
<<
        /Private <<
            /Blocks <<
                /job_title 14 0 R
                /logo 15 0 R
            >>
            /Version 7
            /PluginVersion (3.0)
        >>
        /LastModified (D:20060813200730)
>>
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
>>
```

10.6.2 Generating PDFlib Blocks with pdfmarks

As an alternative to creating PDFlib blocks with the plugin, blocks can be created by inserting appropriate *pdfmark* commands into a PostScript stream, and distilling it to PDF. Details of the *pdfmark* operator are discussed in the Acrobat documentation. The following fragment shows *pdfmark* operators which can be used to generate the block definition in the preceding section:

```
% ---------- Setup for the blocks on a page ----------
[/_objdef {B1} /type /dict /OBJ pdfmark         % Blocks dict

[{ThisPage} <<
        /PieceInfo <<
            /PDFlib <<
```

```
                /LastModified (D:20060813200730)
                /Private <<
                    /Version 7
                    /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 03, 2006	► Updates and restructuring for PDFlib 7.0.0
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 3.03; separate COM edition of the manual
August 8, 2000	► Delphi documentation and minor additions for PDFlib 3.02
July 1, 2000	► Additions and clarifications for PDFlib 3.01
Feb. 20, 2000	► Changes for PDFlib 3.0
Aug. 2, 1999	► Minor changes and additions for PDFlib 2.01
June 29, 1999	► Separate sections for the individual language bindings ► Extensions for PDFlib 2.0
Feb. 1, 1999	► Minor changes for PDFlib 1.0 (not publicly released)
Aug. 10, 1998	► Extensions for PDFlib 0.7 (only for a single customer)
July 8, 1998	► First attempt at describing PDFlib scripting support in PDFlib 0.6
Feb. 25, 1998	► Slightly expanded the manual to cover PDFlib 0.5
Sept. 22, 1997	► First public release of PDFlib 0.4 and this manual

Index