PDFlib GmbH

PDFlib, PDFlib+PDI, PPS

A library for generating PDF on the fly

PDFlib 9.3.1

Tutorial

For use with C, C++, Java, .NET, .NET Core, Objective-C,
Perl, PHP, Python, RPG, Ruby

PDFlib GmbH
Franziska-Bilek-Weg 9, 80339 München, Germany
www.pdflib.com
phone +49 • 89 • 452 33 84-0

sales@pdflib.com
support@pdflib.com (please include your license number)

Contents

0 Applying the PDFlib License Key

Restrictions of the evaluation version. All binary versions of PDFlib, PDFlib+PDI, and PPS supplied by PDFlib GmbH can be used as fully functional evaluation versions regardless of whether or not you obtained a commercial license. However, unlicensed versions display a *www.pdflib.com* demo stamp across all generated pages, and the integrated pCOS interface is limited to small documents (up to 10 pages and 1 MB file size). Unlicensed binaries must not be used for production purposes, but only for evaluating the product. Using any PDFlib GmbH product for production purposes requires a valid license.

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

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

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

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

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

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

- In C++, Java, .NET/C#, Python and Ruby:

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

- In C:

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

- In Objective-C:

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

- In Perl and PHP:

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

- In RPG:

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

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

```
# Licensing information for PDFlib GmbH products
PDFlib license file 1.0
PDFlib      9.3.1           ...your license key...
```

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

- A file called *licensekeys.txt* will be searched in all default locations (see »Default file search paths«, page 13).
- You can specify the *licensefile* option with the *set_option()* API function:

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

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

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

- On IBM System i the license file must be encoded in ASCII (see *asciifile* option). The license file can be specified as follows (this command can be specified in the startup program *QSTRUP* and works for all PDFlib GmbH products):

  ```
  ADDENVVAR ENVVAR(PDFLIBLICENSEFILE) VALUE('/PDFlib/9.3/licensefile.txt') LEVEL(*SYS)
  ```

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

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

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

```
HKLM\SOFTWARE\PDFlib\PDFlib9\license
HKLM\SOFTWARE\PDFlib\PDFlib9\9.3.1\license
```

The installer writes the license key to the last of these entries.

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

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

Default file search paths. On Unix, Linux, macOS and IBM System i some directories will be searched for files by default even without specifying any path and directory names. Before searching and reading the UPR file (which may contain additional search paths), the following directories will be searched:

```
<rootpath>/PDFlib/PDFlib/9.3/resource/cmap
<rootpath>/PDFlib/PDFlib/9.3/resource/codelist
<rootpath>/PDFlib/PDFlib/9.3/resource/glyphlst
<rootpath>/PDFlib/PDFlib/9.3/resource/fonts
<rootpath>/PDFlib/PDFlib/9.3/resource/icc
<rootpath>/PDFlib/PDFlib/9.3
<rootpath>/PDFlib/PDFlib
<rootpath>/PDFlib
```

On Unix, Linux, and macOS <*rootpath*> will first be replaced with */usr/local* and then with the HOME directory. On IBM System i <*roothpath*> is empty.

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

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

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

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 an upgrade (change from PDFlib to PDFlib+PDI or PPS, or from PDFlib+PDI to PPS), or received a new license key as part of your support contract, 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.

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« (digit zero) to enable functionality of a higher product class for evaluation. This will enable the previously disabled functions, and re-activate the demo stamp across all pages.

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

1 Introduction

1.1 Roadmap to Documentation and Samples

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

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

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

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

Note *Most examples in this PDFlib Tutorial are provided in the Java language. Although syntax details vary with each language, the basic concepts of PDFlib programming are the same for all language bindings.*

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

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

PDFlib Cookbook. The *PDFlib Cookbook* contains hundreds of PDFlib coding fragments for solving specific problems. The Cookbook topics are available for Java and PHP, but can easily be adjusted to other programming languages since the PDFlib API is identical for all supported language bindings. The PDFlib Cookbook is available at the following URL:

`www.pdflib.com/pdflib-cookbook/`

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

```
www.pdflib.com/pcos-cookbook/
```

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

```
www.pdflib.com/tet-cookbook/
```

1.2 PDFlib Programming

What is PDFlib? PDFlib is a development component which allows you to generate files in the 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 (macOS 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, macOS and EBCDIC-based systems such as IBM System i and IBM Z. PDFlib is written in the C language, but it can be also accessed from several other languages and programming environments which are called language bindings. These language bindings cover all current Web and stand-alone application environments. The Application Programming Interface (API) is easy to learn, and is identical for all bindings. Currently the following bindings are supported:

- C and C++
- Java
- .NET and .NET Core
- Objective-C
- Perl
- PHP
- Python
- RPG (IBM System i)
- Ruby

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 PDF. However, a reasonably experienced application programmer who has dealt with any graphics API for screen display or printing shouldn't have much trouble adapting to the PDFlib API.

1.3 What's new in PDFlib/PDFlib+PDI/PPS 9.0?

The following list discusses the most important new or improved features in PDFlib/PDFlib+PDI/PPS 9.0 and Block Plugin 5. There are many more new features; see Table 1.1 and the PDFlib API Reference for details.

Create PDF/A-2 and PDF/A-3. PDFlib supports two new parts of the PDF/A standard for archiving. PDF/A-2 is based on PDF 1.7 and supports transparency, JPEG 2000 compression, layers, and many other features. While PDF/A-2 allows embedding of PDF/A-1 and PDF/A-2 documents, PDF/A-3 allows embedding of arbitrary file types.

Create Tagged PDF and PDF/UA. Creating Tagged PDF is much easier through various convenience features, such as abbreviated tagging and automatic tagging of Artifacts. PDFlib's table formatter automatically tags tables. Tagged PDF documents including structure elements can be imported with PDI.

Accessible documents can be created according to the PDF/UA standard (Universal Accessibility). PDF/UA is based on PDF 1.7 and improves Tagged PDF for accessibility.

Create PDF/VT. PDF/VT is a standard for optimized PDF for variable and transactional printing. PDFlib can create output according to ISO 16612-2 for Variable Document Printing (VDP). Document Part Metadata (DPM) can be attached according to the PDF/VT standard.

Import Scalable Vector Graphics (SVG). PDFlib imports vector graphics in the SVG format. SVG is the standard format for vector graphics on the Web.

Font handling and text output. PDFlib's font engine and text processing have been enhanced in several ways:

- ideographic variation sequences (IVS) for CJK variant glyphs
- WOFF fonts (Web Open Font Format), a new container format for TrueType and OpenType fonts specified by the W3C
- SVG fonts, i.e. vector fonts specified in SVG format
- CEF fonts (Compact Embedded Font), a variant of OpenType used for embedding fonts in SVG graphics
- automatically create UPR font configuration files with all fonts found in an arbitrary number of directories

Import PDF documents with PDFlib+PDI. The following features are new in the PDF Import library PDI:

- Tagged PDF documents including structure elements can be imported.
- Layer definitions can be imported.

PDFlib Personalization Server (PPS) and Block Plugin. The following features are new in PPS:

- The new Block type »*Graphics*« can be used to fill PDFlib Blocks with SVG graphics.
- PDFlib Blocks can not only be filled with PPS, but also imported into the output PDF.
- A few new Block properties have been introduced.

Create PDFlib Blocks programmatically. In addition to creating PDFlib Blocks interactively with the PDFlib Block Plugin, PDFlib Blocks can be created programmatically with PPS. Existing PDFlib Blocks in imported documents can be copied to the generated PDF output. These features enable advanced document composition workflows where templates for PPS can themselves be built programmatically.

PDF Object Creation API (POCA). POCA provides a set of methods for creating low-level PDF objects which are included in the generated PDF output. POCA can be used for the following purposes:

- create Document Part Metadata (DPM) for PDF/VT
- programmatically create PDFlib Blocks for use with PPS

Embed multimedia content. PDFlib can create rich media annotations with Sound, Movie, or 3D content. The multimedia content can be controlled with JavaScript and PDF actions. The following new multimedia features are available:

- rich media annotations
- rich media execute actions

Enhanced encryption algorithm. PDFlib supports PDF document encryption according to Acrobat X/XI/DC. This encryption scheme is based on AES-256 and is specified in PDF 1.7 Adobe extension level 8 and PDF 2.0 according to ISO 32000-2.

Other enhancements. The following enhancements have been implemented:

- improvements in the Table and Textflow formatters
- convenience functions for creating path objects with geometric shapes
- enhanced support for importing JPEG 2000 raster images
- query details of files in the PDFlib Virtual Filesystem (PVF)
- Removed most restrictions related to function scopes, e.g. pages, patterns and templates can now be nested arbitrarily.

1.4 What's new in PDFlib/PDFlib+PDI/PPS 9.1?

PDFlib/PDFlib+PDI/PPS 9.1 introduces new features related to color handling:

- support for *DeviceN* and *NChannel* color spaces with an arbitrary number of colorants
- PDF/X-5n for exchange of n-colorant production files, e.g. in the packaging industry
- SVG color extension for ICC profiles, spot and DeviceN color as well as Gray/RGB/CMYK device color for increased usability of SVG for print production.
- Pantone Extended Gamut Coated (XGC) spot colors and Pantone Plus 2016 update
- color shadings with an arbitrary number of stop colors for flexible color blends
- color shadings between different spot colors, e.g. blends of Pantone colors
- default color spaces for pattern, templates and Type 3 font glyphs
- extended treatment of color-related topics in the PDFlib Tutorial and Cookbook

PDFlib/PDFlib+PDI/PPS 9.1 also updates support for several language bindings.

1.5 What's new in PDFlib/PDFlib+PDI/PPS 9.2?

PDFlib/PDFlib+PDI/PPS 9.2 contains many bug fixes and improvements including the following:

- updates for most language bindings
- new language binding for .NET Core
- clarifications of structure element nesting rules in anticipation of PDF 2.0
- PDF/UA-1 implementation aligned to latest recommendations and validators
- improved import of Tagged PDF pages
- new options for modifying the color of raster images *(chromakey, decode)*
- improved color controls for non-sRGB colors in SVG
- PDF/X-4/5 convenience features for handling conflicts with identical CMYK profiles
- identify several non-standard JPEG flavors
- improved PDF/VT encapsulation of Form XObjects for better RIP performance
- optimized subsetting of TrueType fonts significantly reduces output file size, especially for fonts with a large number of unused glyphs
- identification of deprecated API functions at compile-time for C, C++, .NET, Java or at run-time for Perl and PHP (see PDFlib Migration Guide)
- overhauled the coding samples for all language bindings
- updated and extended the sample applications in the PDFlib Cookbook
- reduced memory requirements for PDFlib Mini Edition (ME) for embedded systems

1.6 What's new in PDFlib/PDFlib+PDI/PPS 9.3?

- create form field appearances as a requirement for using fields in PDF/A
- improved Tagged PDF import to correctly handle certain rare constructs which previously triggered errors in PDF/UA validators
- Tagged PDF and PDF/UA-1 enhancements according to the »Tagged PDF Best Practice Guide« published by the PDF Association
- identify all deprecated API features in anticipation of their future removal (see PDFlib Migration Guide for details)
- adjustments for new versions of operating systems and development environments
- updated and extended the sample applications in the PDFlib Cookbook
- updates for most language bindings

- bug fixes and improvements in many areas
- adjusted the Block Plugin to latest Acrobat versions and implemented several bug fixes

1.7 Features in PDFlib

Table 1.1 lists features for generating PDF. New and improved features are marked.

Table 1.1 Feature list for PDFlib

topic	features
PDF flavors	PDF 1.4 – PDF 1.7 extension level 8 and PDF 2.0
	Linearized (web-optimized) PDF for byteserving over the Web
	High-volume output and arbitrary PDF file size (beyond 10 GB)
ISO standards for PDF	ISO 32000-1: standardized version of PDF 1.7
	ISO 32000-2: PDF 2.0 (including dated revision ISO 32000-2:2020)
	ISO 15930: PDF/X-3/4/5 for the graphic arts industry
	ISO 19005-1/2/3: PDF/A-1/2/3 for archiving
	ISO 16612-2: PDF/VT-1 for variable and transactional printing
	ISO 14289-1: PDF/UA-1 for universal accessibility
Fonts	TrueType (TTF and TTC) and PostScript Type 1 fonts
	OpenType fonts with PostScript or TrueType outlines (TTF, OTF, OTC)
	WOFF fonts (Web Open Font Format)
	Support for dozens of OpenType layout features for Western and CJK text output, e.g. ligatures, small caps, old-style numerals, swash characters, simplified/traditional forms, vertical alternates
	Access fonts which are installed on Windows or macOS
	Font embedding for all font types; subsetting for TrueType, OpenType, and Type 3 fonts
	User-defined (Type 3) fonts for bitmap fonts or custom logos
	EUDC and SING fonts (glyphlets) for CJK Gaiji characters
	Fallback fonts (use missing glyphs from another font)
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
	Kerning for improved character spacing
	Artificial bold, italic, and shadow text
	Text on a path
	Configurable replacement of missing glyphs
Accessibility	Create Tagged PDF for accessibility
	Tagging of interactive elements, e.g. annotations and form fields
	Automatic table and artifact tagging
	PDF/UA-1 for universal accessibility
	Additional structure element types and attributes
Internationalization	Full Unicode support
	CJK fonts and CMaps for Chinese, Japanese, and Korean text
	Support for a variety of 8-bit and legacy multi-byte CJK encodings (e.g. Shift-JIS; Big5)
	Ideographic variation sequences (IVS) for CJK variant glyphs
	Vertical writing mode for Chinese, Japanese, and Korean text
	Character shaping for complex scripts, e.g. Arabic, Thai, Devanagari
	Bidirectional text formatting for right-to-left scripts, e.g. Arabic and Hebrew
SVG vector graphics	Import vector graphics in SVG format; ICC profiles; CMYK and spot colors in SVG, CSS
Images	Load BMP, GIF, PNG, TIFF, JBIG2, JPEG, JPEG 2000, and CCITT raster images
	Query image information (pixel size, resolution, ICC profile, clipping path, etc.)

Table 1.1 Feature list for PDFlib

topic	features
	Use clipping path in TIFF and JPEG images
	Use alpha channel (transparency) in TIFF and PNG images
	Image masks (transparent images with a color applied), colorize images with a spot or DeviceN color
Color	Grayscale, RGB (numerical, hexadecimal, HTML color names), CMYK, CIE L*a*b* color
	Integrated support for PANTONE® and HKS® colors
	DeviceN (n-colorant) color space based on process or spot colors
	User-defined spot colors
	Color shadings (smooth shadings) between process colors or spot colors; pattern fills and strokes
Color management	ICC-based color with ICC profiles
	Rendering intent for text, graphics, and raster images
	ICC profiles as output intent for PDF/A and PDF/X; multi-colorant profiles for PDF/X-5n
Archiving	PDF/A-1a/1b, PDF/A-2a/b/u and PDF/A-3a/b/u
	XMP extension schemas for PDF/A
Graphic arts	PDF/X-3, PDF/X-4, PDF/X-4p, PDF/X-5n
	Embedded or externally referenced output intent ICC profile
	External graphical content (referenced pages) for PDF/X-5p and PDF/X-5pg
	Overprint and text knockout
Variable Document Printing (VDP)	PDF/VT-1 for variable and transactional printing
Textflow Formatting	Format text into one or more rectangular or arbitrarily shaped areas with hyphenation (user-supplied hyphenation points required), font and color changes, justification methods, tabs, leaders
	Advanced line-breaking with language-specific processing
	Flexible image placement and formatting
	Wrap text around images or image clipping paths
Table 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, SVG graphics, PDF pages, path objects, annotations, and form fields
	Table cells can be formatted with ruling and shading options
	Flexible stamping function
	Matchbox concept for referencing the coordinates of placed images or other objects
Vector graphics	Common vector graphics primitives: lines, curves, arcs, ellipses, rectangles, etc.
	Transparency (opacity) and blend modes
	Reusable path objects and clipping paths imported from images
Layers	Optional page content which can selectively be displayed
	Annotations and form fields can be placed on layers
Security	Encrypt PDF document or attachments
	Unicode passwords
	Document permission settings, e.g. printing or copying not allowed
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

Table 1.1 Feature list for PDFlib

topic	features
	Create PDF annotations (comments) such as PDF links, launch links (other document types), Web links
	Named destinations for links, bookmarks, and document open action
	Create page labels (symbolic names for pages)
Multimedia	*Embed 3D animations in PDF*
	Embed Sound and Movie in PDF and control it with JavaScript
Georeferenced PDF	*Create PDF with geospatial reference information*
Metadata	*Document information: common fields (Title, Subject, Author, Keywords) and user-defined fields*
	Create XMP metadata from document info fields or XMP streams
	User-supplied custom XMP metadata
	Process XMP image metadata in TIFF, JPEG, JPEG 2000 and SVG graphics
Programming	*Language bindings for C, C++, Java, .NET and .NET Core, Objective-C, Perl, PHP, Python, RPG, Ruby*
	Virtual file system for supplying data in memory, e.g., images from a database
	Generate PDF documents on disk file or directly in memory
Embedded Systems	*PDFlib Mini Edition (ME) for embedded systems with reduced resource requirements*

1.8 Additional Features in PDFlib+PDI

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

Table 1.2 Additional features in PDFlib+PDI

topic	features
PDF input (PDI)	Import pages from existing PDF documents
	Import all PDF versions up to PDF 1.7 extension level 8 (Acrobat X/XI/DC) and PDF 2.0
	Import documents which are encrypted with any of PDF's standard encryption algorithms
	Query information about imported pages
	Clone page geometry of imported pages (e.g. BleedBox, TrimBox, CropBox)
	Delete redundant objects (e.g. identical fonts) across multiple imported PDF documents
	Repair malformed input PDF documents
	Copy PDF/A or PDF/X output intent from imported PDF documents
	Import pages from Tagged PDF documents including structure elements
	Import layer definitions (optional content)
pCOS interface	pCOS interface for querying details about imported PDF documents

1.9 Additional Features in PPS

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

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

topic	***features***
Variable Document Printing (VDP)	*PDF personalization with PDFlib Blocks for text, image, PDF data or SVG vector graphics*
	Create PDFlib Blocks programmatically with PPS
	Copy PDFlib Blocks from imported documents
PDFlib Block Plugin	*PDFlib Block Plugin for creating PDFlib Blocks interactively in Acrobat on Windows and macOS*
	Preview PPS Block filling in Acrobat
	Copy Blocks to Preview file
	Snap-to-grid for interactively creating or editing Blocks in Acrobat
	Clone PDF/X or PDF/A properties of the Block container
	Convert PDF form fields to PDFlib Blocks for automated filling
	Textflow Blocks can be linked so that one Block holds the overflow text of a previous Block
	PANTONE® and HKS® spot color names integrated in the Block plugin
	Support for Retina displays on macOS

1.10 Availability of Features in different Products

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

Table 1.4 Availability of features in different products

feature	*API functions and options*	*PDFlib*	*PDFlib+PDI*	*PPS*
basic PDF generation	*all except those listed below*	X	X	X
linearized (Web-optimized) PDF	`linearize` *option in* PDF_*begin_document( )*	X[1]	X	X
optimize PDF (only relevant for inefficient client code and non-optimized imported PDF documents)	`optimize` *option in* PDF_*end_document( )*	X[1]	X	X
Parsing attachments for PDF/A conformance	PDF_*load_asset( ) with* `type=Attachment` *in PDF/A-2 mode*	X[1]	X	X
Parsing PDF documents for Portfolio creation	`password` *option in* PDF_*add_portfolio_file( )*	X[1]	X	X
PDF import (PDI)	*all PDI functions*	–	X	X
Query information from PDF with pCOS	*all pCOS functions*	–	X	X
Fill Blocks with variable data	PDF_*fill_*block( )*	–	–	X
Create Blocks programmatically	PDF_*poca_new( ): option* `usage=blocks` PDF_*begin/end_page_ext( ): option* `blocks`	–	–	X
Copy Blocks to generated output	PDF_*process_pdi( ): option* `action=copyblock` *or* `action=copyallblocks`	–	–	X
interactively create PDFlib Blocks for use with PPS	*PDFlib Block Plugin for Acrobat*	–	–	X

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

2 PDFlib Language Bindings

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

2.1 C Binding

PDFlib is written in C with some C++ modules. In order to use the PDFlib C binding, you can use a static or shared library (DLL on Windows and MVS), and you need the central PDFlib include file *pdflib.h* for inclusion in your PDFlib client source modules. Alternatively, *pdflibdl.h* can be used for dynamically loading the PDFlib DLL at runtime (see next section for details).

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

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

PDF_delete(p);
```

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

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

output file is left in an incomplete state! Since this is not adequate for a library routine, for serious PDFlib projects it is strongly advised to leverage PDFlib's exception handling facilities. A user-defined catch clause may, for example, present the error message in a GUI dialog box, and take other measures instead of aborting.

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

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

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

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

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

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

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

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

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

        if (error_condition)
        {
```

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

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.

In order to create a shared library from the static library on Linux use the following commands:

```
mkdir tmp
cd tmp
ar x ../libpdf.a
g++ -shared -o libpdf.so *
```

This results in a shared library which no longer requires the application to be linked against the C++ runtime library.

2.2 C++ Binding

In addition to the *pdflib.h* C header file, an object-oriented wrapper for C++ is supplied for PDFlib clients. It requires the *pdflib.hpp* header file, which in turn includes *pdflib.h*. Since *pdflib.hpp* contains a template-based implementation no corresponding *.cpp* module is required.

String handling in C++. The new template-based C++ binding supports the following usage patterns with respect to string handling:

- Strings of the C++ standard library type *std::wstring* are used as basic string type. They can hold Unicode characters encoded as UTF-16 or UTF-32. This is the default behavior since PDFlib 8 and the recommended approach for new applications unless custom data types (see next item) offer a significant advantage over *wstrings*.
- Custom (user-defined) data types for string handling can be used as long as the custom data type is an instantiation of the *basic_string* class template and can be converted to and from Unicode via user-supplied converter methods. As an example a custom string type implementation for UTF-8 strings is included in the PDFlib distribution *(pstring_utf8.cpp)*.

The interface assumes that all strings passed to and received from PDFlib methods are native *wstrings*. Depending on the size of the *wchar_t* data type, *wstrings* are assumed to contain Unicode strings encoded as UTF-16 (2-byte characters) or UTF-32 (4-byte characters). Literal strings in the source code must be prefixed with *L* to designate wide strings. Unicode characters in literals can be created with the *\u* and *\U* syntax.

Note On EBCDIC-based systems the formatting of option list strings for the `wstring`*-based interface requires additional conversions to avoid a mixture of EBCDIC and UTF-16* `wstrings` *in option lists. Convenience code for this conversion and instructions are available in the auxiliary module* `utf16num_ebcdic.hpp`*. Users should also take a look a the compiler option* `CONVLIT(,UNICODE)` *which controls conversion of string literals in C++ code to Unicode.*

Error handling in C++. PDFlib API functions 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 API function which threw the exception.

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

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

Using PDFlib as a DLL loaded at runtime. Similar to the C language binding the C++ binding allows you to dynamically attach PDFlib to your application at runtime (see »Using PDFlib as a DLL loaded at runtime«, page 31). Dynamic loading can be enabled as follows when compiling the application module which includes *pdflib.hpp:*

```
#define PDFCPP_DL    1
```

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

2.3 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.

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 the classes *pdflib* and *PDFlibException*. 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=. hello
```

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

Error handling in Java. The Java binding 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 {

...PDFlib instructions...

} catch (PDFlibException e) {
```

```
            System.err.print("PDFlib exception occurred in hello sample:\n");
            System.err.print("[" + e.get_errnum() + "] " + e.get_apiname() +
                             ": " + e.get_errmsg() + "\n");

} catch (Exception e) {
        System.err.println(e.getMessage());

} finally {
        if (p != null) {
            p.delete();                     /* delete the PDFlib object */
        }
}
```

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

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

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

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

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

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

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

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

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

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

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

2.4 .NET Binding

2.4.1 .NET Binding Variants

The PDFlib binding for .NET is available in two variants:

- .NET Core binding based on C# Interop
- Classic .NET binding based on C++ Interop

Both .NET bindings differ in implementation details and supported target environments according to Table 2.1. Based on this information you can choose the binding which is best suited for your application.

Table 2.1 Comparison of the classic .NET binding and .NET Core binding

	Classic .NET binding based on C++ Interop	***.NET Core binding based on C# Interop***
download package	*Windows installer*	*platform-specific* `zip` *or* `tar.gz` *package*
package contents	*assembly, documentation, samples*	*NuGet package with assembly, documentation, samples*
implementation	*C++/CLI assembly* `pdflib_dotnet.dll` *with unmanaged code*	*C# assembly* `PDFlib_dotnet.dll` *with managed code and auxiliary DLL* `PDFlib_dotnetcore_native.dll` *with unmanaged code*
.NET integration	*C++ Interop via implicit PInvoke*	*C# Interop via explicit PInvoke*
support for .NET Framework	*.NET Framework 4.x*	*.NET Framework 4.6.1 and above*
support for .NET Core	*n/a*	*.NET Standard 2.0*
target operating systems	*Windows x86 and x64*	*Windows x64, Linux x64, macOS, Alpine Linux x64*
Windows registry handling	*installer registers PDFlib and adds the license key to the registry*	*PDFlib registration not required; registry entries for the license key must be added manually*
class name	`PDFlib_dotnet`	`PDFlib_dotnet`
support for deprecated PDFlib 4-8 API methods	*yes*	*no*

2.4.2 .NET Core Binding

Note The .NET Core binding is delivered as a universal package for all supported platforms. However, it still requires a platform-specific license which cannot be transferred across platforms.

PDFlib for .NET Core supports the .NET Standard 2.0 which implies support for .NET Core 2.0, .NET Framework 4.6.1, Mono 5.4 and corresponding newer versions, as well as many other environments. You need the .NET Core SDK for the desired target platform.

The version scheme used for the .NET Core binding conforms to .NET versioning rules. The .NET Core version numbers are visible e.g. in the NuGet cache and *.csproj* project files. These version numbers are not identical to PDFlib major and minor release numbers. A mapping between both versioning schemes can be found in *compatibility.txt*.

The product is supplied as NuGet package which can be installed locally using any of the following methods:

- The *dotnet* command-line tool (all platforms). This method is detailed in the next section.
- Visual Studio's Package Manager UI (Windows and macOS)
- Visual Studio's Package Manager Console (Windows)
- The *nuget* command-line tool (all platforms)

The project files for the supplied samples are prepared for target framework .NET Core 2.0 (target framework moniker TFM=*netcoreapp2.0)*. You may want to adjust the TFM in the project files if you want to target different frameworks (e.g. *net471)*.

Installing PDFlib for .NET Core with the dotnet command-line tool. We describe the installation, configuration and build process with the *dotnet* utility, using the supplied *hello* project as an example:

- Unpack the compressed product package in a directory of your choice.
- In a command shell *cd* to the hello project directory:

  ```
  cd <installdir>\bind\dotnetcore\csharp\hello
  ```

- (This step is not required for the supplied samples which reference the package with a local *NuGet.Config* file) Copy the NuGet package to the application's project directory:

  ```
  <installdir>/bind/dotnetcore/PDFlib_dotnet.X.Y.Z.nupkg
  ```

- (This step is not required for the supplied samples which already contain a reference to PDFlib) Enter the following command with the appropriate version number:

  ```
  dotnet add package PDFlib_dotnet.X.Y.Z
  ```

  This command adds a PDFlib reference to the *.csproj* project file. It also installs PDFlib in the local NuGet package cache if it is not yet present, e.g.

  ```
  ~/.nuget/packages/pdflib_dotnet/X.Y.Z
  ```

  Because of this caching you must copy the **.nupkg* only for the first project. Subsequent projects don't require the package file since it is taken from the cache.
- Now you can build and run the *hello* project to test it:

  ```
  dotnet build
  dotnet run
  ```

  As a result you will find the generated *hello.pdf* output document in the application directory.

2.4.3 Classic .NET Binding

Note Detailed information about the classic .NET binding can be found in the PDFlib-in-.NET-HowTo.pdf document which is contained in the distribution packages and also available on the PDFlib Web site.

Installing the Classic .NET Binding. Install PDFlib with the supplied Windows Installer. It installs the PDFlib assembly plus auxiliary data files, documentation and samples on the machine interactively. The installer also registers PDFlib so that it can easily be referenced on the .NET tab in the *Add Reference* dialog box of Visual Studio.

Referencing the .NET binding in a C# project. In order to use the .NET binding in a C# project you must create a reference to the PDFlib assembly as follows in Visual C# .NET: *Project, Add Reference..., Browse...,* and select *pdflib_dotnet.dll* from the installation directory. With the command line compiler you can reference PDFlib as in the following example:

```
csc.exe /r:..\..\bin\pdflib_dotnet.dll hello.cs
```

2.4.4 Using the .NET Binding in Applications

This section applies to both variants of the .NET binding. Full examples with ready-to-use configuration are included in all packages.

Once the .NET binding is properly referenced you can use the *PDFlib_dotnet.PDFlib* and *PDFlib_dotnet.PDFlibException* classes.

Error handling in .NET. The .NET binding supports .NET exceptions and throws an exception with a detailed error message when a runtime problem occurs. The client is responsible for catching such an exception and properly reacting on it. Otherwise the .NET framework will catch the exception and usually terminate the application.

In order to convey exception-related information PDFlib defines its own exception class *PDFlib_dotnet.PDFlibException* with the members *get_errnum*, *get_errmsg*, and *get_apiname*. PDFlib implements the *IDisposable* interface which means that clients can call the *Dispose( )* method for cleanup.

Client code can handle exceptions thrown by PDFlib with the usual *try...catch* construct:

```
try {
        ...PDFlib instructions...
catch (PDFlibException e)
{
        // caught exception thrown by PDFlib
        Console.WriteLine("PDFlib exception occurred in hello sample:\n");
        Console.WriteLine("[{0}] {1}: {2}\n",
            e.get_errnum(), e.get_apiname(), e.get_errmsg());
} finally {
        if (p != null) {
            p.Dispose();
        }
}
```

Unicode and legacy encoding conversion. For the convenience of PDFlib users we show a useful C# string conversion method. Please refer to the .NET documentation for more details. The following constructor creates a Unicode string from a byte array (at the specified offset and length), using the encoding supplied in the *Encoding* parameter:

```
public String(sbyte*, int, int, Encoding)
```

2.5 Objective-C Binding

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

- *PDFlib* for use on macOS
- *PDFlib_ios* for use on iOS

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

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

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

or

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

In order to embed the PDFlib framework in an app XCode's code signing expects a framework with the version number *A* while PDFlib products use numeric version numbers. In order to get around this you can create an appropriately named framework folder as follows:

```
cd PDFlib.framework/Versions
mv 9.3.1 A
rm Current
ln -s A Current
```

Parameter naming conventions. For PDFlib method calls you must supply parameters according to the following conventions:

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

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

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

corresponds to the following Objective-C method:

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

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

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

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

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

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

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

2.6 Perl Binding

The PDFlib wrapper for Perl consists of a C wrapper file and two Perl package modules, one for providing a Perl equivalent for each PDFlib API function and another one for the PDFlib object. The C module is used to build a shared library which the Perl interpreter loads at runtime, with some help from the package file. Perl scripts refer to the shared library module via a *use* statement.

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

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

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

Unix. Perl will search *pdflib_pl.so* (on macOS: *pdflib_pl.bundle)*, *pdflib_pl.pm* and *PDFlib/PDFlib.pm* in the current directory or the directory printed by the following Perl command:

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

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

```
/usr/lib/perl5/site_perl/5.32/i686-linux
```

Windows. The DLL *pdflib_pl.dll* and the modules *pdflib_pl.pm* and *PDFlib/PDFlib.pm* will be searched in the current directory or the directory printed by the following Perl command:

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

Typical output of the above command looks like

```
C:\Program Files\Perl5.32\site\lib
```

Error Handling in Perl. The Perl binding 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 {
        ...PDFlib instructions...
};
if ($@) {
        die("$0: PDFlib Exception occurred:\n$@");
}
```

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

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

```
use Unicode::String qw(utf8 utf16 uhex);
...
$p->set_option("stringformat=utf8");
$font = $p->load_font("Arial Unicode MS", "unicode", "");
$p->setfont($font, 24.0);
$p->fit_textline(uhex("U+65E5 U+672C U+8A9E"), $x, $y, "");
```

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

```
$p->set_option("textformat=utf16le");
$font = $p->load_font("Arial Unicode MS", "unicode", "");
$p->setfont($font, 24.0);
$p->fit_textline("\xE5\x65\x2C\x67\x9E\x8A", $x, $y, "");
```

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

```
$p->set_option("searchpath={{../../../resource/cmap}}");
$font = $p->load_font("Arial Unicode MS", "cp932", "");
$p->setfont($font, 24.0);
$p->fit_textline("\x93\xFA\x96\x7B\x8C\xEA", $x, $y, "");
```

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

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

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

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

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

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

2.7 PHP Binding

Installing the PDFlib PHP Edition. Detailed information about the various flavors and options for using PDFlib with 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=php_pdflib.so        ; for Unix and macOS
extension=php_pdflib.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 *PDFlib*.
- Load PDFlib at runtime with one of the following lines at the start of your script:

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

Modified error return for PDFlib functions in PHP. Since PHP uses the convention of returning the value 0 (FALSE) when an error occurs within a function, all PDFlib functions have been adjusted to return 0 instead of -1 in case of an error. This difference is noted in the function descriptions in the PDFlib API Reference. However, take care when reading the code fragment examples in Section 3, »Creating PDF Documents«, page 51, since they use the usual PDFlib convention of returning -1 in case of an error.

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

- PHP on Unix systems will find files without any path component in the directory where the script is located.
- PHP on Windows will find files without any path component only in the directory where the PHP DLL is located. Note that PDFlib expects UTF-8-encoded file names, while PHP's *dirname( )* function usually returns *WinAnsi* or some other host encoding. In this case you must convert directory or file names to UTF-8; see Section 5.2.2, »Language Bindings with UTF-8 Support«, page 107, for details. Example:

```
$searchpath = dirname(dirname(__FILE__)).'/data';
$searchpath = $p->convert_to_unicode("auto", $searchpath, "outputformat=utf8");
```

In order to provide platform-independent file name handling the use of PDFlib's *SearchPath* facility is strongly recommended (see Section 3.1.4, »Resource Configuration and File Search«, page 55).

Exception handling in PHP. Since PHP supports structured exception handling, PDFlib exceptions will be propagated as PHP exceptions. PDFlib will throw an exception of the 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 {
        $p = new PDFlib();
        ...PDFlib instructions...

} catch (PDFlibException $e) {
        print "PDFlib exception occurred:\n";
        print "[" . $e->get_errnum() . "] " . $e->get_apiname() . ": "
                $e->get_errmsg() . "\n";
}
catch (Throwable $e) {
        die("PHP exception occurred: " . $e->getMessage() . "\n");
}
```

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

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

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

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

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

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

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

2.8 Python Binding

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

- Unix and macOS: *pdflib_py.so*
- Windows: *pdflib_py.pyd*

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

- *PDFlib/PDFlib.py*
- *PDFlib/__init__.py* (only for Python 2.7)

Error Handling in Python. The Python binding throws a *PDFlibException* in case of an error. The Python exceptions can be dealt with by the usual *try/catch* technique:

```
try:
        p = PDFlib()

        ...PDFlib instructions...

except PDFlibException:
        print("PDFlib exception occurred:"
        print("[%d] %s: %s" % (ex.errnum, ex.apiname, ex.errmsg))

except Exception:
        print(ex)

finally:
        if p:
            p.delete()
```

2.9 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.

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

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

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

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

```
d/copy 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 and PDFLIB_RPG service programs 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 and PDFLIB_RPG service programs 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):

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

Before executing the generated program you should apply the following command:

```
chgcurdir '/pdflib/pdflib/x.y/x.y.z/bind/rpg'
```

The generated PDF documents can be found in the same directory.

Parameters must be passed to the PDFlib API according to the data types listed in Table 2.2.

Table 2.2 Data types in the RPG binding

API data type	***data types in the RPG binding***
string data type	*Unicode string (use* `%ucs2`)
binary data type	`data`

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

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

2.10 Ruby Binding

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

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

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

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

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

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

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

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

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

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

- Edit *config/routes.rb:*

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

- Edit *app/controllers/home_controller.rb* as follows and insert PDFlib code for creating PDF contents. Keep in mind that the PDF output must be generated in memory, i.e. an empty file name must be supplied to *begin_document( ).* As a starting point you can use the code in the *hello-rails.rb* sample:

```
class HomeController < ApplicationController
def demo
require "PDFlib"
```

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

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

  ```
  $ rails server
  ```

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

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

3 Creating PDF Documents

3.1 General PDFlib Programming Aspects

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

3.1.1 Exception Handling

Errors of a certain kind are called exceptions in many languages for good reasons – they are mere exceptions, and are not expected to occur very often during the lifetime of a program. The general strategy is to use conventional error reporting mechanisms (i.e. special error return codes such as -1) for function calls which may often fail, and use a special exception mechanism for those rare occasions which don't warrant cluttering the code with conditionals. This is exactly the path that PDFlib goes: Some operations can be expected to go wrong rather frequently, for example:

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

PDFlib signals such errors by returning a special value (usually –1, but 0 in the PHP binding) as documented in the PDFlib API Reference. This error code must be checked by the application developer for all functions which are documented to return -1 on error.

Other events may be considered harmful, but will occur rather infrequently, e.g.

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

When PDFlib detects such a situation, an exception will be thrown instead of passing a special error return value to the caller. It is important to understand that the generated PDF document cannot be finished when an exception occurred. The only methods which can safely be called after an exception are *PDF_delete()*, *PDF_get_apiname()*, *PDF_get_errnum()*, and *PDF_get_errmsg()*. Calling any other PDFlib method after an exception may lead to unexpected results. The exception will contain the following information:

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

Querying the reason of a failed function call. As noted above, the generated PDF output document must always be abandoned when an exception occurs. However, when a function reports a non-fatal problem by returning an error value, the document can be continued. The client application is free to continue the document by adjusting the program flow or supplying different data. For example, when a particular font cannot be loaded most clients will give up the document, while others may prefer to work with a different font. In this case it may be desirable to retrieve an error message which describes the problem in more detail. In this situation the functions *PDF_get_errnum()*, *PDF_get_errmsg()*, and *PDF_get_apiname()* can be called immediately after a failed function call, i.e., a function call which returned a -1 (in PHP: 0) error value.

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

- *errorpolicy=legacy:* this deprecated setting ensures behavior which is compatible to earlier versions of PDFlib. Some functions return an error code, while others throw an exception according to the respective API description. The *legacy* setting is the default error policy.
- *errorpolicy=return:* when an error condition is detected the function returns with a -1 (in PHP: 0) error value. 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 is thrown when an error condition is detected. The output document will be incomplete and unusable after an exception. This can be used for lazy programming without any error conditionals at the expense of sacrificing the output document even for problems which may be fixable by the application.

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

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

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

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

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

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

Warnings. Some problem conditions are detected by PDFlib internally, but do not justify interrupting the program flow by throwing an exception. Instead of throwing an exception, a description of the condition is logged (see Section 3.1.2, »Logging«, page 53, for more details about the logging feature). 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 Logging

PDFlib can create a log file which records the following items:

- all API calls with corresponding parameters and options; note that logging may contain additional API calls issued by the PDFlib language wrapper (e.g. for string conversion).
- return values and handles returned by API functions;
- a timestamp for each call;
- information regarding the use of deprecated API features;
- warning messages which don't justify an exception, but should be examined during the development phase of an application;
- details about internal operation which may be useful for investigating support cases.

The contents of a log file may can be an important aid for application developers for identifying problems in the program flow. Logging can be enabled at runtime as follows:

```
p.set_option("logging={filename={mylogfile.log}}");
```

Alternatively, logging can be enabled via environment variables. The amount of logged information can be controlled with various logging classes and levels for each class; see PDFlib API Reference for details.

Logging should not normally be enabled in production situations, but only during the development phase and when problems need to be analyzed. PDFlib GmbH support may request log files to discuss user problems.

The following excerpt shows lines from a typical log file with default logging classes:

```
PDF_load_image(p_0x2201c20, "auto", "nesrin.jpg", /*c*/0, "")
[0]

PDF_begin_page_ext(p_0x2201c20, 10.000000, 10.000000, "")
[Begin page #1]
PDF_fit_image(p_0x2201c20, 0, 0.000000, 0.000000, "adjustpage")
PDF_close_image(p_0x2201c20, 0)
PDF_end_page_ext(p_0x2201c20, "")
[End page #1]

PDF_end_document(p_0x2201c20, "")
[Full product name: "PDFlib Personalization Server"]
[Document ID: <C98301CB2D4EAC2972D34CAAAE929021> <C98301CB2D4EAC2972D34CAAAE929021>]
[Size of document: 34842 bytes]
[End document #1]
```

3.1.3 The PDFlib Virtual File System (PVF)

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

In addition to disk files a facility called *PDFlib Virtual File System* (PVF) allows clients to directly supply data in memory without any disk files involved. This offers performance benefits and can be used for data fetched from a database which does not even exist on

an isolated disk file, as well as other situations where the client already has the required data available in memory as a result of some processing.

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

- Raster image files: */pvf/image/filename*
- font outline and metrics files (it is recommended to use the actual font name as the base portion of the file name): */pvf/font/filename*
- ICC profiles: */pvf/iccprofile/filename*
- 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_end_document( )*. 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_end_document( )*. The second call is required because PDFlib may still need access to the data so that the first call refuses to unlock the virtual file. However, in some cases the first call will already free the data, and the second call doesn't do any harm. The client may free the file contents only when *PDF_delete_pvf( )* succeeded.
- Optimize performance by reusing virtual files: some clients may wish to reuse some data (e.g., font definitions) within various output documents, and avoid multiple create/delete cycles for the same file contents. In this case it is recommended not to call *PDF_delete_pvf( )* as long as more PDF output documents using the virtual file will be generated.

- Lazy programming: if memory usage is not a concern the client may elect not to call *PDF_delete_pvf( )* at all. In this case PDFlib will internally delete all pending virtual files in *PDF_delete( )*.

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

Creating PDF output in a virtual file. In addition to supplying user data to PDFlib, PVF can also hold the PDF document data generated by PDFlib. This can be achieved by supplying the *createpvf* option to *PDF_begin_document( )*. The PVF file name can later be supplied to other PDFlib API functions. This is useful, for example, when generating PDF documents for inclusion in a PDF Portfolio. It is not possible to directly retrieve the PVF data created by PDFlib; use the active or passive in-core PDF generation interface to fetch PDF data from memory (see Section 3.1.5, »Generating PDF Documents in Memory«, page 60).

3.1.4 Resource Configuration and File Search

In most advanced applications PDFlib needs access to resources such as font files, 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_option( )*. For the configuration file PDFlib uses a simple text format called *Unix PostScript Resource* (UPR). We extended the original UPR format for our purposes. The UPR file format as used by PDFlib is described below.

With the *enumeratefonts* option PDFlib can be instructed to collect all fonts which are accessible on the search path (see »File search and the SearchPath resource category«, page 56). Using the *saveresources* option the current list of PDFlib resources can be written to a file:

```
/* add font directory to the search path */
p.set_option("searchpath={{C:/fonts}}");

/* enumerate all fonts on the searchpath and create a UPR file */
p.set_option("enumeratefonts saveresources={filename={C:/fonts/pdflib.upr}}");
```

Resource categories. The resource categories supported by PDFlib are listed in Table 3.1. Most categories map a resource name (to be used in the PDFlib API) to the name of a virtual or disk-based file. Resource categories other than those in Table 3.1 are ignored. Category names are case-insensitive. The values are treated as name strings; they can be encoded in ASCII or UTF-8 (with BOM at the start of a line), or in EBCDIC-UTF-8 on IBM Z. Unicode values may be useful for localized font names with the *HostFont* resource.

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

- Lines can have a maximum of 1023 characters.
- A backslash character '\' at the end of a line cancels the line end. This may be used to extend lines.

Table 3.1 Resource categories supported in PDFlib

category	*format*	*explanation*
SearchPath	`pathname`	*relative or absolute path name of directories containing data files*
CMap	`cmapname=filename`	*CMap file for CJK encoding*
FontAFM	`fontname=filename`	*PostScript font metrics file in AFM format*
FontPFM	`fontname=filename`	*PostScript font metrics file in PFM format*
FontOutline	`fontname=filename`	*PostScript, TrueType, OpenType, WOFF, or CEF font outline file*
Encoding	`encodingname=filename`	*text file containing an 8-bit encoding or code page table*
HostFont	`fontname=hostfontname`	*name of a font installed on the system (usually both font names are identical)*
FontnameAlias	`aliasname=fontname`	*create an alias for a font which is already known to PDFlib*
ICCProfile	`profilename=filename`	*name of an ICC color profile*

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

UPR files consist of the following components:

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

```
PS-Resources-1.0
```

- An optional section listing all resource categories described in the file. Each line describes one resource category. The list is terminated by a line with a single period character. Available resource categories are described below. If this optional section is not present, a single period character must be present nevertheless.
- A section for each of the resource categories listed at the beginning of the file. Each section starts with a line showing the resource category, followed by an arbitrary number of lines describing available resources. The list is terminated by a line with a single period character. Each resource data line contains the name of the resource (equal signs have to be quoted). If the resource requires a file name, this name has to be added after an equal sign. The *SearchPath* (see below) will be applied when PDFlib searches for files listed in resource entries.

File search and the SearchPath resource category. PDFlib reads a variety of data items, such as raster images, font outline and metrics information, 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 tries 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 are 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_option("SearchPath={{/path/to/dir1} {/path/to/dir2}}");
```

The search path can be set multiply, and multiple directory names can be supplied in a single call. It is recommended to use double braces even for a single entry to avoid problems with directory names containing space characters. An empty string list (i.e. {{}}) deletes all existing search path entries including the default entries.

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

  ```
  HKLM\SOFTWARE\PDFlib\PDFlib9\9.3.1\SearchPath
  HKLM\SOFTWARE\PDFlib\PDFlib9\SearchPath
  HKLM\SOFTWARE\PDFlib\SearchPath
  ```

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

  ```
  /PDFlib/PDFlib/9.3/resource/icc
  /PDFlib/PDFlib/9.3/resource/fonts
  /PDFlib/PDFlib/9.3/resource/cmap
  /PDFlib/PDFlib/9.3
  /PDFlib/PDFlib
  /PDFlib
  ```

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

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

```
<rootpath>/PDFlib/PDFlib/9.3/resource/cmap
<rootpath>/PDFlib/PDFlib/9.3/resource/codelist
<rootpath>/PDFlib/PDFlib/9.3/resource/glyphlst
<rootpath>/PDFlib/PDFlib/9.3/resource/fonts
<rootpath>/PDFlib/PDFlib/9.3/resource/icc
<rootpath>/PDFlib/PDFlib/9.3
<rootpath>/PDFlib/PDFlib
<rootpath>/PDFlib
```

On Unix, Linux, and macOS <*roothpath*> will first be replaced with */usr/local* and then with the HOME directory. On IBM System i <*roothpath*> is empty.

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

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

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

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

```
PS-Resources-1.0
.
SearchPath
/usr/local/lib/fonts
C:/psfonts/pfm
/users/kurt/my_images
.
FontOutline
ArialMT=Arial.ttf
.
HostFont
Wingdings=Wingdings
.
ICCProfile
highspeedprinter=cmykhighspeed.icc
.
```

Searching for the UPR resource file. If only the built-in resources (e.g., PDF core font, 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_option( )* (see below) or in a UPR resource file. PDFlib reads this file automatically when the first resource is requested. The detailed process is as follows:

- On Unix systems, macOS and IBM System i some directories will be searched by default for license and resource files even without specifying any path and directory names. Before searching and reading the UPR file, the following directories will be searched (in this order):

  ```
  <rootpath>/PDFlib/PDFlib/9.3/resource/icc
  <rootpath>/PDFlib/PDFlib/9.3/resource/fonts
  <rootpath>/PDFlib/PDFlib/9.3/resource/cmap
  <rootpath>/PDFlib/PDFlib/9.3
  <rootpath>/PDFlib/PDFlib
  <rootpath>/PDFlib
  ```

  On Unix systems and macOS <*roothpath*> will first be replaced with */usr/local* and then with the HOME directory. On IBM System i <*roothpath*> is empty. This feature can be used to work with a license file, UPR file, or resources without setting any environment variables or runtime options.

- If the environment variable *PDFLIBRESOURCEFILE* 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 *PDFLIBRESOURCEFILE* 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 System i)
  pdflib.upr (Windows, Unix, and all other systems)
  ```

  If this file cannot be read no exception will be thrown.
- On Windows PDFlib will additionally try to read the following registry entries which will be searched in the order provided below:

  ```
  HKLM\Software\PDFlib\PDFlib9\9.3.1\resourcefile
  HKLM\Software\PDFlib\PDFlib9\resourcefile
  HKLM\Software\PDFlib\resourcefile
  ```

  The values of these entries will be taken as the name of the resource file to be used. If this file cannot be read an exception will be thrown. Be careful when manually accessing the registry on 64-bit Windows systems: as usual, 64-bit PDFlib binaries will work with the 64-bit view of the Windows registry, while 32-bit PDFlib binaries running on a 64-bit system will work with the 32-bit view of the registry. If you must add registry keys for a 32-bit product manually, make sure to use the 32-bit version of the *regedit* tool. It can be invoked as follows from the *Start* dialog:

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

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

  ```
  p.set_option("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 *PDF_set_option()*. 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_option("FontOutline={Foobar-Bold=foobb.otf}");
```

Note *Font configuration is discussed in more detail in Section 6.4.4, »Searching for Fonts«, page 140.*

Querying resource values. In addition to setting resource entries you can query values using *PDF_get_option()*. Specify the category name as key and the number of the resource (starting at 1) as option. For example, the following call:

```
idx = p.get_option("SearchPath", "resourcenumber=" + n);
sp = p.get_string(idx, "");
```

retrieves 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.

3.1.5 Generating PDF Documents in Memory

In addition to generating PDF documents on a file, PDFlib can also be instructed to generate the PDF directly in memory *(in-core)*. This technique offers performance benefits since no disk-based I/O is involved, and the PDF document can, for example, directly be streamed via HTTP. Webmasters will be especially happy to hear that their server will not be cluttered with temporary PDF files.

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

You can use the *createpvf* option to create PDF data in memory and subsequently pass it to PDFlib without writing a disk file (see »Creating PDF output in a virtual file«, page 55).

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( )*.

3.1.6 Maximum Size of PDF Documents and other Limits

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

- 10 GB file size limit: PDF 1.4 documents are limited by the cross-reference table to 10 decimal digits and therefore 10^{10}-1 bytes, which equates to roughly 9.3 GB. If you plan to create output documents beyond 10 GB you must use PDF 1.5 or above. This version supports cross-reference streams which are no longer subject to the 10-digits limit and therefore allow creation of PDF documents beyond 10 GB.
- Number of objects: while the object count in a document is not limited by PDF in general, the PDF/A, PDF/X-4 and PDF/X-5 standards limit the number of indirect objects in a document to 8.388.607. If a document requires objects beyond this limit PDFlib will throw an exception in PDF/A, PDF/X-4 and PDF/X-5 mode. In other modes documents with more objects can always be created. This check can be disabled with the document option *limitcheck=false*.
  The number of objects in PDF depends on the complexity of the page contents, number of interactive elements, etc. Since typical high-volume documents with simple contents require ca. 4-10 objects per page on average, documents with ca. 1-2 million pages can be created without exceeding the object limit mandated by the standards.

PDF limits. PDFlib imposes limits on certain entities in order to create PDF output which conforms to the limitations imposed by the PDF Reference, Acrobat, or some PDF standard. These limits are documented below.
The following limits are enforced by suitably modifying the values:

- Smallest absolute floating point value in PDF: 0.000015. Numbers with a smaller absolute value are replaced with 0.
- (PDF 1.4, but not newer PDF versions) Largest absolute value which can be expressed as floating point number in PDF: 32767.0. Numbers with a larger absolute value are replaced with the closest integer.

The PDF format imposes certain restrictions. Exceeding one of the following limits results in an exception:

- Largest allowed numerical value in PDF: 2.147.483.647
- Maximum length of hypertext strings: 65535
- Maximum length of text strings on the page: 32.763 bytes (i.e. 16.381 characters for CID fonts) if *kerning=false* and *wordspacing=0*; otherwise 4095 characters
- The following options are limited to a maximum of 8191 list entries:
  views, namelist, polylinelist, fieldnamelist, itemnamelist, itemtextlist, children, group
- Maximum number of indirect objects in PDF/A-1/2/3 and PDF/X-4/5: 8.388.607

3.1.7 Multi-threaded Programming

The threading behavior of PDFlib can be characterized as follows: While PDFlib itself is single-threaded, it can safely be used in multi-threaded applications. In the common situation that a PDFlib object is only used within one thread, no particular multi-threading precautions are necessary. If the same PDFlib object will be used within multiple threads the application must synchronize the threads to make sure that the PDFlib

object is not accessed simultaneously by more than one thread. A typical scenario would involve a pool of PDFlib objects where each thread fetches an existing PDFlib object from the pool instead of creating a new one, and returns it to the pool after creating a document if the object is no longer needed. Using the same PDFlib object in another thread before the output document is finished will rarely provide any advantage for the application, and is not recommended.

3.1.8 Using PDFlib on EBCDIC-based Platforms

The operators and structure elements in the PDF file format are based on ASCII which doesn't work well with EBCDIC-based platforms such as IBM System i and IBM Z (but not zLinux which is based on ASCII). However, a special mainframe version of PDFlib is available in order to allow mixing of ASCII-based PDF operators and EBCDIC (or other) text output. The EBCDIC-safe version of PDFlib is available for various operating systems and machine architectures.

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

- 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 in ASCII format you can set the *asciifile* option to *true* (default is *false* on IBM Z and *true* on IBM System i). PDFlib will then expect these files in ASCII encoding. String parameters are still expected in EBCDIC encoding, however.

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

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

3.2 Page Descriptions

3.2.1 Coordinate Systems

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

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

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

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

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

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

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

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

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

```
p.set_option("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 calculates 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* option has been set to *true*.

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

You can visualize page coordinates in Acrobat as follows:

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

Note that the coordinates displayed refer to an origin in the top left corner of the page, and not PDF's default origin in the lower left corner. See »Using top-down coordinates«, page 64, for details on selecting a coordinate system which aligns with Acrobat's coordinate display.

Rotating objects. It is important to understand that objects cannot be modified once they have been drawn on the page. Although there are PDFlib functions for rotating, translating, scaling, and skewing the coordinate system, these do not affect existing objects on the page but only subsequently drawn objects.

Rotating text, images, and imported PDF pages can easily be achieved with the *rotate* option of *PDF_fit_textline( )*, *PDF_fit_textflow( )*, *PDF_fit_image( )*, and *PDF_fit_pdi_page( )*. Rotating such objects by multiples of 90 degrees inside the respective fitbox can be accomplished with the *orientate* option of these functions. The following example generates some text at an angle of 45° degrees:

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

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

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

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

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

In order to facilitate the use of top-down coordinates PDFlib supports a special mode in which all relevant coordinates will be interpreted differently. The *topdown* feature has been designed to make it quite natural for PDFlib users to work in a top-down coordinate system. Instead of working with the default PDF coordinate system with the origin *(o, o)* at the lower left corner of the page and *y* coordinates increasing upwards, a modified coordinate system will be used which has its origin at the upper left corner of the page with *y* coordinates increasing downwards. This top-down coordinate system for a page can be activated with the *topdown* option of *PDF_begin_page_ext( ):*

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

For the sake of completeness we'll list the detailed consequences of establishing a top-down coordinate system below.

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

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

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

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

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

3.2.2 Page Size

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

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

Page size limits. Although PDF and PDFlib don't impose any restrictions on the usable page size, Acrobat implementations suffer from architectural limits regarding the page size. Other PDF interpreters may be able to deal with larger or smaller document formats. The page size limits for Acrobat are shown in Table 3.2. In PDF 1.6 and above the *userunit* option in *PDF_begin/end_page_ext( )* can be used to specify a global scaling factor for the page.

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):

Table 3.2 Minimum and maximum page size of Acrobat

PDF viewer	*minimum page size*	*maximum page size*
without `userunit` *option (default)*	*1/24" = 3 pt = 0.106 cm*	*200" = 14400 pt = 508 cm*
with `userunit` *option*	*3 user units*	*14 400 user units* *The maximum value 75 000 for* `userunit` *allows page sizes up to 14 400 * 75 000 = 1 080 000 000 points = 381 km*

- 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}");
```

3.2.3 Direct Paths and Path Objects

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

- Stroking draws a line along the path, using client-supplied options (e.g., color, line width) for drawing.
- Filling paints the entire region enclosed by the path, using client-supplied options for filling.
- Clipping reduces the imageable area for subsequent drawing operations by replacing the current clipping area (which is unlimited by default) with the intersection of the current clipping area and the area enclosed by the path.
- Merely terminating the path results in an invisible path, which will nevertheless be present in the PDF file. This will only rarely be useful.

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

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

set 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.set_graphics_option("strokecolor=red");
p.moveto(100, 100);
p.lineto(200, 100);
p.stroke();
```

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

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

Path objects. Path objects are a convenient and powerful alternative to direct paths. They encapsulate all drawing operations for constructing a path. Path objects can be created in different ways:

- *PDF_add_path_point()* adds a point and associated path element to a path object. This function can also add a reference to an existing path object to a newly constructed path. *PDF_add_path_point()* supports several convenience options to facilitate path construction. The function also accepts SVG path descriptions. The following code fragment creates a simple path object with a circle, strokes it at two different locations on the page, and finally deletes it:

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

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

- A clipping path contained in a raster image can be retrieved with *PDF_info_image()* and the keyword *clippingpath:*

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

path = (int) p.info_image(image, "clippingpath", "");
if (path == -1)
     throw new Exception("Error: clipping path not found!");

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

- The enclosing rectangle (bounding box) of a placed PDF page, SVG graphics, path, raster image, table, matchbox, Textflow or Textline can be retrieved with the corresponding *PDF_info_*()* function:

```
optlist = "boxsize={400 300} fitmethod=clip matchbox={name=border}";
p.fit_image(image, 200, 150, optlist);

int path = (int) p.info_matchbox("border", 1, "boundingbox");
p.draw_path(path, 0, 0, "close stroke linewidth=10 strokecolor=red");
p.delete_path(path);
```

Once a path object has been created or retrieved it can be used for different purposes:

- *PDF_draw_path( )* can be used to fill or stroke the path on the page, or use it as a clipping path.
- Wrap multi-line Text around the text with *PDF_fit_textflow( )*: the text is formatted so that it wraps inside or outside of an arbitrary shape (see Section 9.2.11, »Wrapping Text around Paths and Images«, page 247).
- Place text on the path with *PDF_fit_textline( )*, i.e. the characters follow the curves of the path (see Section 9.1.7, »Text on a Path«, page 227).
- Place path objects in table cells with *PDF_add_table_cell( )*.

Unlike direct paths, path objects can be used multiply until they are explicitly destroyed with *PDF_delete_path( )*. Information about a path can be retrieved with *PDF_info_path( )*.

3.2.4 Templates (Form XObjects)

Templates (Form XObjects). 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 directly acting on a page). When the template is finished it can be used 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 are recommended 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. Templates are also recommended for raster images with a clipping path if the image is placed more than once. Templates can be created in the following ways:

- directly with *PDF_begin_template_ext( );*
- indirectly from vector graphics with *PDF_load_graphics( )* and the *templateoptions* option;
- indirectly from raster images with *PDF_load_image( )* and the *templateoptions* option; without this option *PDF_load_image( )* creates a similar PDF construct called Image XObject.

Note PDF pages imported with PDF_open_pdi_page( ) also create PDF Form XObjects, but these are handled with PDI functions, not template functions.

Templates can be used in the following ways:

- place the template on a page or another content stream with *PDF_fit_image( )* (see below);
- create a graphics state with a luminosity soft mask defined by the template (suboption *template* of option *softmask* of *PDF_create_gstate( )*, see Section 4.9.2, »Changing the Color with Soft Masks«, page 99);
- as fallback (background) for loading SVG graphics (option *fallbackimage* of *PDF_load_graphics( ));*
- as appearance of annotations (suboptions *normal/rollover/down* of option *template* of *PDF_create_annotation( ));*

- as appearance of pushbutton form fields (options *icon/icondown/iconrollover* of *PDF_create_field( )*)

Creating and using templates. A template can be placed on the page or on another template with the *PDF_fit_image( )* function just like raster images (see Section 8.4, »Placing Images, Graphics, and imported PDF Pages«, page 213). The general idiom for creating and using templates in PDFlib looks as follows:

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

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

- *PDF_begin_item( )* and the *tag* option of various functions: structure elements cannot be created within a template.
- All interactive functions: these must be defined on the page where they should appear in the document, and cannot be generated as part of a template.

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

3.3 PDF Password Security

3.3.1 Password Security in PDF

PDF password security offers the following protection features:

- The user password (also referred to as open password) is required to open the file for viewing. Only files with a user password are safe from cracking!
- The master password (also referred to as owner or permissions 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 viewing by supplying either password.
- Permission settings restrict certain actions for the PDF document, such as printing or extracting text.
- An attachment password can be specified to encrypt only file attachments, but not the actual contents of the document itself.

If a PDF document uses any of these protection features it will be encrypted. In order to display or modify a document's security settings with Acrobat, click *File, Properties..., Security, Show Details...* or *Change Settings...*, respectively.

Encryption algorithms and key lengths. PDF encryption makes use of the following encryption algorithms:

- RC4, a symmetric stream cipher (i.e. the same algorithm can be used to encrypt and decrypt). RC4 no longer offers adequate security and has been deprecated in PDF 2.0.
- AES (Advanced Encryption Standard) as specified in the standard FIPS-197. AES is a modern block cipher which is used in a variety of applications.

Since the actual encryption keys are unwieldy binary sequences, they are derived from more user-friendly passwords which consist of plain characters. In the course of PDF and Acrobat development the PDF encryption methods have been enhanced to use stronger algorithms, longer encryption keys, and more sophisticated passwords. Table 3.3 details encryption, key and password characteristics for all PDF versions.

Table 3.3 Encryption algorithms, key length, and password length in PDF versions

PDF and Acrobat version, pCOS algorithm number	***encryption algorithm and key length***	***max. password length and password encoding***
PDF 1.1 - 1.3 (Acrobat 2-4), pCOS algorithm 1	*RC4 40-bit (weak; deprecated in PDF 2.0)*	*32 characters (Latin-1)*
PDF 1.4 (Acrobat 5), pCOS algorithm 2	*RC4 128-bit (weak; deprecated in PDF 2.0)*	*32 characters (Latin-1)*
PDF 1.5 (Acrobat 6), pCOS algorithm 3	*same as PDF 1.4, but different application of encryption method (weak, deprecated in PDF 2.0)*	*32 characters (Latin-1)*
PDF 1.6 (Acrobat 7) and PDF 1.7 = ISO 32000-1 (Acrobat 8), pCOS algorithm 4	*AES-128 (weak; deprecated in PDF 2.0)*	*32 characters (Latin-1)*
PDF 1.7ext3 (Acrobat 9), pCOS algorithm 9	*AES-256 with shortcomings in password handling (weak,; deprecated in PDF 2.0)*	*127 UTF-8 bytes (Unicode)*
PDF 1.7ext8 (Acrobat X/XI/DC) and PDF 2.0 = ISO 32000-2, pCOS algorithm 11	*AES-256 with improved password handling*	*127 UTF-8 bytes (Unicode)*

Passwords. PDF encryption internally works with encryption keys of 40, 128, or 256 bit depending on the PDF version. The binary encryption key is derived from a password provided by the user. The password is subject to length and encoding constraints:

- Up to PDF 1.7 (ISO 32000-1) passwords were restricted to a maximum length of 32 characters and could contain only characters from the Latin-1 encoding.
- PDF 1.7ext3 introduced Unicode characters and bumped the maximum length to 127 bytes in the UTF-8 representation of the password. Since UTF-8 encodes characters with a variable length of 1-4 bytes the allowed number of Unicode characters in the password is less than 127 if it contains non-ASCII characters. For example, since Japanese characters usually require 3 bytes in UTF-8 representation, up to 42 Japanese characters can be used in passwords.

In order to avoid ambiguities, Unicode passwords are normalized by a process called *SASLprep* (specified in RFC 4013 based on *Stringprep* in RFC 3454). This process eliminates non-text characters and normalizes certain character classes (e.g. non-ASCII space characters are mapped to the ASCII space character U+0020). The password is normalized to Unicode normalization form NFKC, and special bidirectional processing is applied to avoid ambiguities which may otherwise arise if right-to-left and left-to-right characters are mixed in a password.

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.

Permission restrictions. PDF can encode various restrictions on document operations which can be granted or denied individually:

- *Printing Allowed*: If printing is not allowed, the print button in Acrobat remains disabled. Acrobat supports a distinction between *Low Resolution (150 dpi)* and High Resolution printing. Low-resolution printing generates a raster image of the page which is suitable only for personal use, but prevents high-quality reproduction. Note that image-based printing not only results in low output quality, but also considerably slows down the printing process.
- *Changes Allowed:* the corresponding list provides control over various document modification operations:

  ```
  Inserting, deleting, and rotating pages
  Filling in form fields and signing existing signature fields
  Commenting, filling in form fields, and signing existing signature fields
  Any except extracting pages
  ```

  Note that Adobe Reader does not reliably display permission restrictions. For example, since it does not include assembly functions, it always displays *Document Assembly: Not allowed* regardless of the actually permission settings in a document.
- Content copying is controlled via *Enable copying of text, images, and other content*. While this can be enabled for accessibility with *Enable text access for screen reader devices for the visually impaired*, this setting is deprecated in PDF 2.0 since a PDF reader should always support accessibility.

Specifying access restrictions for a document, such as *Printing allowed: None* 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 are honored. Indeed, several PDF tools are known to ignore permission settings altogether; commercially available PDF cracking tools can be used to disable all 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 described as follows in ISO 32000-1:

»Once the document has been opened and decrypted successfully, a conforming reader technically has access to the entire contents of the document. There is nothing inherent in PDF encryption that enforces the document permissions specified in the encryption dictionary.«

Encrypted document components. By default, PDF encryption always covers all components of a document. However, there are use cases where it is desirable to encrypt only some components of the document, but not others:

- PDF 1.5 (Acrobat 6) introduced a feature called plaintext metadata. With this feature encrypted documents can contain unencrypted document XMP metadata. This is for the benefit of search engines which can retrieve document metadata even from encrypted documents.
- Since PDF 1.6 (Acrobat 7) file attachments can be encrypted even in otherwise unprotected documents. This way an unprotected document can be used as a container for confidential attachments.

Security recommendations. Keep in mind that only PDFs with a user password (required to open the document) are safe from cracking. The following recommendations should be obeyed to avoid which encryption which could be cracked:

- Passwords consisting of 1-6 characters should be avoided since they are susceptible to attacks which try all possible passwords (brute-force attack against the password).
- Passwords should not resemble a plain text word since the password would be susceptible to attacks which try all plaintext words (dictionary attack). Passwords should contain non-alphabetic characters. Don't use your spouse's or pet's name, birthday, or other items which are easy to determine.
- The modern AES algorithm is preferable over the older RC4 algorithm.
- AES-256 according to PDF 1.7ext3 (Acrobat 9) should be avoided because it contains a weakness in the password checking algorithm which facilitates brute-force attacks against the password. For this reason Acrobat X/XI/DC and PDFlib never use Acrobat 9 encryption for protecting new documents (only for decrypting existing documents).

In summary, AES-256 according to PDF 1.7ext8/PDF 2.0 should be used. Passwords should be longer than 6 characters and should contain non-alphabetic characters.

Protecting PDFs on the Web. When PDFs are served over the Web users can always produce a local copy of the document with their browser. There is no way for a PDF document to prevent users from saving a local copy.

3.3.2 Password-Protecting PDF Documents with PDFlib

PDFlib can apply standard security features when generating PDF documents. In order to import pages from protected PDF documents with PDFlib+PDI or PDFlib Personalization Servers (PPS) the master password or the *shrug* option is required. Querying document properties with the pCOS interface is governed by the pCOS mode. For example, XMP document metadata, document info fields, bookmarks, and annotation contents can be retrieved without the master password if the document does not require a user password (or only the user password has been supplied). The pCOS Path Reference discusses this in more detail.

Note You cannot Reader-enable PDF documents (e.g. allow annotations with Adobe Reader) with PDFlib products.

Encryption algorithm and key length. The encryption algorithm and key length used to protect a document with a password depend on the PDF version of the generated document, which in turn depends on the *compatibility* option of *PDF_begin_document()*. The encryption algorithm is selected as follows:

- PDF 1.4 and 1.5: the respective flavor of RC4 encryption with 128-bit keys is used.
- PDF 1.6, PDF 1.7 and PDF 1.7ext3: AES-128 is used. Note that AES-256 according to PDF 1.7ext3 (Acrobat 9) is never used because of known weaknesses.
- PDF 1.7ext8 and PDF 2.0: AES-256 according to Acrobat X/XI/DC is used.

Setting passwords with PDFlib. Passwords can be set with the *userpassword* and *masterpassword* options in *PDF_begin_document()*. PDFlib interacts with client-supplied passwords for the generated document in the following ways:

- If a user password or permission settings, but no master password has been supplied, a regular user would easily be able to change the security settings, thereby defeating any protection. For this reason PDFlib considers this situation as an error.
- If the 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.
- Unicode passwords are allowed for AES-256. All older encryption algorithms require passwords which are restricted to the Latin-1 character set. An exception will be thrown for older encryption algorithms if the supplied password contains characters outside the Latin-1 character set.
- Passwords will be truncated to 127 UTF-8 bytes for AES-256, and to 32 characters for older encryption algorithms.

Setting permissions with PDFlib. Access restrictions can be set with the *permissions* option in *PDF_begin_document()*. It contains one or more access restriction keywords. When setting the *permissions* option the *masterpassword* option must also be set, because otherwise Acrobat users could easily remove the permission settings. By default, all actions are allowed. Specifying an access restriction will disable the respective feature in Acrobat. Access restrictions can be applied without any user password. Multiple restriction keywords can be specified as in the following example:

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

Table 3.4 lists all supported access restriction keywords.

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

Table 3.4 Access restriction keywords for the `permissions` *option of 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 accessibility.*
noannots	*Acrobat will prevent adding or changing comments or form fields.*
noforms	*(Implies* `noannots`*) Acrobat will prevent form field filling, even if noannots hasn't been specified.*
noaccessible	*(Deprecated in PDF 2.0) Acrobat will prevent extracting text or graphics for accessibility purposes.*
noassemble	*(Implies* `nomodify`*) Acrobat will prevent inserting, deleting, or rotating pages and creating bookmarks and thumbnails, even if* `nomodify` *hasn't been specified.*
nohiresprint	*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.*

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

4 Color Spaces

Color spaces can be specified separately for filling and stroking operations, e.g. with the *fillcolor* and *strokecolor* options. While color spaces in PDF are also used for images, PDFlib usually determines the image color space automatically unless color-related options are supplied in *PDF_load_image()*.

Cookbook *Code samples for color handling can be found in the* `color` *category of the PDFlib Cookbook. The* `color/starter_color` *sample demonstrates the use of all color spaces.*

4.1 Device Color Spaces

Device-specific color spaces are perhaps the most common methods for describing color. As the name implies they describe color as rendered by a specific device such as a monitor, printer or printing machine. Device-specific colors appear differently depending on which device produces the output. Because of this undesirable property device colors are not allowed directly in PDF/A and PDF/X, but only with a suitable output intent ICC profile or default color space.

Grayscale color space. Grayscale colors can be requested with the color space keyword *gray* and a single gray value. Gray values are in the range 0=black and 1=white. Example:

```
p.set_graphics_option("fillcolor={{ gray 0.5 }");
```

RGB color space. Colors in the additive RGB color space are mixed from the red, green and blue components. RGB colors can be requested with the color space keyword *rgb* and three RGB values in the range 0..1 specifying the percentage of red, green, and blue with *(0, 0, 0)*=black and *(1, 1, 1)*=white. The commonly used RGB color values in the range 0–255 must be divided by 255 in order to scale them to the range 0–1 as required by PDFlib.

As an alternative to numerical RGB values you can specify RGB colors via their HTML names or hexadecimal values. Examples:

```
p.set_graphics_option("fillcolor={{ rgb 1 0 0 }");
p.set_graphics_option("fillcolor=pink");
p.set_graphics_option("fillcolor={#FFC0CB}");
```

CMYK color space. Colors in the subtractive CMYK color space are mixed from the cyan, magenta, yellow and black (»key«) components. CMYK values can be requested with the color space keyword *cmyk* and four CMYK values in the range 0..1, where 0 = no color and 1 = full color, representing cyan, magenta, yellow, and black values; *(0, 0, 0, 0)*= white, *(0, 0, 0, 1)*=black. Note that the polarity of CMYK colors i different from grayscale and RGB colors. Example:

```
p.set_graphics_option("fillcolor={{ cmyk 0 1 0 0 }");
```

A special property of the CMYK color space is that the overprint behavior of CMYK objects can be controlled with the *overprintmode* option (see »Overprint mode for CMYK colors«, page 102).

Mapping device colors to device-independent colors with default color spaces. Device color spaces can be mapped to device-independent color spaces by means of default color spaces; see »Mapping device colors to ICC-based color spaces«, page 81, for details. This is useful when data with device-specific colors, such as RGB or CMYK images or graphics, is used in PDF/A or PDF/X without any matching output intent.

4.2 Color Management with ICC Profiles

PDFlib supports color management with ICC profiles and rendering intents. ICC profiles play a crucial role in color-managed workflows and most PDF standards such as PDF/X and PDF/A.

Cookbook *Full code samples can be found in the* `starter_color` *sample and the Cookbook topic* `color/iccprofile_to_image`.

ICC profiles. The International Color Consortium (ICC) 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 for the use cases listed in Table 4.1. Color management does not change the number of components in a color specification (e.g., from RGB to CMYK).

Note *Recommendations and links to freely available ICC color profiles for common printing conditions are available at www.pdflib.com.*

Table 4.1 Different uses of ICC profiles

Use case	*Relevant API functions and options*
Set ICC-based color space for text and vector graphics on the page	*PDF_set_option( ) with* `iccprofilegray/rgb/cmyk` *and PDF_setcolor( ) with* `colorspace=iccbasedgray/rgb/cmyk`; *color options with* `iccbased` *keyword*
Apply ICC profile to an imported image	*PDF_load_image( ): option* `iccprofile`
Process or ignore ICC profile embedded in an image	*PDF_load_image( ): option* `honoriccprofile`
Query ICC profile embedded in an image	*PDF_info_image( ) with* `keyword=iccprofile`
Set default color space for mapping grayscale, RGB, or CMYK data to ICC-based color spaces	*PDF_begin_page_ext( ), PDF_begin_template_ext( ), PDF_begin_pattern_ext( ) and PDF_begin_font( ): options* `defaultgray/defaultrgb/defaultcmyk`
Specify a PDF/X or PDF/A output intent with a referenced or embedded ICC profile	*PDF_load_iccprofile( ): option* `usage=outputintent`
Specify a blending color space for transparency and blend modes	*PDF_begin/end_page_ext( ), PDF_open_pdi_page( ), PDF_begin_template_ext( ) and PDF_load_graphics( ) with* `templateoptions`: *option* `transparencygroup`, *suboption* `colorspace`
Alternate color space for spot color	*PDF_set_graphics_option( ): color option with keyword* `spotname` *and* `iccbased` *alternate color, or PDF_set_graphics_option( ) with option* `fillcolor` *followed by PDF_makespotcolor( ).*
Alternate color space for DeviceN color	*PDF_create_devicen( ): option* `alternate` *with suboption* `iccbased`
Query number of color components in an ICC profile	*PDF_get_option( ) with keyword* `icccomponents`

Acceptable ICC profiles. Color profiles must satisfy certain conditions regarding the ICC version number of the profile, its device class, and its data color space. The ICC version number is restricted as follows:

- PDF output compatibility 1.4: ICC version 2.x
- PDF output compatibility 1.5 and above: ICC version 2.x or 4.x

Table 4.2 details additional requirements regarding device class and data color space for ICC profiles depending on their intended use.

Table 4.2 Acceptable ICC profiles for various uses

ICC profile usage	*device class*	*data color space*
output intent for PDF/X-3/4 and PDF/X-5p/5pg	`prtr`	*Gray, RGB, CMYK*
output intent for PDF/X-5n	`prtr`	*xCLR (n-colorant)*
output intent for PDF/A	`prtr, mntr`	*Gray, RGB, CMYK*
transparency group color space	`prtr, mntr, scnr, spac`	*Gray, RGB, CMYK*
all other uses of ICC profiles	`prtr, mntr, scnr, spac`	*Gray, RGB, CMYK*

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

- If *profilename=sRGB*, PDFlib uses the internal sRGB profile and terminates 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>                (only on Windows and macOS)
<colordir>/<filename>.icc            (only on Windows and macOS)
<colordir>/<filename>.icm            (only on Windows and macOS)
```

On Windows *colordir* designates the directory where device-specific ICC profiles are stored by the operating system (e.g. *C:\Windows\System32\spool\drivers)*. On macOS the following paths are tried for *colordir:*

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

The sRGB color space and sRGB ICC profile. PDFlib supports the industry-standard RGB color space called sRGB. It 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*. As a convenient shortcut the keyword *srgb* can be supplied as an alternative in all places where an ICC profile handle created with *PDF_load_iccprofile( )* is expected.

The sRGB profile is of device class *mntr* (output device), i.e. it can be used as output intent for PDF/A, but not for PDF/X.

Using embedded profiles in images (ICC-tagged images). Some images 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 handles embedded ICC profiles in the JPEG, JPEG 2000, PNG and TIFF image file formats. If the *honoriccprofile* option is set to *true* (which is the default) the ICC profile embedded in an image is extracted from the image and embedded in the PDF output.

The keyword *iccprofile* of *PDF_info_image( )* can be used to obtain an ICC profile handle for the profile embedded in an image. This may be useful when the same profile must be applied to multiple images.

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

Applying external ICC profiles to images. 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 *iccprofilegray, iccprofilergb, iccprofilecmyk* options. Subsequently ICC-based color values can be supplied to a color option or *PDF_setcolor( )* along with one of the color space keywords *iccbasedgray*, *iccbasedrgb*, or *iccbasedcmyk:*

```
p.set_option("errorpolicy=return");
icchandle = p.load_iccprofile("myCMYK", "usage=iccbased");
if (icchandle == -1)
{
        return;
}
p.set_graphics_option("fillcolor={iccbased=" + icchandle + " 0 1 0 0}");
```

Mapping device colors to ICC-based color spaces. PDF provides a feature for mapping device-dependent gray, RGB, or CMYK colors to device-independent ICC-based colors. This can be used to attach a precise colorimetric specification to color values which otherwise would be device-dependent. It can be achieved by supplying the *defaultgray, defaultrgb,* or *defaultcmyk* options of *PDF_begin_page_ext( ), PDF_begin_template_ext( ), PDF_begin_pattern_ext( )* and *PDF_begin_font( )* with a suitable ICC profile handle. The following example sets the sRGB color space as the default RGB color space for text, images, and vector graphics on the page:

```
p.begin_page_ext(595, 842, "defaultrgb=srgb");
```

If the default color space is derived from an external ICC profile, a profile handle must first be created:

```
/* create ICC profile handle*/
icchandle = p.load_iccprofile("myRGB", "usage=iccbased");
p.begin_page_ext(595, 842, "defaultrgb=" + icchandle);
```

Output intents for PDF/X and PDF/A. An output device profile can be used to specify an output condition for PDF/X or PDF/A. This is done by supplying *usage=outputintent* in the call to *PDF_load_iccprofile()*. For PDF/A a printer or monitor profile can be specified as output intent, while PDF/X allows only printer profiles. For details see Section 12.4, »PDF/X for Print Production«, page 331, and Section 12.3, »PDF/A for Archiving«, page 319.

4.3 Device-Independent CIE L*a*b* Color

Cookbook *A full code sample can be found in the* `starter_color sample`.

Device-independent color values can be specified in the CIE 1976 L*a*b* color space *(Lab* for short). Colors in the L*a*b* color space are specified by three values *L*, *a* and *b* (see Figure 4.1). The luminance (or lightness) *L* runs in the range 0-100. The values *a* and *b* in the range -128 to 127 describe the color. The value *a* runs from green (-128) to magenta-red (+127); *b* runs from blue (-128) to yellow (+127). Positive numbers describe warm colors (yellow, magenta-red), while negative numbers describe cool colors (green, blue). The intersection of the *a* and *b* axes *(a=b=0)* describes neutral gray values ranging from black (0, 0, 0) to white (100, 0, 0).

An important property of the Lab color space is that unlike RGB and CMYK it is perceptually uniform, which means that the same numerical difference corresponds to the same visual difference between colors. This makes it much easier to calculate mixtures of multiple colors simply by calculating the weighted average of their Lab values. We will take advantage of this when creating a PostScript transform function in »DeviceN color space based on spot colors with Lab alternate space«, page 90.

Lab values specify absolute colors only if the white point is also specified. PDFlib uses the white point of the standard illuminant D50 (daylight 5000 K, 2° observer).

PDFlib supports the Lab color space as follows:

- Lab colors for text and vector graphics can be specified in color options and *PDF_setcolor( )* using the color space keyword *lab* plus three numbers for the *L*, *a* and *b* values, e.g.

  ```
  p.set_graphics_option("fillcolor={lab 100 0 0}");
  ```

  In PDF/A mode the default black fill and stroke colors are specified as device-independent Lab (0, 0, 0) unless an output intent has been specified.
- PDFlib uses the Lab color space for the alternate color values of all HKS and Pantone spot colors in its internal database.
- TIFF images with Lab color space can be imported.
- ICC profiles with Lab color space can be loaded with *PDF_load_iccprofile( )* unless they are intended for use as output intent or for the *transparencygroup* option since PDF doesn't allow Lab in these places.

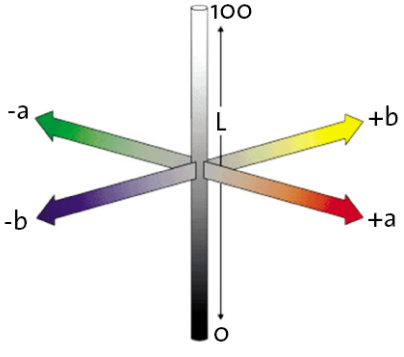


Fig. 4.1
*The CIE L*a*b* color space*

4.4 Pantone, HKS, and custom Spot Colors

Cookbook *Full code samples can be found in the* `starter_color` *sample and the Cookbook topic* `color/spot_color`.

Separation color space for spot colors. PDFlib supports spot colors (technically known as Separation color space in PDF) 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 must be accompanied by an alternate color which closely, but not exactly, resembles the spot color. The PDF viewer or RIP uses 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 is applied in addition to any process colors which may be used in the document.

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 first checks whether the requested spot color can be found in one of its built-in libraries. If so, PDFlib uses built-in Lab 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 in the *spotname* color keyword of color options (or via the current fill color when using *PDF_setcolor( )*. Spot colors can be tinted, i.e., they can be used with a percentage which specifies the color intensity from 0=no color=white to 1=maximum intensity. Separation color spaces are always subtractive (unlike additive color spaces like RGB where 0=black). Once a spot color has been defined, it can be used to draw text and vector graphics, to colorize grayscale images, or to construct a shading. Note that PDFlib automatically inverts the color polarity when colorizing images with a spot color (see »Colorize a grayscale image with a spot color«, page 194).

By default, built-in spot colors cannot be redefined with custom alternate values. This behavior can be changed with the *spotcolorlookup* option of *PDF_set_option( )*. 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.

Note *Built-in Pantone® and HKS® 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® with thousands of named colors. All color swatch names from the digital color libraries listed in Table 4.3 can be used.

Spot color names are case-sensitive; use uppercase as shown in the examples. The *PANTONE* prefix must always be provided in the swatch name as shown in the examples. Pantone color names are 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 or another designation (e.g., *C* for coated). A single space character must be

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

```
p.set_graphics_option("fillcolor={ spotname {PANTONE 281 U} 0.7 }");
```

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., 2003-2016.

Table 4.3 Pantone spot color libraries built into PDFlib

color library name	***sample color name***	***remarks***
PANTONE solid coated	*PANTONE 185 C*	
PANTONE+ Solid Coated-336 New	*PANTONE 2071 C*	*336 colors introduced in 2012*
PANTONE PLUS Solid Coated	*PANTONE 2337 C* *PANTONE 3514 C*	*84 colors in the range 2337-2427 introduced in 2014;* *112 colors in the range 2428-3599 added in 2016*
PANTONE solid uncoated	*PANTONE 185 U*	
PANTONE+ Solid Uncoated-336 New	*PANTONE 2071 U*	*336 colors introduced in 2012*
PANTONE PLUS Solid Uncoated	*PANTONE 2337 U* *PANTONE 3514 U*	*84 colors in the range 2337-2427 introduced in 2014;* *112 colors in the range 2428-3599 added in 2016*
PANTONE solid matte	*PANTONE 185 M*	
PANTONE Extended Gamut Coated (XGC)	*PANTONE 185 XGC*	*1729 XGC colors introduced in 2015*
PANTONE process coated	*PANTONE DS 35-1 C*	
PANTONE process uncoated	*PANTONE DS 35-1 U*	
PANTONE process coated EURO	*PANTONE DE 35-1 C*	
PANTONE process uncoated EURO	*PANTONE DE 35-1 U*	*introduced in 2006*
PANTONE pastel coated	*PANTONE 9461 C*	*includes new colors introduced in 2006*
PANTONE pastel uncoated	*PANTONE 9461 U*	*includes new colors introduced in 2006*
PANTONE metallic coated	*PANTONE 871 C*	*includes new colors introduced in 2006*
PANTONE color bridge CMYK PC	*PANTONE 185 PC*	*replaces PANTONE solid to process coated*
PANTONE color bridge CMYK EURO	*PANTONE 185 EC*	*replaces PANTONE solid to process coated EURO*
PANTONE color bridge uncoated	*PANTONE 185 UP*	*introduced in 2006*
Deprecated color libraries (not recommended)		
PANTONE hexachrome coated	*PANTONE H 305-1 C*	*discontinued in 2008*
PANTONE hexachrome uncoated	*PANTONE H 305-1 U*	*discontinued in 2008*
PANTONE solid in hexachrome coated	*PANTONE 185 HC*	
PANTONE solid to process coated	*PANTONE 185 PC*	*replaced by PANTONE color bridge CMYK PC*
PANTONE solid to process coated EURO	*PANTONE 185 EC*	*replaced by PANTONE color bridge CMYK EURO*

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 digital color libraries listed in Table 4.4 can be used.

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. HKS color names are constructed according to the following scheme:

```
HKS <id> <paperstock>
```

where <*id*> is the identifier of the color (e.g., 43) and <*paperstock*> the abbreviation of the paper stock in use (e.g., *N* for natural paper). A single space character must be provided between the *HKS*, <*id*>, and <*paperstock*> components constituting the swatch name. If a spot color is requested where the name starts with the HKS prefix, but the name does not represent a valid HKS color a warning is logged. The following code snippet demonstrates the use of an HKS color with a tint value of 70 percent:

```
p.set_graphics_option("fillcolor={ spotname {HKS 38 E} 0.7 }");
```

Table 4.4 HKS spot color libraries built into PDFlib

color library name	*sample color name*	*remarks*
HKS K	*HKS 43 K*	*88 colors for gloss art paper (Kunstdruckpapier)*
HKS N	*HKS 43 N*	*86 colors for natural paper (Naturpapier)*
HKS E	*HKS 43 E*	*88 colors for continuous stationary/coated (Endlospapier)*
HKS Z	*HKS 43 Z*	*50 colors for newsprint (Zeitungspapier)*

Custom spot colors. In addition to built-in spot colors as detailed above, PDFlib supports custom spot colors. These can have 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.

Spot colors can be set with the *fillcolor/strokecolor* text or graphics appearance options and other color-related options. The alternate color can be supplied directly in the spot color definition:

```
fillcolor={spotname={CompanyRed} 1.0 {cmyk 0 0.78 0.88 0}}
```

Alternatively, spot colors can be defined by using the current fill color as alternate color. Except for an additional call to set the current fill color as alternate color, defining and using custom spot colors works similarly to using built-in spot colors:

```
/* set current fill color for use as alternate color */
p.setcolor("fill", "cmyk", 0 0.78 0.88 0);
/* derive spot color from the current fill color */
spot = p.makespotcolor("CompanyRed");

/* set spot color for filling */
p.set_graphics_option("fillcolor={ spotname {CompanyRed} 0.7 }");
```

Separation color space based on CMYK process colors. While spot color names generally can be chosen arbitrarily, the colorant names *Cyan, Magenta, Yellow* and *Black* always refer to the CMYK process colors. This fact can be used to paint into a specific CMYK channel only (e.g. Magenta), without affecting the other channels. This differs from painting in full CMYK color with *Cyan=Yellow=Black=0* which erases existing contents in the Cyan, Yellow and Black channels (this behavior can be modified with device-dependent settings, see Section 4.11, »Overprint Control«, page 102). For example, the following call:

```
fillcolor={spotname=Magenta 0.5 {cmyk 0 1 0 0}}
```

is different from directly using CMYK color:

```
fillcolor={cmyk 0 0.5 0 0}
```

since the former call leaves existing contents in the Cyan, Yellow and Black channels unchanged, while the second calls sets them to 0% (subject to the overprint settings).

4.5 DeviceN Colors

Cookbook *Full code samples can be found in the* `starter_color` *sample and the Cookbook topic* `color/devicen_color`.

DeviceN color spaces support an arbitrary number of named color components and can be regarded as a generalization of spot colors. The colorants may be taken from the set of process colors (typically CMYK, but other process color spaces are possible) or may be arbitrary spot colors. Applications of DeviceN color include the following:

- Some printing systems use more than four process colors to extend the color gamut (the set of printable colors). For example, some seven-color printing systems are based on CMYK plus Orange, Green and Violet.
- A DeviceN color space containing a subset of the CMYK process colors, e.g. only Cyan and Magenta, can be used when an object in CMYK color should overprint other CMYK objects.
- DeviceN color can be used to construct a shading (smooth color transition) between a spot color and a process color or between multiple spot colors.
- Package printing often uses n-colorant color because traditional spot colors cannot be used. Additional color channels are also used to record information which is not directly related to visible colors, such as varnish or die lines.

The function *PDF_create_devicen( )* returns a DeviceN color space handle which can be used for drawing operations with color-related options such as *fillcolor/strokecolor*, for constructing a shading with *PDF_shading( )*, or for colorizing a raster image with the *colorize* option of *PDF_load_image( )*. DeviceN colors can be tinted, i.e., they can be used with a percentage for each color channel which specifies the intensity of the respective color channel from 0=no color=white to 1=maximum intensity. DeviceN color spaces are always subtractive (unlike additive color spaces like RGB where 0=black).

PDF_create_devicen( ) needs the list of colorant names, the alternate color space, and PostScript code which implements the tint transform function. The transform function must convert *N* color values of the DeviceN color space to the corresponding color values in the alternate color space. The alternate color values are used for rendering if the output device doesn't support the named colors in the DeviceN color space, e.g. on a monitor. Below we provide suitable PostScript transform functions for DeviceN color spaces constructed from CMYK process colors or from spot colors when using Lab as alternate color space. Devising a suitable PostScript transform function for other combinations is a non-trivial task.

When creating output for PDF 1.6 or above, PDFlib emits a DeviceN attributes dictionary with *Colorants* entries for all known spot colors. This ensures that the PDF viewer has enough information to render the named colorants individually and can blend the colorants. The advantage of this method is that the color display in Acrobat is correct regardless of the Overprint Preview setting. It also means that Acrobat doesn't need the PostScript transform function for rendering the page. However, the transform function must nevertheless be provided since third-party viewers or RIPs may resort to the PostScript function for rendering the DeviceN color to the alternate color space.

In PDF/X-4/5 mode *PDF_makespotcolor( )* must be called before *PDF_create_devicen( )* for all custom spot colors used in the DeviceN color space. Since this generally improves the screen display it is recommended even if no PDF/X-4/5 is generated.

NChannel color spaces. PDF 1.6 introduced an extension of DeviceN color spaces called NChannel. NChannel color spaces contain additional information which aids the PDF viewer or RIP in accurate color rendering. This information includes alternate color spaces for individual colorants in the DeviceN color space (not only the combined tint transform function), the process color space and component names. NChannel may also include additional information describing the mixing behavior of printing inks, but this is not currently supported in PDFlib. NChannel color spaces require the *Colorants* entry with individual alternate colors for the colorants. This is created automatically by PDFlib. NChannel color spaces also require the name of the process color space and its colorant names. These must be supplied in the *process* option and the *colorspace* and *components* suboptions.

NChannel color spaces can be created with the option *subtype=nchannel* in *PDF_create_devicen( )*. The advantage of this setting is that DeviceN color spaces with this subtype no longer depend on Acrobat's *Overprint Preview* setting. In some cases this Acrobat preference is required for correct screen preview, but NChannel automatically forces Acrobat into this display mode.

DeviceN color space based on CMYK process colors. In the following example we use a subset of CMYK colors consisting of the Magenta and Yellow colorants, and DeviceCMYK as alternate color space. The transform function is simple as it only supplies additional zero values for the Cyan and Black components of the CMYK alternate color space:

```
devicen = p.create_devicen(
        "names={Magenta Yellow} alternate=devicecmyk transform={{0 0 4 1 roll}}");
p.set_graphics_option("fillcolor={devicen " + devicen + " 0.5 1}");
```

N color values must be supplied when setting the color *(N=2* in the example above). The DeviceN color space created above paints only in the Magenta and Yellow channels, but leaves the Cyan and Black channels unchanged. This differs from painting in full CMYK color with *Cyan=Black=0* which erases existing contents in the Cyan and Black channels (subject to the overprint settings).

For convenience Table 4.3 lists the DeviceN PostScript transform functions for all possible subsets of CMYK colorants with CMYK as alternate color space.

Table 4.5 DeviceN PostScript transform functions for subsets of CMYK process colorants and `alternate=devicecmyk`

DeviceN colorant names for CMYK subsets	***PostScript transform function***	***DeviceN colorant names for CMYK subsets***	***PostScript transform function***
Cyan	`{0 0 0}`	*Magenta Black*	`{0 3 1 roll 0 exch}`
Magenta	`{0 0 0 4 1 roll}`	*Yellow Black*	`{0 0 4 2 roll}`
Yellow	`{0 0 0 4 2 roll}`	*Cyan Magenta Yellow*	`{0}`
Black	`{0 0 0 4 3 roll}`	*Cyan Magenta Black*	`{0 exch}`
Cyan Magenta	`{0 0}`	*Cyan Yellow Black*	`{0 3 1 roll}`
Cyan Yellow	`{0 exch 0}`	*Magenta Yellow Black*	`{0 4 1 roll}`
Cyan Black	`{0 0 3 -1 roll}`	*Cyan Magenta Yellow Black*	`{ }`
Magenta Yellow	`{0 0 4 1 roll}`		

A similar technique can be used to render only a subset of CMYK image channels. For example, the following code fragment creates a DeviceN color space which uses only the Black channel and replaces the other three channels with *None*. Since four color channels are present in the image data, the PostScript transform function replaces the Cyan, Magenta and Yellow channels with zero values (this differs from the PostScript functions in Table 4.3 which add missing 0 values instead of replacing existing color values with zero):

```
devicen = p.create_devicen(
        "names={None None None Black} " +
        "alternate=devicecmyk transform={{4 1 roll pop pop pop 0 0 0 4 -1 roll}}");
optlist = "width=4000 height=3000 bpc=8 colorize=" + devicen;
image = p.load_image("raw", filename, optlist);
```

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

DeviceN color space based on spot colors with Lab alternate space. Using two or more spot colors as DeviceN colorants provides the basis for shadings between spot colors. Implementing a suitable PostScript transform function for device-dependent alternate color spaces is rather challenging due to the nonlinear characteristics of RGB and CMYK. Blending colors in the Lab color space is much easier because it is perceptually uniform. This means we can combine colors simply by calculating the weighted average of their Lab values. The following PostScript code implements this method for two Pantone colors whose Lab alternate values must be provided at the start of the code:

```
% PostScript transform function for DeviceN with N=2 and Lab alternate;
% blend Lab colors values by using the tint values as
% weights and calculating the weighted average of the L/a/b values

80 28 75                    % Lab values of color 1=PANTONE 123 U
31.7647 0 -17               % Lab values of color 2=PANTONE 289 U

% blend L values
7 index 6 index mul         % t1*L1
7 index 4 index mul         % t2*L2
add 9 1 roll                % bottom: L = t1*L1 + t2*L2

% blend a values
7 index 5 index mul         % t1*a1
7 index 3 index mul         % t2*a2
add 9 1 roll                % bottom: a = t1*a1 + t2*a2

% blend b values
7 index 4 index mul         % t1*b1
7 index 2 index mul         % t2*b2
add 9 1 roll                % bottom: b = t1*b1 + t2*b2

% pop 2 tint and 2x3 input color values
pop pop pop pop pop pop pop pop
% result: Lab values of blended color
```

The PostScript function *transformFunc2* above can be used to create a DeviceN color space based on the two Pantone spot colors as follows:

```
devicen = p.create_devicen(
        "names={{PANTONE 123 U} {PANTONE 289 U}} alternate=lab " +
        "transform={{" + transformFunc2 + "}}");
```

PDFlib automatically constructs such a DeviceN color space when a shading is constructed based on different spot colors with Lab alternate values.

Cookbook *A more extensive code sample and Lab-based PostScript transform functions for higher values of N can be found in the Cookbook topic* `color/devicen_color`.

4.6 Shadings and Shading Patterns

Cookbook *Full code samples can be found in the* `starter_color` *sample and the Cookbook topic* `color/color_gradient`.

Smooth shadings, also called color blends or gradients, provide a continuous transition between two or more colors in the same color space, e.g. two RGB colors or two tints of a spot color. Simple shadings define a transition between two colors. The first color is taken from the *startcolor* option of *PDF_shading( )* or the current fill color, The second color is provided in the *endcolor* option or the *c1, c2, c3,* and *c4* parameters. Shadings with an arbitrary number of intermediate colors can be created by using the *stopcolors* option instead of *startcolor/endcolor*.

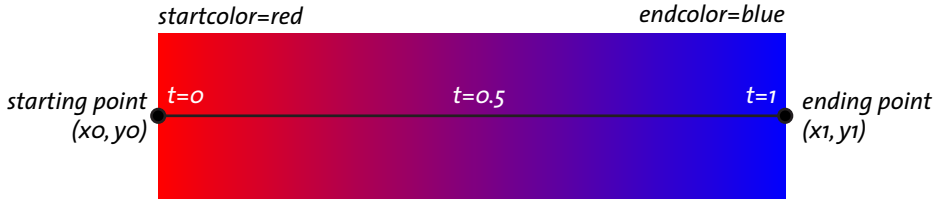
The shading color is controlled by a variable *t* which varies linearly from 0 at the start color to 1 at the end color. The linear transition can be changed to an exponential transition with the *N* option as exponent. Since the RGB and CMYK color spaces are not perceptually linear, the resulting blends may not appear as smooth as desired. Specifying the blend with start and end colors in the Lab color space may result in a smoother color transition. PDFlib supports two different kinds of geometry for shadings:

- The color in axial shadings *(type=axial)* varies along a line between a starting point and an ending point (see Figure 4.2). The shading color varies linearly from the starting point *(x0, y0)* to the ending point *(x1, y1)*. The shading can optionally be extended with the boundary color beyond the starting and/or ending point, subject to the *extend0* and *extend1* options.
- The color in radial shadings *(type=radial)* varies between two circles (see Figure 4.3). Radial shadings may be used to create a 3D-like visualization of a sphere. The shading color varies from a starting circle with center *(x0, y0)* and radius *r0* to an ending circle with center *(x1, y1)* and radius *r1*. One of the circles may collapse to a point.

PDF_shading( ) returns a handle to a shading object which can be used in two ways:

- Fill an area directly with *PDF_shfill( )*. This method is preferable if the geometry of the object to be filled is the same as the geometry of the shading. Contrary to its name this function not only fills the interior of the object, but also affects the exterior. This behavior can be modified with *PDF_clip( )*.
- Define a shading pattern (not to be confused with tiling patterns) 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.

Fig. 4.2
Main parameters of an axial shading between two colors



Shadings between two process colors. The following code creates an axial shading from red to blue in the RGB color space and uses it to fill a circle:

```
sh  = p.shading("axial", 100, 100, 500, 500, 0, 0, 0, 0, "startcolor=red endcolor=blue");
shp = p.shading_pattern(sh, "");
p.set_graphics_option("fillcolor={pattern " + shp + "}");

p.circle(300, 300, 200);
p.fill();
```

Shadings between spot colors. Since shadings can only be used to create transitions between colors in the same color space, you cannot use arbitrary spot colors or a spot color and a process color as start and end color. However, for spot colors with Lab alternate values shadings can be achieved with a suitable DeviceN color space similar to the one shown in the section »DeviceN color space based on spot colors with Lab alternate space«, page 90. PDFlib automatically creates such a DeviceN color space for spot colors as stop colors, provided certain conditions are met (see PDFlib API Reference). The following code fragment creates a shading between two spot colors:

```
sh  = p.shading("axial", 100, 100, 500, 500, 0, 0, 0, 0,
        "stopcolors={ 0% {spotname {PANTONE 123 U} 1} 100% {spotname {PANTONE 289 U} 1}}"
shp = p.shading_pattern(sh, "");
p.set_graphics_option("fillcolor={pattern " + shp + "}");

p.circle(300, 300, 200);
p.fill();
```

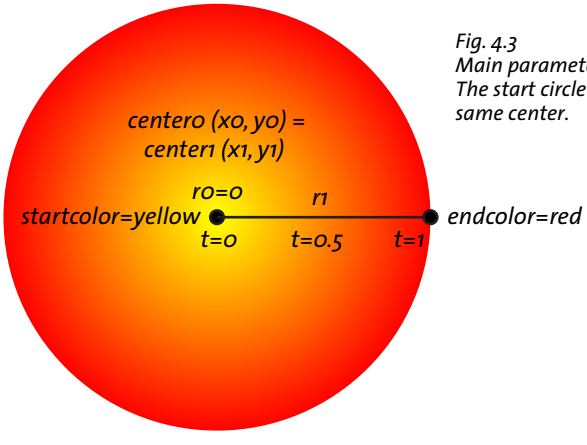


Fig. 4.3
Main parameters of a radial shading between two colors. The start circle collapses to a point; both circles have the same center.

4.7 Tiling Patterns

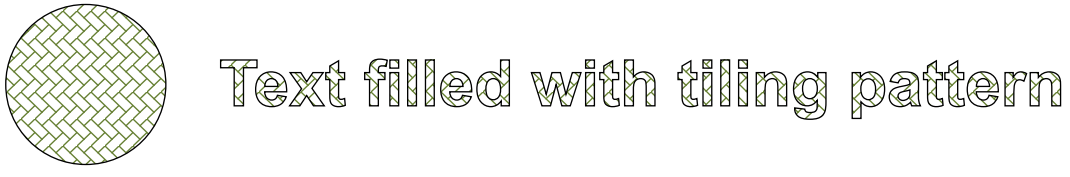
Cookbook *Full code samples can be found in the* `starter_color` *sample and the Cookbook topics* `graphics/fill_pattern` *and* `images/tiling_pattern`.

A tiling 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 with drawing operators between *PDF_begin_pattern_ext( )* and *PDF_end_pattern( )*. Most graphics operators can be used to define a pattern.
- The pattern handle returned by *PDF_begin_pattern_ext( )* can be used to set the pattern as the current color with the options *fillcolor/strokecolor* in *PDF_set_graphics_option( )* or *PDF_setcolor( )*.

Depending on the *painttype* option of *PDF_begin_pattern_ext( )* the pattern definition may or may not include its own color specifications. If *painttype=colored*, the pattern definition must contain its own color specification and will always look the same; if *painttype=uncolored*, 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.

Fig. 4.4
A tiling pattern used for filling a circle and some text. The text outlines are additionally stroked for emphasis.



4.8 Transparency Blend Modes

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

The blend mode controls how the colors of an object blend with the background in the transparent imaging model (as opposed to the opaque imaging model where each object completely obscures all underlying objects). Since the background color may have been created by blending multiple other objects, it is referred to as *backdrop*. The object being painted is referred to as *source*. Source and backdrop colors are blended against each other with a blend function. The blend modes listed in Table 4.6 create a variety of artistic effects and can also be leveraged for specialized applications (see Section 4.9.1, »Changing the Color with Blend Modes«, page 98). The available blend modes can be categorized as follows:

- The default blend mode *Normal* makes the source completely opaque, i.e. the source object completely replaces the background. Setting a blend mode other than *Normal* implicitly makes the source object transparent, regardless of any opacity settings or image transparency.
- The blend modes *Darken*, *Multiply* and *ColorBurn* create darkening effects.
- The blend modes *Lighten, Screen* and *ColorDodge* create lightening effects.
- The blend modes *Overlay*, *SoftLight* and *HardLight* increase the contrast by modifying light and dark areas.
- The differencing blend modes *Difference* and *Exclusion* subtract color values. They are the only modes where painting a white object on a white backdrop does not result in white.
- The blend modes *Hue, Saturation, Color* and *Luminosity* are defined by their effect on the dimensions of the HSL color representation. In the HSL model hue describes the perceived color (red and green are different hues), saturation describes how colorful a color appears (how much gray is mixed into the color), and luminosity describes how dark or light a color appears (white has maximum luminosity).

Table 4.6 Blend modes and their effect when blending a source object against background objects (backdrop)

Blend mode	Description according to ISO 32000-2
Blend modes without any blending effect	
None	*PDFlib-specific blend mode for creating a gstate without any blend mode specification. This ensures that a blend mode set in an enclosing gstate comes into effect.*
Normal	*Selects the source color, ignoring the backdrop.*
Darkening blend modes	
Darken[1]	*Selects the darker of the backdrop and source colors. The backdrop is replaced with the source where the source is darker; otherwise, it is left unchanged.*
Multiply[1]	*Multiplies the backdrop and source color values. The result color is always at least as dark as either of the two constituent colors. When working with additive colors, multiplying any color with black produces black while multiplying with white leaves the original color unchanged. For subtractive colors, the maximum tint value used for all colorants of the color space acts as black does for additive spaces. Painting successive overlapping objects with a color other than black or white produces progressively darker colors.*
ColorBurn	*Darkens the backdrop color to reflect the source color. Painting with white produces no change.*

Table 4.6 Blend modes and their effect when blending a source object against background objects (backdrop)

Blend mode	Description according to ISO 32000-2
Lightening blend modes	
Lighten[1]	*Selects the lighter of the backdrop and source colors. The backdrop is replaced with the source where the source is lighter; otherwise, it is left unchanged.*
Screen[1]	*Multiplies the complements of the backdrop and source color values, then complements the result. The result color is always at least as light as either of the two constituent colors. When working with additive colors, screening any color with white produces white while screening with black leaves the original color unchanged. For subtractive colors, the maximum tint value of all colorants of the color space acts as black does for additive spaces. The effect is similar to projecting multiple photographic slides simultaneously onto a single screen.*
ColorDodge	*Brightens the backdrop color to reflect the source color. Painting with black produces no change.*
Contrasting blend modes	
HardLight	*Multiplies or screens the colors if the source color value is smaller or larger than 0.5, respectively. The effect is similar to shining a harsh spotlight on the backdrop.*
SoftLight	*Darkens or lightens the colors if the source color value is smaller or larger than 0.5, respectively. The effect is similar to shining a diffused spotlight on the backdrop.*
Overlay	*Multiplies or screens the colors, depending on the backdrop color value. Source colors overlay the backdrop while preserving its highlights and shadows. The backdrop color is not replaced but is mixed with the source color to reflect the lightness or darkness of the backdrop.*
Differencing blend modes	
Difference[1,2]	*Subtracts the darker of the two constituent colors from the lighter color: Painting with white inverts the backdrop color; painting with black produces no change. For subtractive colors, the maximum tint value for all colorants of the color space acts as black does for additive spaces.*
Exclusion[1,2]	*Produces an effect similar to that of the* `Difference` *mode but lower in contrast. Painting with white inverts the backdrop color; painting with black produces no change. For subtractive colors, the maximum tint value for all colorants of the color space acts as black does for additive spaces.*
HSL blend modes	
Hue[2]	*Creates a color with the hue of the source color and the saturation and luminosity of the backdrop color.*
Saturation[2]	*Creates a color with the saturation of the source color and the hue and luminosity of the backdrop color. Painting with this mode in an area of the backdrop that is a pure gray (no saturation) produces no change.*
Color[2]	*Creates a color with the hue and saturation of the source color and the luminosity of the backdrop color. This preserves the gray levels of the backdrop and is useful for coloring monochrome images or tinting color images.*
Luminosity[2]	*Creates a color with the luminosity of the source color and the hue and saturation of the backdrop color. This produces the same effect as the* `Color` *mode, but with source and backdrop exchanged.*

1. This mode is symmetric, i.e. swapping the source and backdrop colors doesn't change the result of the blending operation.
2. This mode doesn't have any effect on spot colors; blend mode `Normal` *is used instead.*

Blending color space. The PDF viewer performs all transparency calculations in a blending color space. This color space plays an important role since the blending results depend on the selected color space. It is determined as follows:

- The color space specified with the *colorspace* suboption of the *transparencygroup* page option if present.

- Otherwise the PDF/X or PDF/A output intent ICC profile is used if present. In PDF/X an output intent is always present. If a PDF/A document doesn't contain any output intent it must specify the *colorspace* suboption of the *transparencygroup* page option.
- Otherwise the native color space of the output device (or the simulated output device for soft proofing/output preview) is used. It is recommended to avoid this situation by specifying the blending color space explicitly since otherwise the rendering of transparent objects depends on the output device.

The following code fragment specifies DeviceRGB as blending color space and blend mode *Multiply*:

```
p.begin_page_ext(0, 0, "width=a4.width height=a4.height " +
                       "transparencygroup={colorspace=DeviceRGB}");

gstate = p.create_gstate("blendmode=Multiply");
p.set_gstate(gstate);
```

Multiple blend modes. In PDF only one blend mode at a time can be active in the graphics state. If you want to apply the effects of multiple blend modes you can paint another object on top of the already blended backdrop and object. For example, in order to invert the result of some other blend mode, first blend the backdrop and object with some blend mode; then change the blend mode to *Difference* and paint a white rectangle on top of the result of the first blend operation.

4.9 Changing the Color of Objects

PDFlib supports several methods for changing the color of objects such as imported PDF pages, raster images or SVG graphics, or arbitrary text and vector elements placed on the page:

- Raster images can be colorized, see Section 8.1.5, »Colorize Images with Spot or DeviceN Color«, page 194.
- The color of arbitrary objects can be changed with blend modes.
- GStates with luminosity soft masks can be used to apply color to arbitrary objects.

4.9.1 Changing the Color with Blend Modes

Cookbook *A code sample for all effects discussed in this section can be found in the Cookbook topic* `color/blendmode_effects`.

In this section we present some useful applications of blend modes. A description of all available blend modes can be found in Section 4.8, »Transparency Blend Modes«, page 95. In all examples the colors of an object, e.g. an imported image, PDF page, or SVG graphics, are modified in some way.

All examples can be implemented with the basic code fragment shown below. It specifies a transparency group colorspace for the page to avoid device-specific rendering and sets a specific blend mode. In order to avoid subsequent objects to be affected by the specified blend mode the sequence should be bracketed with a *save/restore* pair:

```
p.begin_page_ext(width, height, "transparencygroup={colorspace=DeviceCMYK}");

// place raster image, imported PDF page or SVG graphics
p.fit_image(image, 200, 150, optlist);

// Colorize, decolorize or invert existing content by blending with the desired color
p.save();
        // Create gstate with the desired blend mode
        gs_blendmode = p.create_gstate("blendmode=Color");
        p.set_graphics_option("fillcolor=red gstate=" + gs_blendmode);
        p.rect(0, 0, width, height);
        p.fill();
p.restore();
```

Colorize an object with arbitrary color. Blend mode *Color* can be used to colorize an object with some new color, i.e. the perceived gray levels of the object control the tint value of an additional color while the object's original colors are ignored. In order to colorize an object apply blend mode *Color* and place a colored rectangle (or other shape) on top of the object. In order to colorize transparent objects you it is recommend to place a white area below the placed object.

If you want the backdrop color to show through the white areas of the source object instead (while dark areas remain dark), use blend mode *Multiply*.

Decolorize or invert an object. In some situations you may want to decolorize (or desaturate, or bleach) an object, i.e. create grayscale output for an object which originally contained color, where the gray levels correspond to the perceived brightness (luminosity) of the object's original colors. This can be achieved with blend mode *Color* and a white area on top of the object.

The colors of raster images can be inverted with the *invert* image option. The colors of arbitrary objects can be inverted with blend mode *Difference* and white on top of the object.

4.9.2 Changing the Color with Soft Masks

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

Image alpha channels can be generalized to arbitrary objects and color spaces with the help of a luminosity soft mask which is contained in a graphics state (gstate). The soft mask is constructed on the basis of a template which may contain arbitrary contents such as imported PDF pages or SVG graphics. The luminosity (perceived gray level) of the template contents determines the visibility of subsequently drawn objects: light areas of the template are transparent (i.e. drawn objects are visible), while dark areas prevent objects from being visible. This technique entails the following steps:

- Create a template with the *transparencygroup* option and suboption *colorspace.*Apply arbitrary drawing operations in the template.
- When applying the soft mask the light (white) areas of the soft mask template will be colorized with the current color while dark (black) areas will not get any color. As a result dark and light areas of the soft mask are exchanged. In order to preserve dark and light areas you can invert the template colors with blend mode *Difference* as explained in the section »Decolorize or invert an object«, page 98.
- Create a gstate based on the template with the *softmask* option list and the suboption *type=luminosity*. By default the soft mask background is initialized to black, i.e. opaque. This can be changed by setting the *backdropcolor* option to white (with a suitable number of color components according to the *colorspace* suboption of the template's *transparencygroup* option.
- Set the gstate, usually bracketed with a save/restore sequence to limit the effect of the soft mask. The geometry of the soft mask template in the gstate is subject to the current coordinate system. For example, you can control the position of the soft mask by applying a translation before setting the gstate. Now draw the contents which will be masked on the page.

The following code fragment implements this sequence including mask inversion and applies it to colorize an imported SVG graphics file:

```
// Create the template which will be used to define the soft mask
tpl = p.begin_template_ext(595, 842, "transparencygroup={colorspace=devicecmyk}");

    // Place arbitrary contents, e.g. SVG graphics
    p.fit_graphics(graphics, 0, 0, "boxsize={595 842} position={center} fitmethod=meet");

    // Invert the template luminosity to preserve dark and light areas
    // Create a gstate with blend mode for color inversion
    gstate_invert = p.create_gstate("blendmode=Difference");
    p.set_graphics_option("fillcolor=white gstate=" + gstate_invert);
    p.rect(0, 0, 595, 842);
    p.fill();
p.end_template_ext(0, 0);


// Activate the gstate with soft mask and apply color
// Result: SVG graphics is colorized with red
p.save();
```

```
    // Create a gstate with soft mask based on the template
    // The "backdropcolor" option initializes the template area to transparent
    gstate_softmask = p.create_gstate(
        "softmask={type=luminosity template=" + tpl + " backdropcolor={0 0 0 0} }");
    p.set_graphics_option("fillcolor=red gstate=" + gstate_softmask);
    p.rect(0, 0, 595, 842);
    p.fill();
p.restore();
```

Multiple soft masks. PDF supports only up to one soft mask per compositing operation. For example, if an image contains its own alpha channel (=soft mask), a soft mask supplied in a graphics state doesn't have any effect. As a workaround to this PDF restriction you must create the contents in a template and place this template on the page, effectively forcing the PDF viewer to apply a separate compositing operation where the soft mask can be used.

4.10 Rendering Intents

Although PDFlib applications can specify device-independent color values, a particular output device may not be able to accurately reproduce the colors. In this situation some compromises have to be made in a process called gamut compression, i.e., reducing the range of colors to a smaller range which can be reproduced by the device. The rendering intent can be used to control this process. Rendering intents can be specified for individual images by supplying the *renderingintent* option to *PDF_load_image()*. In addition, rendering intents can be specified for text and vector graphics by supplying the *renderingintent* option to *PDF_create_gstate()*. Table 4.6 lists all available rendering intents.

Table 4.7 Available rendering intents

intent	*description*	*typical use*
Auto	*Do not specify any rendering intent in the PDF file, but use the device's default intent instead.*	*not enough information about the page contents is available*
AbsoluteColorimetric	*No correction is made for the device's white point (such as paper white). Colors which are out of gamut are mapped to nearest value within the device's gamut.*	*exact reproduction of solid colors; not recommended for other uses*
RelativeColorimetric	*The color data is compressed into the device's gamut, mapping the white points onto one another while slightly shifting the remaining colors.*	*vector graphics*
Saturation	*Color saturation is preserved while the color values may be shifted.*	*business graphics*
PerceptualColor	*Color relationships are preserved by modifying both in-gamut and out-of-gamut colors in order to provide a pleasing appearance.*	*real-world images*

4.11 Overprint Control

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

By default, painting in a specific color channel replaces the corresponding areas of the other color channels. For example, color 0/0/1/0, i.e. Yellow, on a CMYK device erases existing Cyan, Magenta and Black channels of objects which have been painted earlier at the same location of the page. This behavior can be changed with a process called overprinting which relates to mixing of color ink particles on a printing device. Overprinting can be used to create »enriched Black« by printing Black over another dark color.

The overprint flag in PDF is targeted at CMYK printing. It controls whether painting in a particular channel has an effect on existing contents in other channels. For example, if Cyan is printed on top of Yellow, the result will be green with overprinting (Cyan and Yellow are mixed). On the other hand, the same sequence without overprinting results in the last printed color Cyan since pure Cyan has no Yellow component, therefore 0% Yellow replaces the existing Yellow contents.

Overprint settings for fill and stroke operations. The overprinting behavior can be controlled separately for stroke and other operations with the graphics appearance options *overprintstroke* (for stroking operations) and *overprintfill* (for all non-stroking operations including image placement). Both options default to *false:*

- If *overprintfill=false* or *overprintstroke=false*, filling or stroking in any color channel replaces the corresponding areas of unspecified colorants: the foreground color wins.
- If the options are *true* and the output device supports overprinting, previous markings in unspecified colorants are left unchanged. Overprinting is usually used for very dark colors.

Overprint mode for CMYK colors. The overprint behavior of CMYK objects can further be modified with the *overprintmode* graphics appearance option which controls the behavior of zero CMYK components if *overprintfill/overprintstroke=true:*

- *overprintmode=0* (default): each color component replaces previously placed marks (»foreground color wins«). In other words, all channels of the foreground object color knock out the underlying objects.
- *overprintmode=1* (sometimes called »Illustrator overprint mode«): tint value 0 leaves the corresponding component of previously painted color unchanged instead of setting it to 0 (»foreground tint value 0 is ignored«). In other words, only foreground channels with non-zero values knock out underlying objects. This is equivalent to painting in a DeviceN color space which includes only those CMYK components which have non-zero values.

This means that a small difference in CMYK tint values (e.g. 0% vs. 1%) can result in a big difference depending on the overprint mode: with overprint mode 0 the result is 0% or 1%, respectively. With overprint mode 1, the result is the background color (whatever that was) or 1%, respectively. Due to its definition overprint mode doesn't have any effect on color spaces other than CMYK.

Only colors in the DeviceCMYK color space are affected by overprinting. If the Cyan, Magenta, Yellow or Black components are used in a DeviceN color space the CMYK overprinting rules don't apply.

Device-dependent overprint rendering. Since PDF viewers must ignore overprint settings if the output device doesn't support overprinting, documents with overprinting objects are inherently non-portable and may require specific configuration of the PDF viewer or RIP to achieve the intended page representation. Whether or not Acrobat honors overprint settings for screen display can be configured via *Edit, Preferences, Page Display, Use Overprint Preview*. We recommend to set this preference to *Always* when working with overprint settings to ensure reliable screen display. In addition, the *Output Preview* panel in Acrobat also offers an item *Simulate Overprinting*.

If transparency or blend modes are used on the page, Acrobat's overprint preview works as expected only if the current blending color space is set to use the DeviceCMYK color space, e.g. with the following option in *PDF_begin_page_ext( ):*

```
transparencygroup={colorspace=devicecmyk}
```

Third-party PDF viewers without support for overprint preview may display unexpected colors. In fact, most simple PDF viewers don't support overprint preview.

Device-independent overprinting effect. Because of the viewer dependencies described above the rendering of overprinting may be fragile. You can achieve overprinting effects in a device-independent manner by using blend mode *Darken* as follows:

```
gstate = p.create_gstate("blendmode=darken");
p.set_gstate(gstate);
```

A particular situation where overprint simulation is useful is the recombination of color separations. Assuming you have the Cyan, Magenta, Yellow and Black separations of an image or page and need to recombine these into a composite PDF page, this can be achieved with the *Darken* blend mode.

Cookbook *A code sample for recombining CMYK separations can be found in the Cookbook topic* `color/recombine_color_channels`.

5 Unicode and Legacy Encodings

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

5.1 Important Unicode Concepts

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

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

There is no one-to-one relationship between characters and glyphs. For example, a ligature is a single glyph which represents two or more separate characters. On the other hand, a specific glyph may represent different characters depending on the context (some characters look identical, see Figure 5.1).

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

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

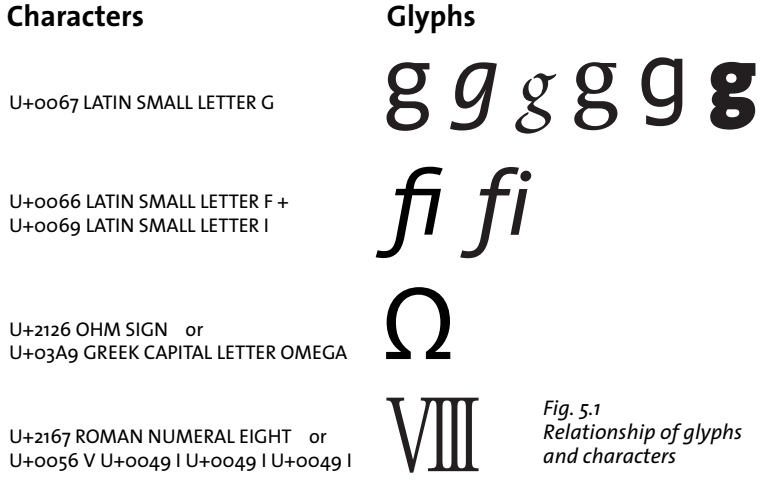


Fig. 5.1
Relationship of glyphs and characters

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

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

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 5.1 lists the representation of the BOM for various encoding forms.

Table 5.1 Byte order marks for various Unicode encoding forms

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

1. The square ■ denotes a null byte.

5.2 Unicode-capable Language Bindings

Some aspects of the PDFlib API vary depending on whether or not the used language binding is Unicode-capable. This concept is discussed in this section and the next section.

5.2.1 Language Bindings with native Unicode Strings

If a programming language or environment supports Unicode strings natively we call the binding Unicode-capable. The following PDFlib language bindings are Unicode-capable:

- C++
- .NET and .NET Core
- Java
- Objective-C
- Python
- RPG

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

- All strings supplied by the client arrive in PDFlib in Unicode encoding and UTF-16 format.
- The distinction between different string types (content strings, hypertext strings and name strings) in the API descriptions is not relevant. The options *textformat*, *hypertextformat* and *hypertextencoding* are not required and are not allowed. The Textflow option *fixedtextformat* is forced to *true*.
- Using *unicode* encoding for the contents of a page is the easiest way to deal with encodings in Unicode-capable languages, but 8-bit encodings and single-byte text for symbol fonts can also be used if so desired.
- Non-Unicode legacy CMaps for Chinese, Japanese, and Korean text (see Section 5.5, »Chinese, Japanese, and Korean CMaps«, page 116) cannot be used since the wrapper always supplies Unicode to the PDFlib kernel; only Unicode CMaps can be used.

The overall effect is that clients can provide native Unicode strings to PDFlib API functions without any additional configuration.

5.2.2 Language Bindings with UTF-8 Support

Programming languages without a native Unicode string data type can nevertheless deal with Unicode strings in UTF-8 format. The following PDFlib language bindings can be made Unicode-capable by setting the *stringformat=utf8* option:

- C
- Perl
- PHP
- Ruby

UTF-8 is recommended if you work with one of these language bindings. The following function call can be used immediately after creating a new PDFlib object to make the language binding Unicode-aware:

```
p.set_option("stringformat=utf8");
```

If Unicode processing is required in the application it is recommended to use the call above to make the language binding Unicode-aware based on UTF-8. After this call the language binding behaves like a Unicode-capable binding except that the client must make sure to supply UTF-8 strings to all API functions. The call has the following additional consequences:

- All strings at the API, i.e. name strings, content strings, hypertext strings and option lists are expected in UTF-8 format with or without BOM.
- In the C language binding name strings as function parameters are still interpreted as UTF-16 if the *length* parameter is supplied with a value larger than 0.

Unicode conversion. If you must deal with strings in other encodings than Unicode, you must convert them to Unicode in UTF-8 or UTF-16 format before passing them to PDFlib. This can be achieved with *PDF_convert_to_unicode()* or language-specific methods. Chapter 2, »PDFlib Language Bindings«, page 29, provides more details regarding useful Unicode string conversion methods provided by common language environments.

5.3 Non-Unicode-capable Language Bindings

The following PDFlib language bindings are not Unicode-capable by default:

- C
- Perl
- PHP
- Ruby

It is recommended to make these language bindings Unicode-capable with the *stringformat* option; see »Language Bindings with UTF-8 Support«, page 107). The remainder of this section is only relevant for applications which are written in one of the languages listed above and which don't set the option *stringformat=utf8*.

Unicode conversion. PDFlib offers the *PDF_convert_to_unicode( )* function which converts between UTF-8, UTF-16, and UTF-32 strings, or from arbitrary encodings to Unicode with an optional BOM.

The format UTF-8 with BOM has the advantage for C users that PDFlib automatically recognizes such strings via the BOM. This makes it possible to load fonts with *encoding=unicode*, treat hypertext strings with *hypertextencoding=unicode* and name strings with *usehypertextencoding*=true, resulting in a full Unicode workflow.

The language-specific sections in Chapter 2, »PDFlib Language Bindings«, page 29, provide more details regarding useful Unicode string conversion methods provided by common language environments.

Unicode handling and string types. Unicode strings can still be used in non-Unicode-capable languages, but string handling is a bit more complicated and depends on the type of string. The PDFlib API uses the string types content string, hypertext string, and name string (these designations are historical misnomers). Parameters and options are marked as one of these types in the PDFlib API Reference. Handling of these string types is summarized in Table 5.2 and detailed in separate sections below.

Table 5.2 Summary of string handling for different string types

string type	*sample parameters and options*	*relevant options/interpretation*
content string	*The* `text` *parameter of PDF_fit_textline( ) and PDF_add_textflow( ).*	`textformat` `encoding`
hypertext string	▸ *The* `fieldname` *option of PDF_add_table_cell( )* ▸ *The* `name` *option of PDF_define_layer( )* ▸ *The* `destname` *option of PDF_create_action( )* ▸ *The* `text` *parameter of PDF_create_bookmark( )*	`hypertextformat` `hypertextencoding`
name string	▸ *The* `filename` *parameter of PDF_begin_document( ) and PDF_create_pvf( )* ▸ *The* `fontname` *parameter of PDF_load_font( )* ▸ *The* `profilename` *parameter of PDF_load_iccprofile( )*	`usehypertextencoding`, `filenamehandling`
strings in option lists		*with BOM: UTF-8* *without BOM: depends on string type*

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

Interpretation of content strings is controlled by the *textformat* option (detailed below) and the *encoding* parameter or option 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 the C language the length of UTF-16 strings must be supplied in a separate *length* parameter.

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

Interpretation of hypertext strings is controlled by the *hypertextformat* and *hypertextencoding* options (detailed below). If *hypertextformat=auto* (which is the default) *utf16* format will be used if *hypertextencoding=unicode*, and *bytes* otherwise. In the C language the length of UTF-16 strings must be supplied in a separate *length* parameter.

Name strings. Name strings are used for external file names, font names, Block names, etc., and are marked as *Name string* in the PDFlib API Reference. They slightly differ from Hypertext strings.

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

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

In C the *length* parameter must be 0 for UTF-8 strings. If it is different from 0 the string is interpreted as UTF-16. In all other non-Unicode-capable 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 you must convert the string to UTF-8.

Text format for content strings and hypertext strings. Unicode strings can be supplied in UTF-8, UTF-16, or UTF-32 format with any byte ordering. The choice of format can be controlled with the *textformat* option for all text on page descriptions and the *hypertextformat* option for interactive elements. Table 5.3 lists the values which are supported for both of these options. The default for the *[hyper]textformat* option is *auto*. Use the *usehypertextencoding* option to enforce the same behavior for name strings. The default for the *hypertextencoding* option is *auto*.

Although the *textformat* setting is in effect for all encodings, it is most useful for *unicode* encoding. Table 5.4 details the interpretation of text strings for various combi-

Table 5.3 Values for the `textformat` *and* `hypertextformat` *options*

[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 IBM System i and IBM Z).*
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 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` *option.* *This setting will provide proper text interpretation in most environments which do not use Unicode natively.*

nations of encodings and *textformat*. If a code or Unicode value in a content string cannot be represented with a suitable glyph in the selected font, the option *glyphcheck* controls the behavior of PDFlib (see »Glyph replacement«, page 131).

Option lists. Strings within option lists require special attention since in non-Unicode-capable 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 is used to determine the format of the string, and the string type (content string, hypertext string, or name string as defined above) is used to determine the appropriate encoding. More precisely, interpreting a string option works as follows:

- If the option starts with a UTF-8 BOM *(oxEF oxBB oxBF)* it is interpreted as UTF-8. On EBCDIC-based systems: if the option starts with an EBCDIC UTF-8 BOM *(ox57 ox8B oxAB)* it is interpreted as EBCDIC UTF-8.
- If no BOM is found, string interpretation depends on the type of string:
  - Content strings are interpreted according to the applicable *encoding* option or the encoding of the corresponding font (whichever is present).
  - Hypertext strings are interpreted according to the *hypertextencoding* option.
  - Name strings are interpreted according to the *hypertext* settings if *usehypertextencoding=true*, and *auto* 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

Table 5.4 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 113*	
unicode *and UTF16 CMaps*	*8-bit codes are Unicode values from U+0000 to U+00FF*	*any Unicode value, encoded according to the chosen text format*[1]
any other CMap (not Unicode-based)	*any single- or multibyte codes according to the selected CMap*	*PDFlib will throw an exception*
Only content strings:		
8-bit and builtin	*8-bit codes*	*Convert Unicode values to 8-bit codes according to the chosen encoding*[1]*.*
glyphid	*8-bit codes are glyph ids from 0 to 255*	*Unicode values will be interpreted as glyph ids*[2]*. Surrogate pairs will not be interpreted.*

1. *If the Unicode character is not available in the font, PDFlib throws an exception or replaces it subject to the* glyphcheck *option.*
2. *If the glyph id is not available in the font, PDFlib throws an exception or replaces it with glyph id 0 subject to the* glyphcheck *option.*

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).

5.4 Single-Byte (8-Bit) Encodings

Note The information in this section is unlikely to be required in Unicode workflows.

8-bit encodings (also called single-byte encodings) map a byte value 0x01-0xFF to a single character with a Unicode value in the BMP (i.e. U+0000...U+FFFF). They are limited to 255 different characters at a time since code 0 (zero) is reserved for the *.notdef* character U+0000. PDFlib contains internal definitions of the following encodings:

```
winansi (identical to cp1252; superset of iso8859-1),
macroman (the original Macintosh character set),
macroman_apple (similar to macroman, but replaces currency with Euro),
ebcdic (EBCDIC code page 1047), ebcdic_37 (EBCDIC code page 037),
pdfdoc (PDFDocEncoding),
iso8859-1, iso8859-2, iso8859-3, iso8859-4, iso8859-5, iso8859-6, iso8859-7, iso8859-8,
iso8859-9, iso8859-10, iso8859-13, iso8859-14, iso8859-15, iso8859-16,s
cp1250, cp1251, cp1252, cp1253, cp1254, cp1255, cp1256, cp1257, cp1258
```

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 IBM System Z with MVS or USS it will be mapped to *ebcdic;*
- on IBM System i it will be mapped to *ebcdic_37;*
- on Windows it will be mapped to *winansi;*
- on all other systems it will be mapped to *iso8859-1;*

Host encoding is primarily useful for writing platform-independent test programs and other simple applications. Host encoding is not recommended for production use, but should be replaced by whatever encoding is appropriate.

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

- On Windows: the current system code page (see below for details)
- On Unix and macOS: *iso8859-1*
- On IBM System i: the current job's encoding *(IBMCCSIDoooooooooooo)*
- On IBM System Z: *ebcdic* (=code page 1047).

For symbol fonts the keyword *auto* is mapped to *builtin* encoding (see Section 6.4.2, »Selecting an Encoding for symbolic Fonts«, page 135). While automatic encoding is convenient in many circumstances, using this method makes your PDFlib client programs inherently non-portable.

Encoding *auto* is used as the default encoding for Name strings (see Section 5.3, »Non-Unicode-capable Language Bindings«, page 109) in non-Unicode-capable language bindings, since this is the most appropriate encoding for file names etc.

Tapping system code pages. PDFlib can fetch code page definitions from the system. 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 7.5.1, »Using TrueType and OpenType CJK Fonts«, page 177, 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*, see »Code pages for custom CJK fonts«, page 117).
- On Linux all codeset identifiers supported the *iconv* facility can be used.
- On IBM IBM System i 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 System Z 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 as PDFlib resource (see Section 3.1.4, »Resource Configuration and File Search«, page 55).
- Provide a font that supports all characters used in the encoding.

The encoding file lists glyph names and codes 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 searches for a suitable Unicode value in its internal tables. A Unicode value can be specified instead of a glyph name:

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

```
0x0021          33
...
```

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 *ox* 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 *oxoooo* or the character name *.notdef* can be provided for unused slots.
- All Unicode values in an encoding or codepage file must be smaller than U+FFFF.

5.5 Chinese, Japanese, and Korean CMaps

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, PDF supports the concept of character collections and character maps *(CMaps)* for organizing the characters in a font.

Note CMaps are mainly used for legacy CJK encodings; they are not necessary in Unicode-based workflows. The function PDF_convert_to_unicode( ) or language- or system-specific methods can be used to convert strings from legacy CJK encodings to Unicode.

Note Unicode-capable language bindings support only Unicode CMaps (UTF16). Other CMaps cannot be used.

Predefined CMaps for common CJK encodings. The predefined CJK CMaps are listed in Table 5.5. They support most CJK encodings used on macOS, 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 CMaps are also available for all locales.

Table 5.5 Predefined CMaps for Japanese, Chinese, and Korean text

locale	CMap name
Simplified Chinese	*UniGB-UCS2-H/V[1], UniGB-UTF16-H/V, GB-EUC-H/V, GBpc-EUC-H/V, GBK-EUC-H/V, GBKp-EUC-H/V, GBK2K-H/V*
Traditional Chinese	*UniCNS-UCS2-H/V[1], UniCNS-UTF16-H/V, B5pc-H/V, HKscs-B5-H/V, ETen-B5-H/V, ETenms-B5-H/V, CNS-EUC-H/V*
Japanese	*UniJIS-UCS2-H/V[1], UniJIS-UCS2-HW-H/V, UniJIS-UTF16-H/V, 83pv-RKSJ-H, 90ms-RKSJ-H/V, 90msp-RKSJ-H/V, 90pv-RKSJ-H, Add-RKSJ-H/V, EUC-H/V, Ext-RKSJ-H/V, H/V*
Korean	*UniKS-UCS2-H/V[1], UniKS-UTF16-H/V, KSC-EUC-H/V, KSCms-UHC-H/V, KSCms-UHC-HW-H/V, KSCpc-EUC-H/V*

1. UCS2 CMaps are deprecated. Unless PDF 1.4 must be created the corresponding UTF16 CMaps should be used.

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. 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 in any convenient directory, and must manually configure the CMap files by setting the *SearchPath* at runtime:

```
p.set_option("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.

Code pages for custom CJK fonts. PDFlib supports the code pages listed in Table 5.6. PDFlib on Windows additionally supports any CJK code page installed on the system.

Table 5.6 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*

5.6 Addressing Characters

Some environments require the programmer to write source code in 8-bit encodings (such as *winansi* or *ebcdic)*. This makes it cumbersome to include isolated Unicode characters in 8-bit encoded text. In order to aid developers in this situation, PDFlib supports several auxiliary methods for expressing text.

5.6.1 Escape Sequences

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

An escape sequence is an instruction to replace a sequence with a single byte value. The sequence starts with the code for the backslash character '\' in the current encoding of the string. Since the backslash has special meaning in many programming languages it may have to be duplicated if used literally in source code. The byte values resulting from substituting escape sequences are listed in Table 5.7; some differ between ASCII and EBCDIC platforms. Only byte values in the range 0-255 can be expressed with escape sequences.

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

Table 5.7 Escape sequences for byte values

sequence	*length*	*macOS, Windows, Unix*	*EBCDIC platforms*	*common interpretation*
\f	2	0C	0C	*form feed*
\n	2	0A	15/25	*line feed*
\r	2	0D	0D	*carriage return*
\t	2	09	05	*horizontal tabulation*
\v	2	0B	0B	*line tabulation*
\\	2	5C	E0	*backslash*
\xNN	4	*two hexadecimal digits specifying a byte value, e.g.* \xFF		
\NNN	4	*three octal digits specifying a byte value, e.g.* \377		

Escape sequences are not substituted by default. You must explicitly set the *escapesequence* option to *true* to use escape sequences in strings. This can be done selectively in the functions where it is required, e.g. *PDF_fit_textline( )*. Alternatively you can activate escape sequence expansion for all content strings (i.e. all text output operations) as follows:

```
p.set_text_option("escapesequence=true");
```

Escape sequence substitution can also be enabled globally with *PDF_set_option( )*. This global option affects all subsequently used name strings, hypertext strings and content strings. Since the values of environment variables are treated as name strings they are also subject to escape sequence expansion. This may result in undesired behavior e.g. with Windows path or file names containing backslash characters. For this reason it is not recommended to activate escape sequence substitution globally with *PDF_set_option( )*, but only selectively with the function where it is required (e.g. *PDF_fit_textline( )*) or only for content strings with *PDF_set_text_option( )*,

As an alternative to setting the *escapesequence* option you can substitute escape sequences in strings using *PDF_convert_to_unicode( )*, using the same input and output encoding, e.g.

```
String s_plain = p.convert_to_unicode("utf16",
        s.character.getBytes("UTF-16"),
        "outputformat=utf16 escapesequence=true");
```

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

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

5.6.2 Character References

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

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

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

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

Examples:

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

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

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

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

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

- Control characters for overriding the default shaping behavior listed in Table 7.4.
- Control characters for overriding the default bidi formatting listed in Table 7.5.
- Control characters for Textflow line breaking and formatting listed in Table 9.1.

Examples:

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

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

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

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

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

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

- Character references with font-specific glyph names are useful in content strings to select alternate character forms (e.g. swash characters) and glyphs without any specific Unicode semantics (symbols, icons, and ornaments). Note that tabular figures and many other features are more easily implemented with OpenType features (see Section 7.3, »OpenType Layout Features«, page 164) if supported by the font.
- Names from the Adobe Glyph List (including the *uniXXXX* and *u1XXXX* forms) plus certain common »misnamed« glyph names are always accepted for content strings and hypertext strings.

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

```
&.#x9F;         bullet symbol in Wingdings font
&.#159;         bullet symbol in Wingdings font
```

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

```
p.fit_textline("Price: 500&euro;", x, y, "charref=true");
```

Supplying *charref* as text option enables character reference substitution for all content strings:

```
p.set_text_option("charref=true font=" + font + " fontsize=24");
p.fit_textline("Price: 500&euro;", x, y, "");
```

The *charref* option can also be set globally with *PDF_set_option()*. However, this is not recommended because it affects all name strings, hypertext strings and content strings, which may have undesired results.

As an alternative to setting the *charref* option you can substitute character references in strings using *PDF_convert_to_unicode()*, using the same input and output encoding, e.g.

```
String s_plain = p.convert_to_unicode("utf16",
        s.character.getBytes("UTF-16"),
        "outputformat=utf16 charref=true");
```

Additional notes on using character references:

- Character references can be used in all content strings, hypertext strings, and name strings. As an exception, font-specific glyph name references work only with contents strings as noted above.
- Character references are not substituted in text with *builtin* encoding. However, you can use character references for symbolic fonts by using *unicode* encoding.
- Character references are not substituted in option lists, but they are recognized in options with the *Unichar* data type; in this case the '&' and ';' decoration must be omitted. This recognition is always enabled; it is not subject to the *charref* option.
- In non-Unicode-capable language bindings character references must be expressed as two-byte values if *textformat=utf16*, *utf16be*, or *utf16le*. If *encoding=unicode* and *textformat=bytes* the character references must be expressed in ASCII (even on EBCDIC-based platforms).
- An ampersand character '&' is considered as start of a character reference if a subsequent semicolon ';' can be found and the characters between '&' and ';' are valid for a character reference. Otherwise the ampersand character '&' is left unchanged. In order to express an ampersand character '&' without starting a character reference it is recommended to use the character reference *&*.
- If a potential character reference sequence is invalid (e.g. *&#* followed by invalid decimal digits or '&' followed by an unknown entity name) the behavior depends on the option *glyphcheck*:
  glyphcheck=none: the sequence is kept as is;
  glyphcheck=replace: the sequence is replaced with the replacement character.
  glyphcheck=error: an error occurs and text processing stops. If *errorpolicy=exception* an exception is thrown.

6 Font Handling

6.1 Font Formats

6.1.1 TrueType Fonts

TrueType file formats. PDFlib supports vector-based TrueType fonts. PDFlib supports the following file formats for TrueType fonts:

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

TrueType font names. If you are working with font files you can assign an arbitrary API name to the font (see »Sources of Font Data«, page 140). This name is used for loading the font and may differ from the font file name or the font's internal name. 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)*.

6.1.2 OpenType Fonts

The OpenType font format combines PostScript and TrueType technology. It is implemented as an extension of the TrueType file format and offers a unified format. OpenType fonts may contain optional tables which can be used to enhance text output, e.g. ligatures and swash characters (see Section 7.3, »OpenType Layout Features«, page 164), as well as tables for complex script shaping (see Section 7.4, »Complex Script Output«, page 170). Note that neither the file name suffix nor the logo displayed by the Windows Explorer says anything about the presence or absence of OpenType layout features in a font. See Section 7.3, »OpenType Layout Features«, page 164, for more information.

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

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

signature it is displayed with the »O« logo; otherwise it is displayed with the »TT« logo. However, since a digital signature is not required in OpenType fonts this cannot be used a reliable criterion for distinguishing plain old TrueType fonts and OpenType fonts.

- The CID (Character ID) architecture is used for CJK fonts. Modern CID fonts are packaged as OpenType **.otf* fonts with PostScript outlines. From a practical standpoint they are indistinguishable from plain OpenType fonts. The Windows Explorer displays OpenType CID fonts with the »O« logo.
- OpenType Collections haven been introduced with the OpenType 1.7 specification. They are similar to TrueType Collections in that they package multiple related OpenType fonts into a single combined file. OpenType Collections use the **.ttc* or **.otc* suffix.
- OpenType font variations, also called variable fonts: fonts that use OpenType font variations mechanisms can be used to package multiple font faces within a font family (such as light, regular and bold) into a single font resource. Since font variations are not supported in PDF this mechanism cannot be used. OpenType font variations with TrueType outlines can be loaded with PDFlib, but only the default font instance without variations will be visible. OpenType font variations with PostScript outlines in a CFF2 table are rejected since they are incompatible with PDF.

6.1.3 WOFF Fonts

WOFF (Web Open Font Format) is a simple compressed file format for TrueType and OpenType fonts. It can be regarded as a new container format for existing font formats, but does not offer any new typographic features. WOFF has been designed for use on the Web and offers compression and subsetting features to achieve small font file sizes. WOFF is detailed in a W3C recommendation; the WOFF specification can be found at

`www.w3.org/TR/WOFF`

WOFF fonts typically use the file name extension *.woff*.

PDFlib supports WOFF fonts to the extent that the underlying TrueType or OpenType font is supported. For example, TrueType bitmap fonts packaged as WOFF are not supported.

6.1.4 PostScript Type 1 Fonts

Note The use of PostScript Type 1 fonts is deprecated.

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:

- The platform-independent AFM (Adobe Font Metrics) and the Windows-specific PFM (Printer Font Metrics) format for metrics information.
- The platform-independent PFA (Printer Font ASCII) and the Windows-specific PFB (Printer Font Binary) format for font outline information in the PostScript Type 1 format.

6.1.5 SING Fonts (Glyphlets)

SING fonts *(Smart Independent Glyphlets)* are technically an extension of the OpenType font format. SING fonts have been developed as a solution to the Gaiji problem with CJK text, i.e. custom glyphs which are not encoded in Unicode.

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

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

It is recommended to use SING fonts as fallback font with the *gaiji* suboption of the *forcechars* option of the *fallbackfonts* option of *PDF_load_font();* see Section 7.5.3, »EUDC and SING Fonts for Gaiji Characters«, page 178, for more information.

6.1.6 Type 3 Fonts

Unlike all other font formats, Type 3 fonts are not fetched from a disk file, but must be defined at runtime with standard PDFlib graphics functions. Type 3 fonts are useful for the following purposes:

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

Since all PDFlib features for vector graphics, raster images, and even text output can be used in Type 3 font definitions, there are no restrictions regarding the contents of the characters in a Type 3 font. Combined with the PDF import library PDI you can even import complex drawings as a PDF page, and use those for defining a character in a Type 3 font. However, Type 3 fonts are most often used for bitmapped glyphs since it is the only font format in PDF which supports raster images for glyphs. 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_ext(-1, "glyphname=circle width=1000 boundingbox={0 0 1000 1000}");
p.arc(500, 500, 500, 0, 360);
p.fill();
p.end_glyph();

p.begin_glyph_ext(-1, "glyphname=ring width=400 boundingbox={0 0 400 400}");
p.arc(200, 200, 200, 0, 360);
p.stroke();
p.end_glyph();

p.end_font();
```

Cookbook *Full code samples can be found in the Cookbook topics* `type3_fonts/starter_type3font`, `type3_fonts/type3_bitmaptext`, `type3_fonts/type3_rasterlogo`, *and* `type3_fonts/type3_vectorlogo`.

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

- If the font has been loaded with *encoding=unicode* the glyphs can be addressed with their Unicode value or with glyph name references of the form *&.<glyphname>;* as in the following example: *&.circle;*
- If the font has been loaded with *encoding=builtin* character codes can be used to address glyphs, where the code of each glyph corresponds to the order in which the glyphs have been created; the *.notdef* glyph always has code 0.
- If only Unicode values have been specified but no glyph names, PDFlib generates glyph names of the form *GXXX* where *XXX* is decimal number of the generated glyph.
- It is recommended to use the *inline* image option for defining bitmaps in Type 3 fonts. The *interpolate* option for images may be useful for enhancing the screen and print appearance of Type 3 bitmap fonts.
- 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* option for constructing the bitmap image.
- Due to restrictions in PDF consumers all characters used in text output must actually be defined in the font: if character code *x* is to be displayed with any text output function, and the encoding contains *glyphname* at position *x*, then *glyphname* must have been defined via *PDF_begin_glyph_ext()*.
- Some PDF consumers require a glyph named *.notdef* if codes will be used for which the corresponding glyph names are not defined in the font. The *.notdef* glyph must be present, but it may simply contain an empty glyph description.
- Type 3 glyph definitions do not supply any typographic properties such as ascender, descender, etc. However, these can be set by using the corresponding options in *PDF_load_font()*.

6.2 Unicode Characters and Glyphs

6.2.1 Glyph IDs

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

The assignment of glyph IDs depends on the font format:

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

PDFlib supports glyph selection via GID as an alternative to Unicode and other encodings (see »Glyphid encoding«, page 135). Direct GID addressing is only useful for specialized applications, e.g. printing font overview tables by querying the number of glyphs and iterating over all glyph IDs.

6.2.2 Unicode Mappings for Glyphs

Unicode mapping and ambiguous glyphs. PDFlib assigns Unicode values to all glyphs in a font. In some fonts a particular glyph may be used to represent multiple Unicode values. Common examples for such ambiguous glyphs are the empty glyph which represents U+0020 Space as well as U+00A0 No-Break Space, or a glyph which represents both U+2126 Ohm Sign and U+03A9 Greek Capital Letter Omega.

If multiple Unicode values are represented by the same glyph PDFlib creates a ToUnicode CMap which maps the glyph to one of the affected Unicode values. The other Unicode values are emitted with the same glyph ID, but are assigned an *ActualText* attribute with the appropriate Unicode value. This way correct semantics are preserved in the generated PDF output.

Text selection problems in Acrobat. Unfortunately, Acrobat DC sometimes has problems with the text selection of text with an *ActualText* attribute. Some glyphs cannot be selected and highlighted. Although the visual highlight is incomplete, the text contained in the *ActualText* attribute is copied to the clipboard correctly. You can work around the selection problems in the following ways:

- Avoid ambiguous glyphs in the first place.
- Use the text filter option *actualtext=false* to disable creation of the *ActualText* attribute.

However, these methods are not recommended since different characters are extracted with the same Unicode value which thwarts content repurposing.

Unmapped glyphs and the Private Use Area (PUA). In some situations the font may not provide a Unicode value for a particular glyph. In this case PDFlib assigns a value

from the Unicode Private Use Area (PUA, see Section 5.1, »Important Unicode Concepts«, page 105) to the glyph. Such glyphs are called *unmapped glyphs*. The number of unmapped glyphs in a font can be queried with the *unmappedglyphs* keyword of *PDF_info_font()*. Unmapped glyphs are represented by the Unicode replacement character U+FFFD in the font's ToUnicode CMap which controls searchability and text extraction. As a consequence, unmapped glyphs cannot be properly extracted as text from the generated PDF. However, this behavior can be changed which is particularly useful for CJK Gaiji characters; see »Preserving PUA values for Gaiji characters«, page 179, for details.

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

- The basis is the Unicode PUA range in the Basic Multilingual Plane (BMP), i.e. the range U+E000 - U+F8FF. Additional PUA values in plane 15 (U+F0000 to U+FFFFD) are used if required.
- PUA values which have already been assigned by the font internally are not used when creating new PUA values.
- PUA values in the Adobe range U+F600-F8FF are not used.

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

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

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

In some cases neither the *cmap* nor the *post* table provide Unicode values for all glyphs in the font. This is true for variant glyphs (e.g. swash characters), extended ligatures, and non-textual symbols outside the Unicode standard. In this case PDFlib assigns PUA values to the affected glyphs as described in »Unmapped glyphs and the Private Use Area (PUA)«, page 127.

Unicode mapping for Type 1 fonts. Type 1 fonts do not include explicit Unicode mappings, but assign a unique name to each glyph. PDFlib tries to assign a Unicode value based on this glyph name, using an internal mapping table which contains Unicode mappings for more than 7 000 common glyph names for a variety of languages and scripts. The mapping table includes ca. 4 200 glyph names from the Adobe Glyph List (AGL)[1]. However, Type 1 fonts may contain glyph names which are not included in the internal mapping table; this is especially true for Symbol fonts. In this case PDFlib assigns PUA values to the affected glyphs as described in »Unmapped glyphs and the Private Use Area (PUA)«, page 127.

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

Unicode mapping for Type 3 fonts. Since Type 3 fonts are also based on glyph names, they are treated in the same way as Type 1 fonts. An important difference, however, is

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

that the glyph names for Type 3 fonts are under user control (directly via the *uv* parameter or indirectly via the *glyphname* option of *PDF_begin_glyph_ext()*). It is therefore strongly recommended to either supply suitable Unicode values or appropriate AGL glyph names for the glyphs in user-defined Type 3 fonts. This ensures that proper Unicode values can be assigned by PDFlib, resulting in searchable text in the generated PDF documents.

6.2.3 Unicode Control Characters

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

- The control characters for overriding the default shaping behavior (listed in Table 7.4) and those for overriding the default bidi formatting (listed in Table 7.5) control complex script shaping and OpenType layout feature processing in Textline and Textflow. After evaluating these control characters they will be removed.
- The formatting control characters for line breaking and Textflow formatting listed in Table 9.1. After evaluating these control characters they will be removed.
- Other Unicode control characters in the ranges U+0001-U+0019 and U+007F-U+009F will be replaced with the *replacementchar* character.

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

6.3 The Text Processing Pipeline

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

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

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

6.3.1 Normalizing Input Strings to Unicode

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

- Unicode-capable language bindings: if a single-byte encoding has been specified UTF-16 text is converted to single-byte text by dropping the high-order bytes.
- Windows: convert multi-byte text (e.g. *cp932)* to Unicode.
- Replace escape sequences (see Section 5.6.1, »Escape Sequences«, page 118) with the corresponding numerical values.
- Resolve character references and replace them with the corresponding Unicode values (see Section 5.6.2, »Character References«, page 119, and next section below).
- Single-byte encodings: convert single-byte text to Unicode according to the specified encoding.
- Normalize the text to one of the Unicode normalization forms (e.g. NFC) according to the *normalize* option.

See also Section 6.2.2, »Unicode Mappings for Glyphs«, page 127, for more details regarding the Unicode assignments for various font formats and types of characters.

Character references with glyph names. A font may contain glyphs which are not directly accessible because the corresponding Unicode values are not known in advance (since PDFlib assigns PUA values at runtime). As an alternative for addressing such glyphs, character references with glyph names can be used; see Section 5.6.2, »Character References«, page 119, for a syntax description. These references are replaced with the corresponding Unicode values.

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

6.3.2 Converting Unicode Values to Glyph IDs

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

- For non-Unicode CMaps: invalid code sequences always trigger an exception.
- For *encoding=glyphid:* invalid glyph IDs are replaced with glyph ID 0. If *glyphcheck=error* an exception is thrown.

Force characters from fallback fonts. Replace Unicode values according to the *forcechars* suboption of the *fallbackfonts* option, and determine the glyph ID of the corresponding fallback font. For more information see Section 6.4.6, »Fallback Fonts«, page 146.

Resolve variation sequences. For some fonts Unicode characters may be followed by a variation selector which selects a specific glyph variant of the character (see Section 7.5.5, »Unicode Variation Selectors and Variation Sequences«, page 181). If the font contains a variant glyph for the variation sequence the glyph ID of the variant glyph is used instead of the original glyph ID.

Convert to glyph IDs. Convert the Unicode values to glyph IDs according to the mappings determined in Section 6.2.2, »Unicode Mappings for Glyphs«, page 127. If no corresponding glyph ID for a Unicode value was found in the font, the next steps depend on the *glyphcheck* option:

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

Glyph replacement. If *glyphcheck=replace,* unmappable Unicode values are recursively replaced as follows:

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

```
U+00A0 (NO-BREAK SPACE)          U+0020 (SPACE)
U+00AD (SOFT HYPHEN)             U+002D (HYPHEN-MINUS)
U+2010 (HYPHEN)                  U+002D (HYPHEN-MINUS)
U+03BC (GREEK SMALL LETTER MU)   U+00C5 (MICRO SIGN)
```

U+212B (ANGSTROM SIGN)	U+00B5 (LATIN CAPITAL LETTER A WITH RING ABOVE Å)
U+220F (N-ARY PRODUCT)	U+03A0 (GREEK CAPITAL LETTER PI)
U+2126 (OHM SIGN)	U+03A9 (GREEK CAPITAL LETTER OMEGA)

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

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

If none of these methods delivers a glyph for the Unicode value, the *replacementchar* option is evaluated as follows:

- If *replacementchar=auto* (which is the default) the characters U+00A0 (NO-BREAK SPACE) and U+0020 (SPACE) are tried. If these are still unavailable, the »missing glyph« symbol is used (not allowed in PDF/A, PDF/UA and PDF/X-4/5).
- If a Unicode character was specified as *replacementchar* it is used instead of the original character.
- If *replacementchar=drop*, the character is dropped from the input stream and no output is created.
- If *replacementchar=error* an exception is thrown. This may be used to avoid unreadable text output.

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

6.3.3 Transforming Glyph IDs

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

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

OpenType layout features. OpenType features can create ligatures, swash characters, small caps, and many other typographic variations by replacing one or more glyph IDs with other values. OpenType features are discussed in Section 7.3, »OpenType Layout Features«, page 164. OpenType layout features are relevant only for suitable fonts (see »Requirements for OpenType layout features«, page 166), and are applied according to the *features* option.

Complex script shaping. Shaping reorders the text and determines the appropriate variant glyph according to the position of a character (e.g. initial, middle, final, or isolated form of Arabic characters). Shaping is discussed in Section 7.4, »Complex Script Output«, page 170. It is relevant only for suitable fonts (see »Requirements for shaping«, page 170, and is applied according to the *shaping* option.

6.4 Loading Fonts

6.4.1 Selecting an Encoding for Text Fonts

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

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

PDFlib's text handling is based on the Unicode standard[1], almost identical to ISO 10646. Since most modern development environments support the Unicode standard our goal is to make it as easy as possible to use Unicode strings for creating PDF output. However, developers who don't work with Unicode are not required to switch their application to Unicode since legacy encodings can be used as well.

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

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

With *encoding=unicode* all glyphs in a font can be addressed; complex script shaping and OpenType layout features are supported. PDFlib checks whether the font contains a glyph for a requested Unicode value. If no glyph is available, a substitute glyph can be pulled from the same or another font (see Section 6.4.6, »Fallback Fonts«, page 146).

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

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

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

Single-byte encodings. 8-bit encodings (also called single-byte encodings) map each byte in a text string to a single character, and are thus limited to 255 different characters at a time (the value 0 is not available). This type of encoding is supported for all font formats. PDFlib checks whether the font contains glyphs which match the selected encoding. If a minimum number of usable glyphs is not reached, PDFlib will log a warning message. If no usable glyph at all for the selected encoding is available in the font, font

1. See www.unicode.org

loading will fail with the message *font doesn't support encoding*. PDFlib checks whether the font contains a glyph for a requested input value. If no glyph is available, a substitute glyph can be pulled from the same or another font (see Section 6.4.6, »Fallback Fonts«, page 146).

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

8-bit encodings are discussed in detail in Section 5.4, »Single-Byte (8-Bit) Encodings«, page 113. They can be pulled from various sources:

- A large number of predefined encodings according to Section 5.4, »Single-Byte (8-Bit) Encodings«, page 113. These cover the most important encodings in use on a variety of systems and in a variety of locales.
- User-defined encodings which can be supplied in an external file or constructed dynamically at runtime with *PDF_encoding_set_char( )*. These encodings can be based on glyph names or Unicode values.
- Encodings pulled from the operating system, also known as a *system encoding*. This feature is available on Windows, IBM IBM System i and IBM Z.

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

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

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

If the font does not contain any builtin encoding font loading fails (e.g. OpenType CJK fonts). You can use the *symbolfont* key in *PDF_info_font( )*. If it returns *false*, the font is a text font which can also be loaded with one of the common single-byte encodings. This is not possible if the *symbolfont* key returns *true*. The glyphs in such symbolic fonts can only be used if you know the corresponding code for each glyph (see Section 6.4.2, »Selecting an Encoding for symbolic Fonts«, page 135).

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

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

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

the Adobe CMaps or Windows codepages (see Section 5.5, »Chinese, Japanese, and Korean CMaps«, page 116).

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

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

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

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

Glyphid encoding. PDFlib supports *encoding=glyphid* for all font formats. With this encoding all glyphs in a font can be addressed, using the numbering scheme explained in Section 6.2.1, »Glyph IDs«, page 127. Numerical glyph IDs run from 0 to a theoretical maximum value of 65 565 (but fonts with such a large number of glyphs are not available). The maximum glyph ID value can be queried with the *maxcode* key in *PDF_info_font( )*.

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

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

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

6.4.2 Selecting an Encoding for symbolic Fonts

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

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

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

- If you prefer to work with 8-bit codes for addressing the glyphs in a symbolic font you can load the font with *encoding=builtin* and supply the 8-bit codes in the text. For example, the digit 4 (code 0x34) will select the check mark symbol in the Zapf-Dingbats font.

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

- The characters in the *Symbol* font all have proper Unicode values.
- The characters in the *ZapfDingbats* font have Unicode values in the range U+2007 - U+27BF.
- Microsoft's symbolic fonts, e.g. Wingdings and Webdings, use PUA Unicode values in the range U+F020 - U+F0FF (although the *charmap* application presents them with single-byte codes).
- For other fonts the Unicode values for individual glyphs in the font must be known in advance or must be determined at runtime with *PDF_info_font()*, e.g. for PostScript Type 1 fonts by supplying the glyph name.

Control characters. The Unicode control characters in the range U+0001 - U+001F which are listed in Table 9.1 are supported in Textflow even with *encoding=builtin*. Codes < 0x20 will be interpreted as control characters if the symbolic font does not contain any glyph for the code. This is true for the majority of symbolic fonts.

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

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

6.4.3 Example: Selecting a Glyph from the Wingdings Symbol Font

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

Understanding the characters in the font. First let's collect some information about the target character in the font, using the Windows *charmap* application (see Figure 6.1):

- *Charmap* displays the glyphs in the Wingdings font, but does not provide any Unicode access in the *Advanced view*. This is a result of the fact that the font contains symbolic glyphs for which no standardized Unicode values are registered. Instead, the glyphs in the font use dummy Unicode values in the Private Use Area (PUA). The *charmap* application does not reveal these values.
- If you look at the lower left corner of the *charmap* window or hover the mouse over the *smileface* character, the *Character code: 0x4A* is displayed. This is the glyph's byte code.
  This code corresponds to the uppercase *J* character in the Winansi encoding. For example, if you copy the character to the clipboard the corresponding Unicode value U+004A, i.e. character *J* will result from pasting the clipboard contents to a text-only application. Nevertheless, this is not the character's Unicode value and therefore U+004A or *J* can not be used to select it in Unicode workflows.
- The Unicode character used internally in the font is not displayed in *charmap*. However, symbolic fonts provided by Microsoft use the following simple rule:

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

  For the *smileface* glyph this yields the Unicode value U+F04A.
- The corresponding glyph name can be retrieved with a font editor and similar tools. In our example it is *smileface*.

You can use *PDF_info_font( )* to query Unicode values, glyph names or codes, see Section 6.6.2, »Font-specific Encoding, Unicode, and Glyph Name Queries«, page 153.

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

- If you know the PUA Unicode value which is assigned to the character in the font you can use a numerical character reference (see »Numerical character references«, page 119):

  

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

  \xEF\x81\x8A

  Unicode values can not be used with the combination of *encoding=builtin* and *textformat=bytes*.
- If you know the character code you can use a byte value reference (see »Byte value references«, page 120):

  ```
  &.#x4A;
  ```

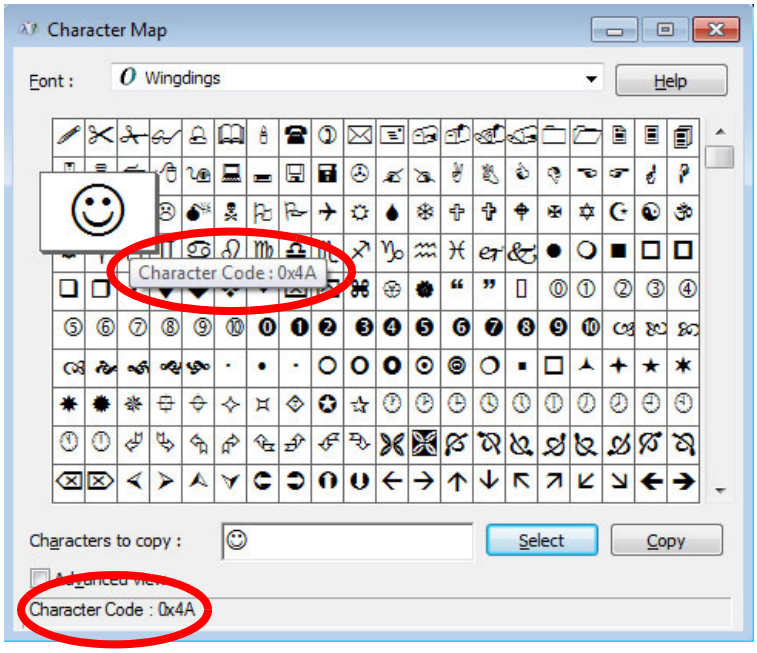


Fig. 6.1
Windows character map with the Wingdings font

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

```
J
\x4A
```

- If you know the glyph name you can use a glyph name reference (see »Glyph name references«, page 120):

  ```
  &.smileface;
  ```

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

Table 6.1 lists methods for Unicode-capable language bindings such as Java and .NET.

Table 6.1 Addressing the `smileface` *glyph in the Wingdings font with Unicode-capable language bindings (e.g. Java)*

encoding	*additional options*	*input string*	*visible result on the page*
`unicode`		*\uF04A* [1]	☺
	`charref`	**	☺
	`charref`	*&.#x4A;*	☺
	`charref`	*&.smileface;*	☺
		J [2]	*(space glyph if available, otherwise .notdef glyph)*
	`escapesequence`	*\x4A*	*(space glyph if available, otherwise .notdef glyph)*
`builtin`	*(same as above with* `encoding=unicode`*)*		

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

Table 6.2 lists methods for non-Unicode-capable language bindings such as C.

Table 6.2 Addressing the `smileface` *glyph in the Wingdings font with non-Unicode-capable language bindings (e.g. C)*

encoding	*textformat*	*additional options*	*input string*	*visible result on the page*
unicode	utf16		*\xFo\x4A* [1]	☺
	utf8	charref	*&#xFo4A;*	☺
		charref	*&.#x4A;*	☺
		charref	*&.smileface;*	☺
			ï•Š [2]	☺
		escapesequence[3]	*\xEF\x81\x8A* [4]	☺
			J [5]	*(space glyph if available, otherwise .notdef glyph)*
		escapesequence	*\x4A*	*(space glyph if available, otherwise .notdef glyph)*
	bytes	charref	*&#xFo4A;*	☺
		bytes	*&.#x4A;*	☺
		charref	*&.smileface;*	☺
			J	*(space glyph if available, otherwise .notdef glyph)*
		escapesequence	*\x4A*	*(space glyph if available, otherwise .notdef glyph)*
builtin	utf16, utf8	*(same as above with* `encoding=unicode`*)*		
	bytes	charref	*&#xFo4A;*	🕮✂☒☞🗀🗐✌▬
		charref	*&.#x4A;*	🕮👐✂☒🗐✌▬
		charref	*&.smileface;*	🕮👐♦❍♓●♏➚♋♍♏▬
			J	☺
		escapesequence	*\x4A*	☺

1. Must be expressed as `\xFO\x4A` *or* `\x4A\xFO` *depending on byte ordering; note that* `\x` *designates C escape syntax*
2. Winansi characters for the three-byte sequence \xEF \x81 \x8A
3. The `escapesequence` *option is only required if the programming language doesn't offer any syntax for direct byte values.*
4. Three-byte UTF-8 sequence for U+Fo4A
5. Winansi character for the byte code \x4A

6.4.4 Searching for Fonts

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

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

- Disk-based or virtual font files
- Fonts pulled from the Windows or macOS 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.

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

With the *enumeratefonts* option PDFlib can be instructed to collect all fonts which are accessible on the search path (see »File search and the SearchPath resource category«, page 56). Using the *saveresources* option the current list of PDFlib resources can be written to a disk file:

```
/* add font directory to the search path */
p.set_option("searchpath={{C:/fonts}}");

/* enumerate all fonts on the searchpath and create a UPR file */
p.set_option("enumeratefonts saveresources={filename=C:/fonts/pdflib.upr}");
```

Font name aliasing. Each font may have an arbitrary number of alias names. This may be useful in situations where fonts are requested via an artificial or virtual name which must be mapped to a physical font. Font name aliases can be created with the *FontnameAlias* resource category (see Table 3.1, page 56) as in the following example:

```
p.set_option("FontnameAlias={sans Helvetica}");
```

The alias name to the left can be chosen arbitrarily and can be used for loading the font under its new alias name. The name on the right must be a valid API name of a font, e.g. the name of a host font or a font which has been connected to a font resource with one of the font resource categories *FontOutline* etc.

Search order for fonts. The font name supplied to PDFlib is a name string. If the specified name is a font name alias it will be replaced with the corresponding API font name. PDFlib uses the API font name to search for fonts of various types in the order described below. The search process stops as soon as one of the steps located a usable font:

- The font name matches the name of a Type 3 font which has previously been created in the same document with *PDF_begin_font()* (see Section 6.1.6, »Type 3 Fonts«, page 125).

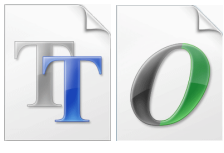
- The font name matches the name in a *FontOutline* resource which connects the font name with the name of a TrueType or OpenType font file.
- The font name matches the name in a *FontAFM* or *FontPFM* resource which connects the font name with the name of a PostScript Type 1 font metrics file.
- The font name matches the name in a *FontOutline* resource which connects the font name with the name of an SVG font file and doesn't match the name in a *HostFont* resource.
- The font name matches the name in a *HostFont* resource which connects the font name with the name of a font installed on the system.
- The font name matches the name of a Latin core font (see »Latin core fonts«, page 142).
- The name matches the name of a host font installed on the system (see Section 6.4.5, »Host Fonts on Windows and macOS«, page 144).
- The font name matches the base name (i.e. without file name suffix) of a font file.

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

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

Details regarding the resource categories can be found in Section 3.1.4, »Resource Configuration and File Search«, page 55. The following sections discuss font loading for the various classes of fonts in more detail.

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



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

The font name to the left of the equal sign (called the font's API name) can be chosen arbitrarily:

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

As an alternative to runtime configuration via *PDF_set_option( )*, the *FontOutline* resource can be configured in a UPR file (see Section 3.1.4, »Resource Configuration and File Search«, page 55). In order to avoid absolute file names you can use the *SearchPath* resource category (again, the *SearchPath* resource category can alternatively be configured in a UPR file), for example:

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

TrueType and OpenType Collections. In order to select a font which is contained in a TrueType or OpenType Collection (TTC/OTC, see Section 7.5.1, »Using TrueType and OpenType CJK Fonts«, page 177) file you directly specify the name of the font:

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

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

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

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

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

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

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

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

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

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

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

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

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

Core fonts found in the internal list are never embedded. In order to embed one of these fonts you must configure a font outline file. In some situations, e.g. for font names in SVG, it may be useful to attach a style keyword to the font name in the font outline configuration, e.g.

```
p.set_option("FontOutline={Helvetica,Bold=/usr/fonts/HelvBd.ttf}");
```

Host fonts. If a font name is not connected to any file name via resources, PDFlib will search the font in the list of fonts installed on Windows or macOS. Fonts installed on

the system are called *host fonts*. Host font names must be encoded in ASCII. On Windows Unicode can also be used. See Section 6.4.5, »Host Fonts on Windows and macOS«, page 144, 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 (this syntax can also be used with the Latin core fonts):

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

In order to load a host font with the name of one of the core fonts, the font name must be connected to the desired host font name via the *HostFont* resource category. The following fragment makes sure that instead of using the built-in core font data, the Symbol font metrics and outline data will be taken from the host system:

```
p.set_option("HostFont={Symbol=Symbol}");
font = p.load_font("Symbol", "unicode", "embedding");
```

The API font name to the left of the equal sign can be chosen arbitrarily. Typically the name of the host font is used on both sides of the equal sign.

Extension-based search for font files. All font types except Type 3 fonts can be searched by using the specified font name as the base name (without any file suffix) of a font metrics or outline file. If PDFlib couldn't find any font with the specified name it will loop over all entries in the *SearchPath* resource category, and add all known file name suffixes to the supplied font name in an attempt to locate the font metrics or outline data. The details of the extension-based search algorithm are as follows:

- The following suffixes will be added to the font name, and the resulting file names tried one after the other to locate the font metrics (and outline in the case of TrueType and OpenType fonts):

  ```
  .tte .ttf .otf .gai .woff .cef .afm .pfm .ttc, .otc, .svg, .svgz,
  .TTE .TTF .OTF .GAI .WOFF .CEF .AFM .PFM .TTC, .OTC, .SVG, .SVGZ
  ```

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

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

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

  ```
  Font cannot be embedded (PFA or PFB font file not found)
  ```

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

This means that PDFlib will find a font without any manual configuration provided the corresponding font file name 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 achieve the same effect with respect to locating the font outline file:

```
p.set_option("FontOutline={Arial=/usr/fonts/Arial.ttf}");
font = p.load_font("Arial", "unicode", "");
```

and

```
p.set_option("SearchPath={{/usr/fonts}}");
font = p.load_font("Arial", "unicode", "");
```

Standard CJK fonts. Acrobat supports various standard fonts for CJK text; see Section 7.5.6, »Standard CJK Fonts«, page 182, for more details and a list of font names. PDFlib will find a standard CJK font at the very beginning of the font search if the specified font name matches the name of a standard CJK font, the specified encoding is the name of one of the predefined CMaps, and the *embedding* option was not specified.

Note that the concept of standard CJK fonts is deprecated. Configure suitable fonts for use with PDFlib.

Type 3 fonts. Type 3 fonts must be defined at runtime by defining the glyphs with standard PDFlib graphics functions (see Section 6.1.6, »Type 3 Fonts«, page 125). If the font name supplied to *PDF_begin_font()* matches the font name requested with *PDF_load_font()* the font will be selected at the beginning of the font search. Example:

```
p.begin_font("PDFlibLogoFont", 0.001, 0.0, 0.0, 0.001, 0.0, 0.0, "");

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

6.4.5 Host Fonts on Windows and macOS

On macOS 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 6.1.4, »PostScript Type 1 Fonts«, page 124, and Section 6.1.1, »TrueType Fonts«, page 123. Host font search can be disabled with the *usehostfonts* option of *PDF_set_option()*.

Finding host font names on Windows. You can find the name of an installed font by double-clicking the font file and taking note of the full font name which is displayed in the window title. Some fonts may have parts of their name localized according to the respective Windows version in use. For example, the common font name portion *Bold* may appear as the translated word *Fett* on a German system. In order to retrieve the host font data from the Windows system you must use the translated form of the font name in PDFlib (e.g. *Arial Fett)*, or use font style names (see below). However, in order to retrieve the font data directly from file you must use the generic (non-localized) form of the font name (e.g. *Arial Bold)*.

Note You can avoid this internationalization problem by appending font style names (e.g. »`,Bold`«*, see below) to the font name instead of using localized font name variants.*

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

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

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

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

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

The keywords are case-insensitive. The *italic* keyword can be specified alternatively or in addition to the above. If two style names are used both must be separated with a comma, for example:

```
font = p.load_font("Verdana,Bold,Italic", "unicode", "");
```

Numerical font weight values can be used as an equivalent alternative to font style names:

```
0 (none), 100 (thin), 200 (extralight), 300 (light), 400 (normal), 500 (medium), 600
(semibold), 700 (bold), 800 (extrabold), 900 (black)
```

The following example will select the bold variant of a font:

```
font = p.load_font("Verdana,700", "unicode", "");
```

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

Windows font substitution. Windows can automatically substitute fonts based on certain registry entries. This kind of font substitution also affects PDFlib's host font mechanism, but is completely under control of the Windows operating system. For example, if the Helvetica font is requested Windows may deliver Arial instead, depending on the following registry entry:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\FontSubstitutes
```

For more information about Windows font substitution please refer to Microsoft documentation.

Host font names on macOS. Using the *Font Book* utility, which is part of macOS, you can find the names of installed host fonts. In order to programmatically create lists of host fonts we recommend Apple's Font Tool Suite[1]. This set of command-line utilities

contains a program called *ftxinstalledfonts* which is useful for determining the exact names of all installed fonts. PDFlib supports several flavors of host font names:

- »Unique« font names: these are font names which can be encoded in Unicode, e.g. for East-Asian fonts. In order to determine unique font names issue the following command in a terminal window (in some cases the output contains entries with a ':' which must be removed):

  ```
  ftxinstalledfonts -u
  ```

- PostScript font names. In order to determine PostScript font names issue the following command in a terminal window:

  ```
  ftxinstalledfonts -p
  ```

Potential problem with host font access on macOS. 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.

6.4.6 Fallback Fonts

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

Fallback fonts provide a powerful mechanism which deals with shortcomings in fonts and encodings. They are also useful for combining fonts for different scripts. For example, a font for the Arabic script may not support Latin characters (for example, the Google Noto font family is designed this way). Fallback fonts can be used in many situations to facilitate text output since necessary font changes are applied automatically by PDFlib. This mechanism augments a given font (called the base font) by merging glyphs from one ore more other fonts into the base font. More precisely: the fonts are not actually modified, but PDFlib applies all necessary font changes in the PDF page description automatically. Fallback fonts offer the following features:

- Glyphs which are unavailable in the base font are automatically searched in one or more fallback fonts. In other words, you can add glyphs to a font. Since multiple fallback fonts can be attached to the base font, you can effectively use all Unicode characters for which at least one of the fonts contains a suitable glyph.
- Glyphs from a particular fallback font can be used to override glyphs in the base font, i.e. you can replace glyphs in a font. You can replace one or more individual glyphs, or specify one or more ranges of Unicode characters which will be replaced.

The size and vertical position of glyphs from a fallback font can be adjusted to match the base font. Somewhat surprisingly, the base font itself can also be used as a fallback font (with the same or a different encoding). This can be used to implement the following tricks:

- Using the base font itself as fallback font can be used to adjust the size or position of some or all glyphs in a font.
- You can add characters outside the actual encoding of the base font.

The fallback font mechanism is controlled by the *fallbackfonts* font loading option, and affects all text output functions. As with all font loading options, the *fallbackfonts* option can be provided in explicit calls to *PDF_load_font( ),* or in option lists for implicit

1. See developer.apple.com/fonts

font loading. Since more than one fallback font can be specified for a base font, the *fallbackfonts* option expects a list of option lists (i.e. an extra set of braces is required).

PDF_info_font( ) can be used to query the results of the fallback font mechanism (see Section 6.6.3, »Querying Codepage Coverage and Fallback Fonts«, page 154).

Caveats. Note the following when working with fallback fonts:

- Not all font combinations result in typographically pleasing results. Care should be taken to use only fallback fonts in which the glyph design matches the glyph design of the base font.
- Font loading options for the fallback font(s) must be specified separately in the *fallbackfonts* option list. For example, if *embedding* is specified for the base font, the fallback fonts will not automatically be embedded.
- Fallback fonts work only if the fonts contain proper Unicode information. The replacement glyph(s) must have the same Unicode value as the replaced glyph.
- Script-specific shaping (options *shaping, script, locale)* and OpenType features (options *features, script, language)* will only be applied to glyphs within the same font, but not across glyphs from the base font and one or more fallback fonts.
- The underline/overline/strikeout features must be used with care when working with fallback fonts, as well as the *ascender* and similar typographic values. The underline thickness or position defined in the base font may not match the values in the fallback font. As a result, the underline position or thickness may jump in unpleasant ways. A simple workaround against such artifacts is to specify a unified value with the *underlineposition* and *underlinewidth* options of *PDF_fit_textline( )* and *PDF_add/create_textflow( )*. This value should be selected so that it works with the base font and all fallback fonts.

In the sections below we describe several important use cases of fallback fonts and demonstrate the corresponding option lists.

Add mathematical characters to a text font. As a very rough solution in case of missing mathematical glyphs you can use the following font loading option list for the *fallbackfonts* option to add mathematical glyphs from the Symbol font to a text font:

```
fallbackfonts={{fontname=Symbol encoding=unicode}}
```

Combine fonts for use with multiple scripts. In some situations the script of incoming text data is not known in advance. For example, a database may contain Latin, Greek, and Cyrillic text, but the available fonts cover only one of these scripts at a time. Instead of determining the script and selecting an appropriate font you can construct a font which chains together several fonts, effectively covering the superset of all scripts. Use the following font loading option list for the *fallbackfonts* option to add Greek and Cyrillic fonts to a Latin font:

```
fallbackfonts={
 {fontname=Times-Greek encoding=unicode embedding forcechars={U+0391-U+03F5}}
 {fontname=Times-Cyrillic encoding=unicode embedding forcechars={U+0401-U+0490}}
}
```

Extend 8-bit encodings. If your input data is restricted to a legacy 8-bit encoding you can nevertheless use characters outside this encoding, taking advantage of fallback fonts (where the base font itself serves as a fallback font) and PDFlib's character refer-

ence mechanism to address characters outside the encoding. Assuming you loaded the Helvetica font with *encoding=iso8859-1* (this encoding does not include the Euro character), you can use the following font loading option list for the *fallbackfonts* option to add the Euro glyph to the font:

```
fallbackfonts={{fontname=Helvetica encoding=unicode forcechars=euro}}
```

Since the input encoding does not include the Euro character you cannot address it with an 8-bit value. In order to work around this restriction use character or glyph name references, e.g. *€* (see Section 5.6.2, »Character References«, page 119).

Enlarge some or all glyphs in a font. Fallback fonts can also be used to enlarge some or all glyphs in a font without changing the font size. Again, the base font itself will be used as fallback font. This feature can be useful to make different font designs visually compatible without adjusting the fontsize in the code. Use the following font loading option list for the *fallbackfonts* option to enlarge all glyphs in the specified range to 120%:

```
fallbackfonts={
 {fontname=Times-Italic encoding=unicode forcechars={U+0020-U+00FF} fontsize=120%}
}
```

Add an enlarged pictogram. Use the following font loading option list for the *fallbackfonts* option to pull a symbol from the ZapfDingbats font:

```
fallbackfonts={
 {fontname=ZapfDingbats encoding=unicode forcechars=.a12 fontsize=150% textrise=-15%}
}
```

Again, we use the *fontsize* and *textrise* suboptions to adjust the symbol's size and position to the base font.

Replace glyphs in a CJK font. You can use the following font loading option list for the *fallbackfonts* option to replace the Latin characters in the ASCII range with those from another font:

```
fallbackfonts={
 {fontname=Courier-Bold encoding=unicode forcechars={U+0020-U+007E}}
}
```

Add Latin characters to an Arabic font. This use case is detailed in Section 7.4.5, »Arabic Text Formatting«, page 176.

Identify missing glyphs. The font *Unicode BMP Fallback SIL*, which is freely available, displays the hexadecimal value of each Unicode character instead of the actual glyph. This font can be very useful for diagnosing font-related problems in the workflow. You can use the following font loading option list for the *fallbackfonts* option to augment any font with this special fallback font to visualize missing characters:

```
fallbackfonts={{fontname={Unicode BMP Fallback SIL} encoding=unicode}}
```

Add Gaiji characters to a font. This use case is detailed in Section 7.5.3, »EUDC and SING Fonts for Gaiji Characters«, page 178.

6.5 Font Embedding and Subsetting

6.5.1 Font Embedding

PDF font embedding and font substitution in Acrobat. PDF documents can include font data in various formats to ensure proper text display. Alternatively, a font descriptor containing only the character metrics and some general font information (but not the actual glyph outlines) can be embedded. If a font is not embedded in a PDF document, Acrobat will take it from the target system if available and configured *(»Use Local Fonts«)*, or try to build a substitute font according to the font descriptor. The use of substitution fonts results in readable text, but the glyphs may look different from the original font. Similarly, substitution fonts don't work if complex script shaping or OpenType layout features have been used. For these reasons font embedding is generally recommended unless you know that the documents are displayed on the target systems acceptably even without embedded fonts. Such PDF files are inherently nonportable, but may be of use in controlled environments, such as corporate networks where the required fonts are known to be available on all workstations.

Embedding fonts with PDFlib. Font embedding is controlled by the *embedding* option when loading a font (although in some cases PDFlib enforces font embedding):

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

Table 6.3 lists different situations with respect to font usage, each of which imposes different requirements on the font and metrics files required by PDFlib. In addition to the requirements listed in Table 6.3 the corresponding CMap files (plus in some cases the Unicode mapping CMap for the respective character collection, e.g. *Adobe-Japan1-UCS2)* must be available in order to use a (standard or custom) CJK font with any of the standard CMaps.

Font embedding for fonts which are exclusively used for invisible text (mainly useful for OCR results) can be controlled with the *optimizeinvisible* option when loading the font.

Table 6.3 Different font usage situations and required files

font usage	*font metrics file required?*	*font outline file required?*
one of the 14 core fonts	*no*	*only if* `embedding=true` *and* `skipembedding={latincore}` *is not set*
TrueType, OpenType, or Type 1 host fonts installed on macOS or Windows	*no*	*no*
non-core Type 1 fonts	*yes*	*only if* `embedding=true`
TrueType, OpenType, SING, WOFF and SVG fonts	*n/a*	*yes*

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 other type foundries allow font embedding provided subsetting is ap-

plied 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.

The legal warning above should be kept in mind especially for Web fonts since most vendors of fonts for use on the Web do not allow embedding of such fonts in PDF documents.

6.5.2 Font Subsetting

In order to decrease the size of the PDF output, PDFlib can embed only those glyphs of a font which are actually used in the document. This process is called font subsetting. It creates a new font which contains fewer glyphs than the original font, and omits font information which is not required for PDF viewing. 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
- WOFF fonts
- Type 3 fonts (special handling required, see »Type 3 font subsetting«, page 151.)

When a font for which subsetting has been requested is used in a document, PDFlib will keep track of the characters actually used for text output. There are several controls for the subsetting behavior (assuming *autosubsetting* is not specified):

- The default subsetting behavior is controlled by the *autosubsetting* option. 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* option 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. The following font option sets the subset limit to 75%:

  ```
  subsetlimit=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* option 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.

Specifying the initial font subset. Font subsets contain outline descriptions for all glyphs used in the document. This means that the generated font subsets will vary among documents since a different set of characters (and therefore glyphs) is generally used in each document. Different font subsets can be problematic when merging many small documents with embedded font subsets to a larger document: the embedded subsets cannot be removed since they are all different.

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

For this scenario PDFlib allows you to specify the initial contents of a font subset with the *initialsubset* option of *PDF_load_font( )*. While PDFlib starts with an empty subset by default and adds glyphs as required by the generated text output, the *initialsubset* option can be used to specify a non-empty subset. For example, if you know that only Latin-1 text output will be generated and the font contains many more glyphs, you can specify the first Unicode block as initial subset:

```
initialsubset={U+0020-U+00FF}
```

This means that the glyphs for all Unicode characters in the specified range will be included in the subset. If this range has been selected so that it covers all text in the generated documents, the generated font subsets will be identical in all documents. As a result, when combining such documents later to a single PDF the identical font subsets can be removed with the *optimize* option of *PDF_begin_document( )*.

Type 3 font subsetting. Type 3 fonts must be defined and therefore embedded before they can be used in a document (because the glyph widths are required). On the other hand, subsetting is only possible after creating all pages (since the glyphs used in the document must be known to determine the proper subset). In order to avoid this conflict, PDFlib supports widths-only Type 3 fonts. If you need subsetting for a Type 3 font you must define the font in two passes:

- The first pass with the *widthsonly* option of *PDF_begin_font( )* must be done before using the font. It defines only the font and glyph metrics (widths); the font matrix in *PDF_begin_font( )* as well as *wx* and the glyph bounding box in *PDF_begin_glyph_ext( ))* must be supplied and must accurately describe the actual glyph metrics. Only *PDF_begin_glyph_ext( )* and *PDF_end_glyph( )* are required for each glyph, but not any other calls for defining the actual glyph shape. If other functions are called between start and end of a glyph description, they will not have any effect on the PDF output, and will not trigger an exception.
- The second pass must be done after creating all text in this font, and defines the actual glyph outlines or bitmaps. Font and glyph metrics are ignored since they are already known from the first pass. After the last page has been created, PDFlib also knows which glyphs have been used in the document, and will only embed the required glyph descriptions to construct the font subset.
  API function calls for the descriptions of unused glyphs are silently ignored. Functions which can return an error code (e.g. *PDF_load_image( )) return an error value which must be ignored by the application. In order to distinguish unused glyphs from real errors the error message number returned by PDF_get_errnum( ) must be checked: if it is zero, the glyph is ignored, i.e. no real error happened.*

The same set of glyphs must be provided in pass 1 and pass 2. A Type 3 font with subsetting can only be loaded once with *PDF_load_font( )*.

Cookbook A full code sample can be found in the Cookbook topic `type3_fonts/type3_subsetting`.

6.6 Querying Font Information

PDF_info_font() can be used to query useful information related to fonts, encodings, Unicode, and glyphs. Depending on the type of query, a valid font handle may be required as parameter for this function. In all examples below we use the variables described in Table 6.4.

Table 6.4 Variables for use in the examples for PDF_info_font()

variable	comments
`int uv;`	*Numerical Unicode value; as an alternative glyph name references without the* & *and* ; *decoration can be used in the option list, e.g.* `unicode=euro`. *For more details see the description of the Unichar option list data type in the PDFlib API Reference.*
`int c;`	*8-bit character code*
`int gid;`	*glyph id*
`int cid;`	*CID value*
`String gn;`	*glyph name*
`int gn_idx;`	*String index for a glyph name; if* `gn_idx` *is different from* `-1` *the corresponding string can be retrieved as follows:* `gn = p.get_string(gn_idx, "");`
`String enc;`	*encoding name*
`int font;`	*valid font handle created with PDF_load_font()*

If the requested combination of keyword and option(s) is not available, *PDF_info_font()* will return -1. This must be checked by the client application and can be used to check whether or not a required glyph is present in a font.

Each of the sample code lines below can be used in isolation since they do not depend on each other.

6.6.1 Font-independent Encoding, Unicode, and Glyph Name Queries

Encoding queries. Encoding queries do not require any valid font handle, i.e. the value *-1* (in PHP: 0) can be supplied for the *font* parameter of *PDF_info_font()*. Only glyph names known internally to PDFlib can be supplied in *gn*, but not any font-specific glyph names.

Query the 8-bit code of a Unicode character or a named glyph in an 8-bit encoding:

```
c = (int) p.info_font(-1, "code", "unicode=" + uv + " encoding=" + enc);
c = (int) p.info_font(-1, "code", "glyphname=" + gn + " encoding=" + enc);
```

Query the Unicode value of an 8-bit code or a named glyph in an 8-bit encoding:

```
uv = (int) p.info_font(-1, "unicode", "code=" + c + " encoding=" + enc);
uv = (int) p.info_font(-1, "unicode", "glyphname=" + gn + " encoding=" + enc);
```

Query the registered glyph name of an 8-bit code or a Unicode value in an 8-bit encoding:

```
gn_idx = (int) p.info_font(-1, "glyphname", "code=" + c + " encoding=" + enc);
gn_idx = (int) p.info_font(-1, "glyphname", "unicode=" + uv + " encoding=" + enc);
```

```
/* retrieve the actual glyph name using the string index */
gn = p.get_string(gn_idx, "");
```

Unicode and glyph name queries. *PDF_info_font()* can also be used to perform queries which are independent from a specific 8-bit encoding, but affect the relationship of Unicode values and glyph names known to PDFlib internally. Since these queries are independent from any font, a valid font handle is not required.

Query the Unicode value of an internally known glyph name:

```
uv = (int) p.info_font(-1, "unicode", "glyphname=" + gn + " encoding=unicode");
```

Query the internal glyph name of a Unicode value:

```
gn_idx = (int) p.info_font(-1, "glyphname", "unicode=" + uv + " encoding=unicode");

/* retrieve the actual glyph name using the string index */
gn = p.get_string(gn_idx, "");
```

6.6.2 Font-specific Encoding, Unicode, and Glyph Name Queries

The following queries relate to a specific font which must be identified by a valid font handle. The *gn* variable can be used to supply internally known glyph names as well as font-specific glyph names. In all examples below the return value -1 means that the font does not contain the requested glyph.

Query the 8-bit codes for a Unicode value, glyph ID, named glyph, or CID in a font which has been loaded with an 8-bit encoding:

```
c = (int) p.info_font(font, "code", "unicode=" + uv);
c = (int) p.info_font(font, "code", "glyphid=" + gid);
c = (int) p.info_font(font, "code", "glyphname=" + gn);
c = (int) p.info_font(font, "code", "cid=" + cid);
```

Query the Unicode value for a code, glyph ID, named glyph, or CID in a font:

```
uv = (int) p.info_font(font, "unicode", "code=" + c);
uv = (int) p.info_font(font, "unicode", "glyphid=" + gid);
uv = (int) p.info_font(font, "unicode", "glyphname=" + gn);
uv = (int) p.info_font(font, "unicode", "cid=" + cid);
```

Query the glyph id for a code, Unicode value, named glyph, or CID in a font:

```
gid = (int) p.info_font(font, "glyphid", "code=" + c);
gid = (int) p.info_font(font, "glyphid", "unicode=" + uv);
gid = (int) p.info_font(font, "glyphid", "glyphname=" + gn);
gid = (int) p.info_font(font, "glyphid", "cid=" + cid);
```

Query the glyph id for a code, Unicode value, or named glyph in a font with respect to an arbitrary 8-bit encoding:

```
gid = (int) p.info_font(font, "glyphid", "code=" + c + " encoding" + enc);
gid = (int) p.info_font(font, "glyphid", "unicode=" + uv + " encoding=" + enc);
gid = (int) p.info_font(font, "glyphid", "glyphname=" + gn + " encoding=" + enc);
```

Query the font-specific name of a glyph specified by code, Unicode value, glyph ID, or CID:

```
gn_idx = (int) p.info_font(font, "glyphname", "code=" + c);
gn_idx = (int) p.info_font(font, "glyphname", "unicode=" + uv);
gn_idx = (int) p.info_font(font, "glyphname", "glyphid=" + gid);
gn_idx = (int) p.info_font(font, "glyphname", "cid=" + cid);

/* retrieve the actual glyph name using the string index */
gn = p.get_string(gn_idx, "");
```

Checking glyph availability. Using *PDF_info_font()* you can check whether a particular font contains the glyphs you need for your application. As an example, the following code checks whether the Euro glyph is contained in a font:

```
/* We could also use "unicode=U+20AC" below */
if (p.info_font(font, "code", "unicode=euro") == -1)
{
        /* no glyph for Euro sign available in the font */
}
```

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

Alternatively, you can call *PDF_info_textline()* to check the number of unmapped characters for a given text string, i.e. the number of characters in the string for which no appropriate glyph is available in the font. The following code fragment queries results for a string containing a single Euro character (which is expressed with a glyph name reference). If one unmapped character is found this means that the font does not contain any glyph for the Euro sign:

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

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

6.6.3 Querying Codepage Coverage and Fallback Fonts

PDF_info_font() can also be used to check whether a font is suited for creating text output in a certain language or script, provided the codepage is known which is required for the text. Codepage coverage is encoded in the OS/2 table of the font. Note that it is up to the font designer to decide what exactly it means that a font support a particular codepage. Even if a font claims to support a specific codepage this does not necessarily mean that it contains glyphs for all characters in this codepage. If more precise coverage information is required you can query the availability of all required characters as demonstrated in Section 6.6.2, »Font-specific Encoding, Unicode, and Glyph Name Queries«, page 153.

Checking whether a font supports a codepage. The following fragment checks whether a font supports a particular codepage:

```
String cp="cp1254";

result = (int) p.info_font(font, "codepage", "name=" + cp);
```

```
if (result == -1)
        System.err.println("Codepage coverage unknown");
else if (result == 0)
        System.err.println("Codepage not supported by this font");
else
        System.err.println("Codepage supported by this font");
```

Retrieving a list of all supported codepages. The following fragment queries a list of all codepages supported by a TrueType or OpenType font:

```
cp_idx = (int) p.info_font(font, "codepagelist", "");

if (cp_idx == -1)
        System.err.println("Codepage list unknown");
else
{
        System.err.println("Codepage list:");
        System.err.println(p.get_string(cp_idx, ""));
}
```

This will create the following list for the common Arial font:

```
cp1252 cp1250 cp1251 cp1253 cp1254 cp1255 cp1256 cp1257 cp1258 cp874 cp932 cp936 cp949
cp950 cp1361
```

Query fallback glyphs. You can use *PDF_info_font()* to query the results of the fallback font mechanism (see Section 6.4.6, »Fallback Fonts«, page 146, for details on fallback fonts). The following fragment determines the name of the base or fallback font which is used to represent the specified Unicode character:

```
result = p.info_font(basefont, "fallbackfont", "unicode=U+03A3");
/* if result==basefont the base font was used, and no fallback font was required */
if (result == -1)
{
        /* character cannot be displayed with neither base font nor fallback fonts */
}
else
{
        idx = p.info_font(result, "fontname", "api");
        fontname = p.get_string(idx, "");
}
```

7 Text Output

7.1 Text Output Methods

PDFlib supports text output on several levels:

- Low-level text output with *PDF_show( )* and similar functions;
- Single-line formatted text output with *PDF_fit_textline( );* This function also supports text on a path.
- Multi-line text formatted output with Textflow *(PDF_fit_textflow( )* and related functions); The Textflow formatter can also wrap text inside or outside of vector-based shapes.
- Text in tables; the table formatter supports Textline and Textflow contents in table cells.

Low-level text output. Functions like *PDF_show( )* can be used to place text at a specific location on the page, without using any formatting aids. This is recommended only for applications with very basic text output requirements (e.g. convert plain text files to PDF), or for applications which already have full text placement information (e.g. a driver which converts a page in another format to PDF). The following fragment creates text output with low-level functions:

```
font = p.load_font("Helvetica", "unicode", "");
p.setfont(font, 12);
p.set_text_pos(50, 700);
p.show("Hello world!");
p.continue_text("(says Java)");
```

Formatted single-line text output with Textlines. *PDF_fit_textline( )* creates text output which consists of single lines and offers a variety of formatting features. However, the position of individual Textlines must be determined by the client application. The following fragment creates text output with a Textline. Since font, encoding, and fontsize can be specified as options, a preceding call to *PDF_load_font( )* is not required:

```
p.fit_textline(text, x, y, "fontname=Helvetica encoding=unicode fontsize=12");
```

See Section 9.1, »Placing and Fitting Textlines«, page 221, for more information.

Multi-line text output with Textflow. *PDF_fit_textflow( )* creates text output with an arbitrary number of lines and can distribute the text across multiple columns or pages. The Textflow formatter supports a wealth of formatting functions. The following fragment creates text output with Textflow:

```
tf = p.add_textflow(tf, text, optlist);
result = p.fit_textflow(tf, llx, lly, urx, ury, optlist);
p.delete_textflow(tf);
```

See Section 9.2, »Multi-Line Textflows«, page 231, for more information.

Text in tables. Textlines and Textflows can also be used to place text in table cells. See Section 9.3, »Table Formatting«, page 251, for more information.

7.2 Font Metrics and Text Variations

7.2.1 Font and Glyph Metrics

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

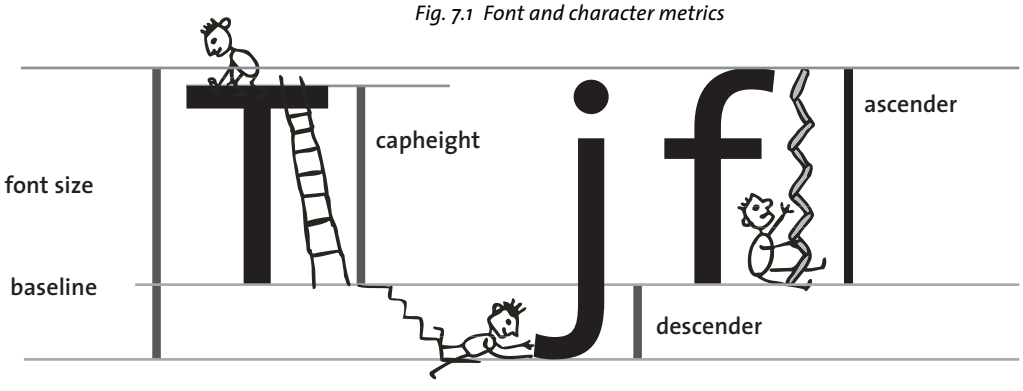
The *gaplen* property is only available in TrueType and OpenType fonts (it will be estimated for other font formats). The *gaplen* value is read from the font file, and specifies the difference between the recommended distance between baselines and the sum of ascender and descender.

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. 7.1 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 with PDFlib.*

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

7.2.2 Kerning

- Some character combinations can lead to unpleasant appearance. For example, two characters *V* next to each other can look like a *W*, and the distance between *T* and *e* must be reduced in order to avoid ugly white space. This compensation is referred to as kerning. Many fonts contain comprehensive kerning information which specifies spacing adjustments for critical letter combinations.

There are two PDFlib controls for the kerning behavior:

- By default, kerning information in a font will be read when loading the font. If kerning is not required the *readkerning* option can be set to *false* in *PDF_load_font()*.
- Kerning for text output must be enabled with the *kerning* text appearance option which is supported by the text output functions.

Temporarily disabling kerning may be useful, for example, for tabular figures when the kerning data contains pairs of figures, since kerned figures wouldn't line up in a table. Note that modern TrueType and OpenType fonts include special figures for this purpose which can be used with the *Tabular Figures* layout feature and the option *features={tnum}*.

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

Kerning is controlled by the font option *readkerning* and the text option *kerning*. By default kerning is enabled.

7.2.3 Text Variations

Simulated bold fonts. PDFlib supports a mechanism for creating artificial bold text for individual text strings via the *fakebold* option. This method simulates a bold font by stroking the glyph outlines; for Type 3 fonts the text will be placed multiply with differ-

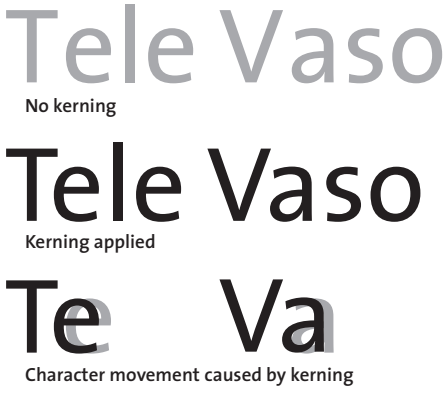


Fig. 7.2 Kerning

ent offsets. It is strongly recommended to use real bold font designs for emphasis. The *fakebold* option creates text output which is inferior to real bold text, and may inhibit text extraction.

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

Note *The* `fontstyle=bold[italic]` *font option can be used to force* `fakebold` *simulation for all text created with a particular font.*

Simulated italic fonts. The *italicangle* 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. 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* option can be used to easily simulate one. This feature may be especially useful for CJK fonts. Typical values for the *italicangle* option are in the range -12 to -15 degrees.

Note *PDFlib does not adjust the glyph width to the new bounding box of the slanted glyph. For example, when generated justified text the italicized glyphs may exceed beyond the fitbox.*

Shadow text. PDFlib can create a shadow effect which will generate multiple instances of text where each instance is placed at a slightly different location. Shadow text can be created with the *shadow* option of *PDF_fit_textline()* and *PDF_add/create_textflow()*. The color of the shadow, its position relative to the main text and graphics state options can be specified in suboptions.

Underline, overline, and strikeout text. PDFlib can be instructed to put lines below, above, or in the middle of text. The stroke width of the bar and its distance from the baseline are calculated based on the font's metrics information. In addition, the current values of the horizontal scaling factor and the text matrix are taken into account when calculating the width of the bar. The respective option names for *PDF_set_text_option()* 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* options can be used for fine-tuning.

The *strokecolor* option is used for drawing the bars. The current linecap option is ignored. The *decorationabove* option controls whether or not the line will be drawn on top of or below the text. Aesthetics alert: in most fonts underlining will touch descenders, and overlining will touch diacritical marks atop ascenders.

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

Text rendering modes. PDFlib supports several rendering modes which affect the appearance of text. This includes outline text and the ability to use text as a clipping path. Text can also be rendered invisibly which may be useful for placing text on scanned images in order to make the text accessible to searching and indexing, while at the same time assuring it will not be visible directly. The rendering modes are described in the *PDFlib API Reference*, and can be set with the *textrendering* option.

When stroking text, text state options such as *strokewidth* and *strokecolor* are applied to the glyph outline. The rendering mode has no effect on text displayed using a Type 3 font.

Cookbook *Full code samples can be found in the Cookbook topics* `text_output/text_as_clipping_path` and `text_output/invisible_text`.

Text color. Text is usually displayed in the color specified in the *fillcolor* option. However, if a rendering mode other than 0 has been selected, both *strokecolor* and *fillcolor* affect the text depending on the selected rendering mode.

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

7.3 OpenType Layout Features

Cookbook *Full code samples can be found in the Cookbook topics* `text_output/starter_opentype` *and* `font/opentype_feature_tester`.

7.3.1 Supported OpenType Layout Features

PDFlib supports enhanced text output according to additional information in some fonts. These font extensions are called OpenType layout features. For example, a font may contain a *liga* feature which includes the information that the *f*, *f*, and *i* glyphs can be combined to form a ligature. Other common examples are small caps in the *smcp* feature, i.e. uppercase characters which are smaller than the regular uppercase characters, and old-style figures in the *onum* feature with ascenders and descenders (as opposed to lining figures which are all placed on the baseline). Although ligatures are a very common OpenType feature, they are only one of many dozen possible features. An overview of the OpenType format and OpenType feature tables can be found at

`www.microsoft.com/typography/developers/opentype/default.htm`

PDFlib supports the following groups of OpenType features:

- OpenType features for Western typography listed in Table 7.1; these are controlled by the *features* option.
- OpenType features for Chinese, Japanese, and Korean text output listed in Table 7.7; these are also controlled by the *features* option, and are discussed in more detail in Section 7.5.4, »OpenType Layout Features for advanced CJK Text Output«, page 179.
- OpenType features for complex script shaping and vertical text output; these are automatically evaluated subject to the *shaping* and *script* options (see Section 7.4, »Complex Script Output«, page 170). The *vert* feature is controlled by the *vertical* font option.
- OpenType feature tables for kerning; however, PDFlib doesn't treat *kerning* as OpenType feature because kerning data may also be represented with other means than OpenType feature tables. Use the *readkerning* font option and the *kerning* text option instead to control kerning (see Section 7.2.2, »Kerning«, page 161).

More detailed descriptions of OpenType layout features can be found at

`www.microsoft.com/typography/otspec/featuretags.htm`

Identifying OpenType features. You can identify OpenType feature tables with the following tools:

- The FontLab font editor is a an application for creating and editing fonts. The free demo version *(www.fontlab.com)* displays OpenType features.
- DTL OTMaster Light *(www.fontmaster.nl)* is a free application for viewing and analyzing fonts, including their OpenType feature tables.
- PDFlib's *PDF_info_font( )* interface can also be used to query supported OpenType features (see »Querying OpenType features programmatically«, page 169).

Table 7.1 Supported OpenType features for Western typography (Table 7.7 lists OpenType features for CJK text)

key-word	***name***	***description***
`_none`	*all features disabled*	*Deactivate all OpenType features listed in Table 7.1 and Table 7.7.*
`afrc`	*alternative fractions*	*Replace figures separated by a slash with an alternative form.*
`c2pc`	*petite capitals from capitals*	*Turn capital characters into petite capitals.*
`c2sc`	*small capitals from capitals*	*Turn capital characters into small capitals.*
`calt`	*contextual alternates*	*Replace default glyphs with alternate forms which provide better joining behavior. Used in script typefaces which are designed to have some or all of their glyphs join.*
`case`	*case-sensitive forms*	*Shift accent marks upwards so that they work better with all-capital sequences or lining figures; also changes oldstyle figures to lining figures.*
`ccmp`	*glyph composition/ decomposition*	*To minimize the number of glyph alternates, it is sometimes desired to decompose a character into two glyphs, or to compose two characters into a single glyph for better glyph processing. This feature permits such composition/decomposition.*
`clig`	*contextual ligatures*	*Replace a sequence of glyphs with a single glyph which is preferred for typographic purposes. Unlike other ligature features,* `clig` *specifies the context in which the ligature is recommended. This capability is important in some script designs and for swash ligatures.*
`cswh`	*contextual swash*	*Replace the default glyph with a corresponding swash glyph in a specified context.*
`dlig`	*discretionary ligatures*	*Replace a sequence of glyphs with a single glyph which is preferred for typographic purposes.* `dlig` *covers ligatures which may be used for special effects, at the user's preference.*
`dnom`	*denominators*	*Replace figures which follow a slash with denominator figures.*
`falt`	*final glyph on line alternates*	*Replace line final glyphs with alternate forms specifically designed for this purpose (they would have less or more advance width as need may be), to help justification of text.*
`fina`	*final forms*	*Replace glyphs at the ends of words with alternate forms designed for this use. This is common in Latin connecting scripts, and required in various non-Latin scripts like Arabic.*
`frac`	*fractions*	*Replace figures separated by a slash with 'common' (diagonal) fractions.*
`hist`	*historical forms*	*Replace the default (current) forms with the historical alternates. Some letter forms were in common use in the past, but appear anachronistic today.*
`hlig`	*historical ligatures*	*Replace the default (current) ligatures with the historical alternates.*
`init`	*initial forms*	*Replace glyphs at the beginnings of words with alternate forms designed for this use. This is common in Latin connecting scripts, and required in various non-Latin scripts like Arabic.*
`isol`	*isolated forms*	*Replace the nominal form of glyphs with their isolated forms.*
`liga`	*standard ligatures*	*Replace a sequence of glyphs with a single glyph which is preferred for typographic purposes.* `liga` *covers ligatures which the designer/manufacturer judges should be used in normal conditions.*
`lnum`	*lining figures*	*Change figures from oldstyle to the default lining form.*
`locl`	*localized forms*	*Enable localized forms of glyphs to be substituted for default forms. This feature requires the* `script` *and* `language` *options.*
`medi`	*medial forms*	*Replace glyphs in the middle of words with alternate forms designed for this use. This is different from the default form, which is designed for stand-alone use. This is common in Latin connecting scripts, and required in various non-Latin scripts like Arabic.*

Table 7.1 Supported OpenType features for Western typography (Table 7.7 lists OpenType features for CJK text)

keyword	*name*	*description*
mgrk	*mathematical Greek*	*Replace standard typographic forms of Greek glyphs with corresponding forms commonly used in mathematical notation.*
numr	*numerators*	*Replace figures which precede a slash with numerator figures and replace the typographic slash with the fraction slash.*
onum	*oldstyle figures*	*Change figures from the default lining style to oldstyle form.*
ordn	*ordinals*	*Replace default alphabetic glyphs with the corresponding ordinal forms for use after figures; commonly also creates the Numero (U+2116) character.*
ornm	*ornaments*	*Replace the bullet character and ASCII characters with ornaments.*
pcap	*petite capitals*	*Turn lowercase characters into petite capitals, i.e. capital letters which are shorter than regular small caps.*
pnum	*proportional figures*	*Replace monospaced (tabular) figures with figures which have proportional widths.*
salt	*stylistic alternates*	*Replace the default forms with stylistic alternates. These alternates don't always fit into a clear category like swash or historical.*
sinf	*scientific inferiors*	*Replace lining or oldstyle figures with inferior figures (smaller glyphs), primarily for chemical or mathematical notation).*
smcp	*small capitals*	*Turn lowercase characters into small capitals.*
ss01 ... ss20	*stylistic set 1-20*	*In addition to, or instead of, stylistic alternatives of individual glyphs (see* salt *feature), some fonts may contain sets of stylistic variant glyphs corresponding to portions of the character set, e.g. multiple variants for lowercase letters in a Latin font.*
subs	*subscript*	*Replace a default glyph with a subscript glyph.*
sups	*superscript*	*Replace lining or oldstyle figures with superior figures (primarily for footnote indication), and replace lowercase letters with superior letters (primarily for abbreviated French titles)*
swsh	*swash*	*Replace default glyphs with corresponding swash glyphs.*
titl	*titling*	*Replace default glyphs with corresponding forms designed for titling.*
tnum	*tabular figures*	*Replace proportional figures with monospaced (tabular) figures.*
unic	*unicase*	*Map upper- and lowercase letters to a mixed set of lowercase and small capital forms, resulting in a single case alphabet.*
zero	*slashed zero*	*Replace the glyph for the figure zero with an alternative form which uses a diagonal slash through the counter.*

7.3.2 OpenType Layout Features with Textlines and Textflows

PDFlib supports OpenType layout features in the Textline and Textflow functions, but not in the simple text output functions *PDF_show( )* etc.

Requirements for OpenType layout features. A font for use with OpenType layout features must meet the following requirements:

- The font must be a TrueType *(*.ttf)*, OpenType *(*.otf)* or TrueType/OpenType Collection *(*.ttc)* font.
- The font file must contain a GSUB table with the OpenType feature(s) to be used.
- The font must be loaded with *encoding=unicode* or *glyphid*, or a Unicode CMap.

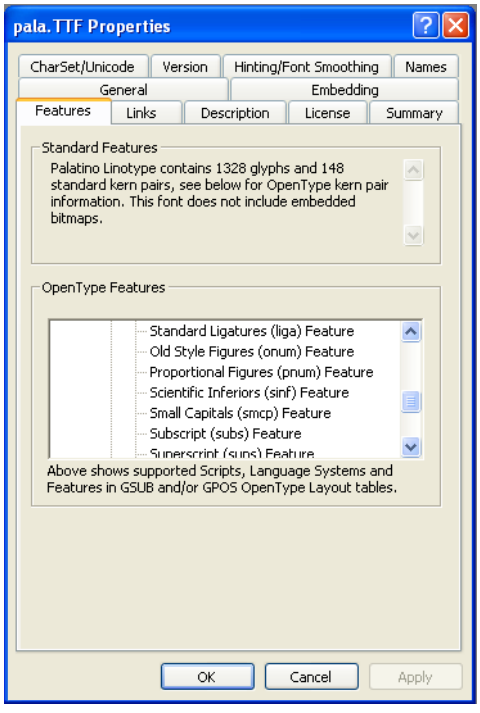


Fig. 7.3
Microsoft's font property extension displays the list of OpenType features in a font

- The *readfeatures* option of *PDF_load_font()* must not be set to *false*.

PDFlib supports OpenType features with GSUB table lookups. Except for kerning PDFlib does not support OpenType features based on the GPOS table.

Caveats. Note the following when working with OpenType features:

- OpenType features (options *features, script, language)* are only applied to glyphs within the same font, but not across glyphs from the base font and one or more fallback fonts if fallback fonts have been specified.
- Make sure to enable and disable features as you need them. Accidentally leaving OpenType features activated for all of the text may lead to unexpected results.

Enabling and disabling OpenType features. You can enable and disable OpenType features for pieces of text as required. Use the *features* text option to enable features by supplying their name, and disable them by prepending *no* to the feature name. For example, with inline option lists for Textflow feature control works as follows:

```
<features={liga}>ffi<features={noliga}
```

For Textlines you can enable OpenType features as follows:

```
p.fit_textline("ffi", x, y, "features={liga}");
```

OpenType features can also be enabled as Block properties for use with the PDFlib Personalization Server (PPS).

More than one feature can be applied to the same text, but the feature tables in the font must be prepared for this situation and must contain the corresponding feature

lookups in the proper order. For example, consider the word *office*, and the ligature *(liga)* and small cap *(smcp)* features. If both features are enabled (assuming the font contains corresponding feature entries) you'd expect the small cap feature to be applied, but not the ligature feature. If this is correctly implemented in the font tables, PDFlib will generate the expected output, i.e. small caps without any ligature.

Disabling ligatures with control characters. Some languages disallow the use of ligatures in certain situations. Typographic rules for German and other languages prohibit the use of ligatures across composition boundaries. For example, the *f+i* combination in the word *Schilfinsel* must not be replaced with a ligature since it spans the boundaries between two combined words.

As described above, you can enable and disable ligatures and other OpenType feature processing with the *features* option. Disabling ligatures via options can be cumbersome in exceptional cases like the one described above. In order to offer simple ligature control you can disable ligatures with control characters in the text, avoiding the need for enabling/disabling features with multiple options. Inserting the character *Zero-width non-joiner* (U+200C, *‌* see also Table 7.4) between the constituent characters will prevent them from being replaced by a ligature even if ligatures are enabled in the *features* option. For example, the following sequence will not create any *f+i* ligature:

```
<features={liga charref=true}>Schilf&zwnj;insel
```

Script- and language-specific OpenType layout features. OpenType features may apply in all situations, or can be implemented for a particular script or even a particular script and language combination. For this reason the *script* and *language* text options can optionally be supplied along with the *features* option. They will have a noticeable effect only if the feature is implemented in a script- or language-specific manner in the font.

As an example, the ligature for the *f* and *i* glyphs is not available in some fonts if the Turkish language is selected (since the ligated form of *i* could be confused with the dotless *i* which is very common in Turkish). Using such a font the following Textflow option will create a ligature since no script/language is specified:

```
<features={liga}>fi
```

However, the following Textflow option list will not create any ligature due to the Turkish language option:

```
<script=latn language=TRK features={liga}>fi
```

The *locl* feature explicitly selects language-specific character forms. The *liga* feature contains language-specific ligatures. Some examples for language-specific features:

```
Variant character for Serbian:
<features={locl} script=cyrl language=SRB charref>&#x0431;

Variant figures for Urdu:
<features={locl} script=arab language=URD charref>&#x0662;&#x0663;&#x0664;&#x0665;
```

See Section 7.4.2, »Script and Language«, page 172, for supported script and language keywords.

Combining OpenType features and shaping. Shaping for complex scripts (see Section 7.4, »Complex Script Output«, page 170) heavily relies on OpenType font features which will be selected automatically. However, for some fonts it may make sense to combine OpenType features selected automatically for shaping with OpenType features which have been selected by the client application. PDFlib will first apply user-selected OpenType features (option *features)* before applying shaping-related features automatically (options *shaping, script* and *language).*

Querying OpenType features programmatically. You can query OpenType features in a font programmatically with *PDF_info_font( )*. The following statement retrieves a space-separated list with all OpenType features which are available in the font and are supported by PDFlib:

```
result = (int) p.info_font(font, "featurelist", "");
if (result != -1)
{
        /* retrieve string containing space-separated feature list */
        featurelist = p.get_string(result, "");
}
else
{
        /* no supported features found */
}
```

Use the following statement to check whether PDFlib and the test font support a particular feature, e.g. ligatures *(liga):*

```
result = (int) p.info_font(font, "feature", "name=liga");
if (result == 1)
{
        /* feature supported by font and PDFlib */
}
```

7.4 Complex Script Output

Cookbook A full code sample can be found in the Cookbook topic `complex_scripts/starter_shaping`.

7.4.1 Complex Scripts

The Latin script basically places one character after the other in left-to-right order. Other writing systems have additional requirements for correct text output. We refer to such writing systems as complex scripts. PDFlib performs text processing for complex scripts for a variety of scripts including those listed in Table 7.2.

In this section we discuss shaping for complex scripts in more detail. Some writing systems (scripts) require additional processing:

- The Arabic and Hebrew scripts place text from right to left. Mixed text (e.g. Arabic with a Latin insert) contains both right-to-left and left-to-right segments. These segments must be reordered, which is referred to as the Bidi (bidirectional) problem.
- Some scripts, especially Arabic, use different character shapes depending on the position of the character (isolated, beginning/middle/end of a word).
- Mandatory ligatures replace sequences of characters.
- The position of glyphs must be adjusted horizontally and vertically.
- Indic scripts require reordering of some characters, i.e. characters may change their position in the text.
- Special word break and justification rules apply to some scripts.

Shaping. Scripts which require one or more of these processing steps are called complex scripts. The process of preparing incoming logical text for proper presentation is called shaping (this term also includes reordering and Bidi processing). The user always supplies text in unshaped form and in logical order, while PDFlib performs the necessary shaping before producing PDF output.

Complex script shaping can be enabled with the *shaping* text option, which in turn requires the *script* option and optionally allows the *language* option. The following option list enables Arabic shaping (and Bidi processing):

```
shaping script=arab
```

Caveats. Note the following when working with complex script shaping:

- PDFlib does not automatically set the *shaping* and *script* options, but expects them to be supplied by the user.
- Script-specific shaping (options *shaping, script, language)* is only applied to glyphs within the same font, but not across glyphs from different fonts. If fallback fonts are used, shaping is only applied within text runs in the same (master or fallback) font.
- Since shaping may reorder characters in the text, care must be taken regarding attribute changes within a word. For example, if you use inline options in Textflow to colorize the second character in a word – what should happen when shaping swaps the first and second characters? For this reason, formatting changes in shaped text should only be applied at word boundaries, but not within words.

Requirements for shaping. A font for use with complex script shaping must meet the following requirements in addition to containing glyphs for the target script:

- It must be a TrueType or OpenType font with GDEF, GSUB, and GPOS feature tables and correct Unicode mappings appropriate for the target script. As an alternative to

Table 7.2 Complex scripts and keywords for the `script` *option*

writing system	***script name***	***language/region (incomplete list)***	***script keyword***
unspecified script	–		`_none`
automatic script detection	–	*This keyword selects the script to which the majority of characters in the text belong, where* `_latn` *and* `_none` *are ignored.*	`_auto`
European Alphabetic	*Latin*	*many European and other languages*	`latn`
	Greek	*Greek*	`grek`
	Cyrillic	*Russian and many other Slavic languages*	`cyrl`
Middle Eastern	*Arabic*	*Arabic, Persian (Farsi), Urdu, Pashto and others*	`arab`
	Hebrew	*Hebrew, Yiddish and others*	`hebr`
	Syriac	*Syrian Orthodox, Maronite, Assyrian*	`syrc`
	Thaana	*Dhivehi/Maldives*	`thaa`
South Asian (India)	*Devanagari*	*Hindi and classical Sanskrit*	`deva`
	Bengali	*Bengali, Assamese*	`beng`
	Gurmukhi	*Punjabi*	`guru`
	Gujarati	*Gujarati*	`gujr`
	Oriya	*Oriya/Orissa*	`orya`
	Tamil	*Tamil/Tamil Nadu, Sri Lanka*	`taml`
	Telugu	*Telugu/Andrha Pradesh*	`telu`
	Kannada	*Kannada/Karnataka*	`knda`
	Malayalam	*Malayalam/Kerala*	`mlym`
Southeast Asian	*Thai*	*Thai*	`thai`
	Lao	*Lao*	`»lao «`[1]
	Khmer	*Khmer (Cambodian)*	`khmr`
East Asian	*Han*	*Chinese, Japanese, Korean*	`hani`
	Hiragana	*Japanese*	`hira`
	Katakana	*Japanese*	`kana`
	Hangul	*Korean*	`hang`
others	*Other four-character codes according to the OpenType specification also work, but are not supported. The full list can be found at the following location:* `www.microsoft.com/typography/developers/OpenType/scripttags.aspx`		

1. Note the trailing space character.

the OpenType tables, for the Arabic and Hebrew scripts, the font may contain glyphs for the Unicode presentation forms (e.g. Arabic Apple fonts are constructed this way). In this case internal tables will be used for the shaping process. For Thai text the font must contain contextual forms according to Microsoft, Apple, or Monotype Worldtype (e.g. used in some IBM products) conventions for Thai.

- The font must be loaded with *encoding=unicode* or *glyphid*.
- The *vertical* option of *PDF_load_font()* must not be used, and the *readshaping* option must not be set to *false*.

7.4.2 Script and Language

Script and language settings play a role in the functional aspects listed below. They can be controlled with the following options:

- The *script* text option identifies the target script (writing system). It supports the four-letter keywords listed in Table 7.2. Examples:

```
script=latn
script=cyrl
script=arab
script=hebr
script=deva
script={lao }
```

  With *script=_auto* PDFlib automatically assigns that script to which the majority of characters in the text belong. Since Latin text doesn't require shaping it will not be counted when determining the script automatically. You can query the scripts used for some text with the *scriptlist* keyword of *PDF_info_textline()*.
- The *language* option specifies the natural language in which the text is written. It supports the three-character keywords listed in Table 7.3. Examples:

```
language=ARA
language=URD
language=ZHS
language=HIN
```

Complex script processing. Complex script processing (option *shaping)* requires the *script* option. The *language* option can additionally be supplied. It controls language-specific aspects of shaping, e.g. different figures for Arabic vs. Urdu. However, only few fonts contain language-specific script shaping tables, so in most cases specifying the *script* option will be sufficient, and shaping cannot be improved with the *language* option.

OpenType layout features. Fonts can implement OpenType layout features in a language-specific manner (see »Script- and language-specific OpenType layout features«, page 168). While a few features may differ in behavior subject to the *script* and *language* options but can also be used without these options (e.g. *liga)*, the *locl* feature only makes sense in combination with the *script* and *language* options.

Note While advanced line breaking for Textflow (see Section 9.2.10, »Advanced script-specific Line Breaking«, page 246) also applies language-specific processing, it is not controlled by the `language` *option, but by the* `locale` *option which identifies not only languages, but also countries and regions.*

Table 7.3 Keywords for the language *option*

keyword	*language*	*keyword*	*language*	*keyword*	*language*
_none	*unspecified language*	FIN	*Finnish*	NEP	*Nepali*
AFK	*Afrikaans*	FRA	*French*	ORI	*Oriya*
SQI	*Albanian*	GAE	*Gaelic*	PAS	*Pashto*
ARA	*Arabic*	DEU	*German*	PLK	*Polish*
HYE	*Armenian*	ELL	*Greek*	PTG	*Portuguese*
ASM	*Assamese*	GUJ	*Gujarati*	ROM	*Romanian*
EUQ	*Basque*	HAU	*Hausa*	RUS	*Russian*
BEL	*Belarussian*	IWR	*Hebrew*	SAN	*Sanskrit*
BEN	*Bengali*	HIN	*Hindi*	SRB	*Serbian*
BGR	*Bulgarian*	HUN	*Hungarian*	SND	*Sindhi*
CAT	*Catalan*	IND	*Indonesian*	SNH	*Sinhalese*
CHE	*Chechen*	ITA	*Italian*	SKY	*Slovak*
ZHP	*Chinese phonetic*	JAN	*Japanese*	SLV	*Slovenian*
ZHS	*Chinese simplified*	KAN	*Kannada*	ESP	*Spanish*
ZHT	*Chinese traditional*	KSH	*Kashmiri*	SVE	*Swedish*
COP	*Coptic*	KHM	*Khmer*	SYR	*Syriac*
HRV	*Croatian*	KOK	*Konkani*	TAM	*Tamil*
CSY	*Czech*	KOR	*Korean*	TEL	*Telugu*
DAN	*Danish*	MLR	*Malayalam reformed*	THA	*Thai*
NLD	*Dutch*	MAL	*Malayalam traditional*	TIB	*Tibetan*
DZN	*Dzongkha*	MTS	*Maltese*	TRK	*Turkish*[1]
ENG	*English*	MNI	*Manipuri*	URD	*Urdu*
ETI	*Estonian*	MAR	*Marathi*	WEL	*Welsh*
FAR	*Farsi*	MNG	*Mongolian*	JII	*Yiddish*

1. Some fonts wrongly use TUR for Turkish; PDFlib treats this tag as equivalent to TRK.

7.4.3 Complex Script Shaping

The shaping process selects appropriate glyph forms depending on whether a character is located at the start, middle, or end of a word, or in a standalone position. Shaping is a crucial component of Arabic and Hindi text formatting. Shaping may also replace a sequence of two or more characters with a suitable ligature. Since the shaping process determines the appropriate character forms automatically, explicit ligatures and Unicode presentation forms (e.g. Arabic Presentation Forms A starting at U+FB50) must not be used as input characters.

Since complex scripts require multiple different glyph forms per character and additional rules for selecting and placing these glyphs, shaping for complex scripts does not

work with all kinds of fonts, but requires suitable fonts which contain the necessary information. Shaping works for TrueType and OpenType fonts which contain the required feature tables (see »Requirements for shaping«, page 170, for detailed requirements).

Shaping can only be done for characters in the same font because the shaping information is specific to a particular font. As it doesn't make sense, for example, to form ligatures across different fonts, complex script shaping cannot be applied to a word which contains characters from different fonts.

Override shaping behavior. In some cases users may want to override the default shaping behavior. PDFlib supports several Unicode formatting characters for this purpose. For convenience, these formatting characters can also be specified with entity names (see Table 7.4).

Table 7.4 Unicode control characters for overriding the default shaping behavior

formatting character	*entity name*	*Unicode name*	*function*
U+200C	*ZWNJ*	*ZERO WIDTH NON-JOINER*	*prevent the two adjacent characters from forming a cursive connection*
U+200D	*ZWJ*	*ZERO WIDTH JOINER*	*force the two adjacent characters to form a cursive connection*

7.4.4 Bidirectional Formatting

Cookbook A full code sample can be found in the Cookbook topic `complex_scripts/bidi_formatting`.

For right-to-left text (especially Arabic and Hebrew, but also some other scripts) it is very common to have nested sequences of left-to-right Latin text, e.g. an address or a quote in another language. These mixed sequences of text require bidirectional (Bidi) formatting. Since numerals are always written from left to right, the Bidi problem affects even text which is completely written in Arabic or Hebrew. PDFlib implements bidirectional text reordering according to the Unicode Bidi algorithm as specified in Unicode Standard Annex #9[1]. Bidi processing does not have to be enabled with an option, but will automatically be applied as part of the shaping process if text in a right-to-left script with an appropriate *script* option is encountered.

Note Bidi processing is not supported for multi-line Textflows, but only for Textlines (i.e. single-line text output).

Overriding the Bidi algorithm. While automatic Bidi processing will provide proper results in common cases, there are situations which require explicit user control. PDFlib supports several directional formatting codes for this purpose. For convenience, these formatting characters can also be specified with entity names (see Table 7.5). The bidirectional formatting codes are useful to override the default Bidi algorithm in the following situations:

- a right-to-left paragraph begins with left-to-right characters;
- there are nested segments with mixed text;

1. See www.unicode.org/unicode/reports/tr9/

- there are weak characters, e.g. punctuation, at the boundary between left-to-right and right-to-left text;
- part numbers and similar entities containing mixed text.

Table 7.5 Directional formatting codes for overriding the bidirectional algorithm

formatting code	entity name	Unicode name	function
U+202A	LRE	LEFT-TO-RIGHT EMBEDDING (LRE)	start an embedded left-to-right sequence
U+202B	RLE	RIGHT-TO-LEFT EMBEDDING (RLE)	start an embedded right-to-left sequence
U+200E	LRM	LEFT-TO-RIGHT MARK (LRM)	left-to-right zero-width character
U+200F	RLM	RIGHT-TO-LEFT MARK (RLM)	right-to-left zero-width character
U+202D	LRO	LEFT-TO-RIGHT OVERRIDE (LRO)	force characters to be treated as strong left-to-right characters
U+202E	RLO	RIGHT-TO-LEFT OVERRIDE (RLO)	force characters to be treated as strong right-to-left characters
U+202C	PDF	POP DIRECTIONAL FORMATTING (PDF)	restore the bidirectional state to what it was before the last LRE, RLE, RLO, or LRO

Options for improved right-to-left document handling. The default settings of various formatting options and Acrobat behavior are targeted at left-to-right text output. Use the following options for right-to-left text formatting and document display:

- Place a Textline right-aligned with the following fitting option:

```
position={right center}
```

- Create a leader between the text and the left border:

```
leader={alignment=left text=.}
```

- Use the following option of *PDF_begin/end_document( )* to activate better right-to-left document and page display in Acrobat:

```
viewerpreferences={direction=r2l}
```

Dealing with Bidi text in your code. The following may also be useful when dealing with bidirectional text:

- You can use the *startx/starty* and *endx/endy* keywords of *PDF_info_textline( )* to determine the coordinates of the logical start and end characters, respectively.
- You can use the *writingdirx* keyword of *PDF_info_textline( )* to determine the dominant writing direction of text. This direction will be inferred from the initial characters of the text or from directional formatting codes according to Table 7.5 (if present in the text).
- You can use the *auto* keyword for the *position* option of *PDF_info_textline( )* to automatically align Arabic or Hebrew text at the right border and Latin text at the left border. For example, the following Textline option list aligns right- or left-aligns the text on the baseline:

```
boxsize={width 0} position={auto bottom}
```

7.4.5 Arabic Text Formatting

Cookbook *A full code sample can be found in the Cookbook topic* `complex_scripts/arabic_formatting`.

In addition to Bidirectional formatting and text shaping as discussed above there are several other formatting aspects related to generating text output in the Arabic script.

Arabic ligatures. The Arabic script makes extensive use of ligatures. Many Arabic fonts contain two kinds of ligatures which are treated differently in PDFlib:

- Required ligatures *(rlig* feature) must always be applied, e.g. the Lam-Alef ligature and variants thereof. Required ligatures are used if the *shaping* option is enabled with *script=arab*.
- Optional Arabic ligatures *(liga* and *dlig* features) are not used automatically, but can be enabled like other user-controlled OpenType features, i.e. *features={liga}*. Optional Arabic ligatures are applied after complex script processing and shaping.

Avoiding ligatures. In some cases joining adjacent characters is not desired, e.g. for certain abbreviations. In such cases you can use the formatting characters listed in Table 7.4 to force or prevent characters from being joined. For example, the zero-width non-joiner character in the following example prevents the characters from being joined in order to form a proper abbreviation:

```
&#x0623;&#x064A;&ZWNJ;&#x0628;&#x064A;&ZWNJ;&#x0625;&#x0645;
```

Tatweel formatting for Arabic text. You can stretch Arabic words by inserting one or more instances of the tatweel character U+0640 (also called kashida). While PDFlib does not automatically justify text by inserting tatweel characters, you can insert this character in the input text to stretch words.

Adding Latin characters to an Arabic font. Some Arabic fonts do not contain any glyphs for Latin characters, e.g. Google's Noto fonts. In this situation you can use the *fallbackfonts* option to merge Latin characters into an Arabic font. PDFlib automatically switches between both fonts depending on the Latin or Arabic text input, i.e. you don't have to switch fonts in your application but can supply mixed Latin/Arabic text with a single font specification.

You can use the following font loading option list for the *fallbackfonts* option to add Latin characters from the NotoSerif-Regular font to the NotoNaskhArabic-Regular font:

```
fontname=NotoNaskhArabic-Regular encoding=unicode
fallbackfonts={ {fontname=NotoSerif-Regular encoding=unicode} }
```

7.5 Chinese, Japanese, and Korean Text Output

7.5.1 Using TrueType and OpenType CJK Fonts

PDFlib supports CJK fonts in the TrueType, TrueType/OpenType Collection (TTC/OTC) and OpenType formats. CJK fonts are processed as follows:

- If the *embedding* option is *true*, the font is converted to a CID font and embedded in the PDF output.
- CJK host font names on Windows can be supplied to *PDF_load_font()* as UTF-8 with BOM or UTF-16. Non-Latin host font names are not supported on macOS.

The following example uses the *ArialUnicodeMS* font to display Chinese text. The font must either be installed on the system or must be configured according to Section 6.4.4, »Searching for Fonts«, page 140):

```
font = p.load_font("Arial Unicode MS", "unicode", "");
if (font == -1) { ... }

p.fit_textline("\u4e00\u500b\u4eba", x, y, "fontsize=24");
```

Accessing individual fonts in a TrueType Collection. TTC/OTC 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/OTC file you can numerically address each font by appending a colon character and the number of the font within the TTC/OTC file (starting with 0). If the index is 0 it can be omitted. For example, the TTC file *msgothic.ttc* contains multiple fonts which can be addressed as follows in *PDF_load_font()* (each line contains equivalent font names):

```
msgothic:0              MS Gothic               msgothic:
msgothic:1              MS PGothic
msgothic:2              MS UI Gothic
```

Note that *msgothic* (without any suffix) will not work as a font name since it does not uniquely identify a font. Font name aliases (see »Sources of Font Data«, page 140) can be used in combination with TTC/OTC indexing. If a font with the specified index cannot be found, the function call will fail.

It is only required to configure the TTC/OTC font file once; all indexed fonts in the TTC/OTC file will be found automatically. The following code is sufficient to configure all indexed fonts in *msgothic.ttc* (see Section 6.4.4, »Searching for Fonts«, page 140):

```
p.set_option("FontOutline={msgothic=msgothic.ttc}");
```

7.5.2 Horizontal and Vertical Writing Mode

PDFlib supports both horizontal and vertical writing modes. Vertical writing mode can be requested in different ways:

- TrueType and OpenType Fonts with encodings other than a CMap can be used for vertical writing mode by supplying the *vertical* font option.
- Font names starting with an '@' character are always processed in vertical mode.
- For CJK 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.

By default all glyphs have the same height in vertical writing mode. However, TrueType and OpenType fonts may contain proportional metrics for vertical writing mode. PDFlib can be instructed to use such proportional vertical metrics by setting the font option *readverticalmetrics* (with the default *false)*.

Note Character spacing must be negative to spread characters apart in vertical writing mode.

OpenType fonts may contain the OpenType layout features listed in Table 7.6. They replace default glyph forms with variants adjusted for vertical writing. The *vkna* feature can be controlled with the *features* text option, while *vrt2/vert* are automatically enabled in vertical writing mode.

Table 7.6 OpenType layout features for vertical text

keyword	***name***	***description***
vert[1]	*vertical writing*	*Replace default forms with variants adjusted for vertical writing.*
vkna	*vertical Kana alternates*	*Replace standard Kana with forms that have been specially designed for vertical writing.*
vrt2[1]	*vertical alternates and rotation*	*Replace some fixed-width (half-, third- or quarter-width) or proportional-width glyphs (mostly Latin or Katakana) with forms suitable for vertical writing, i.e. rotated 90° clockwise. If this feature is present it disables the* vert *feature which is a subset of* vrt2.

1. Automatically enabled for fonts in vertical writing mode

7.5.3 EUDC and SING Fonts for Gaiji Characters

PDFlib supports Windows EUDC (end-user defined characters, **.tte)* and SING fonts *(*.gai)* which can be used to access custom Gaiji characters for CJK text. Most conveniently fonts with custom characters are integrated into other fonts with the fallback font mechanism. Gaiji characters will commonly be provided in EUDC or SING fonts.

Using fallback fonts for Gaiji characters. Typically, Gaiji characters will be pulled from Windows EUDC fonts or SING glyphlets, but the *fallbackfonts* option accepts any kind of font. Therefore this approach is not limited to Gaiji characters, but can be used for any kind of symbol (e.g. a company logo in a separate font). You can use the following font loading option list for the *fallbackfonts* option to add a user-defined (gaiji) character from an EUDC font to the loaded font:

```
fallbackfonts={
 {fontname=EUDC encoding=unicode forcechars=U+E000 fontsize=140% textrise=-20%}
}
```

Once a base font has been loaded with this fallback font configuration, the EUDC character can be used within the text without any need to change the font.

With SING fonts the Unicode value doesn't have to be supplied since it will automatically be determined by PDFlib:

```
fallbackfonts={
 {fontname=PDFlibWing encoding=unicode forcechars=gaiji}
}
```

Preserving PUA values for Gaiji characters. In some cases, e.g. with Windows EUDC fonts (see below), Gaiji characters are mapped to Unicode values in the Private Use Area (PUA) by the font. By default, PDFlib replaces PUA values with U+FFFD (Unicode replacement character) in the ToUnicode CMap. As a result such characters cannot be extracted correctly from the generated PDF.

This behavior can be changed with the *preservepua* font loading option. If it is set to *true* for a font, Gaiji characters with PUA values will retain their Unicode values, i.e. the Gaiji can be extracted correctly from the generated PDF.

Preparing EUDC fonts. You can use the EUDC editor available in Windows to create custom characters for use with PDFlib. Proceed as follows:

- Use the *eudcedit.exe* to create one or more custom characters at the desired Unicode position(s).
- Locate the *EUDC.TTE* file in the directory *\Windows\fonts* and copy it to some other directory. Since this file is invisible in Windows Explorer use the *dir* and *copy* commands in a DOS box to find the file. Now configure the font for use with PDFlib, using one of the methods discussed in (see Section 6.4.4, »Searching for Fonts«, page 140):

  ```
  p.set_option("FontOutline={EUDC=EUDC.TTE}");
  p.set_option("SearchPath={{...directory name...}}");
  ```

  or place *EUDC.TTE* in the current directory.
  As an alternative to this explicit font file configuration you can use the following function call to configure the font file directly from the Windows directory. This way you will always access the current EUDC font used in Windows:

  ```
  p.set_option("FontOutline={EUDC=C:\Windows\fonts\EUDC.TTE}");
  ```

- Integrate the EUDC font into any base font using the *fallbackfonts* option as described above. If you want to access the font directly, use the following call to load the font in PDFlib:

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

  as usual and supply the Unicode value(s) chosen in the first step to output the characters.

7.5.4 OpenType Layout Features for advanced CJK Text Output

As detailed in Section 7.3, »OpenType Layout Features«, page 164, PDFlib supports advanced typographic layout tables in OpenType and TrueType fonts. For example, OpenType features can be used to select alternative forms of the Latin glyphs with proportional widths or half widths, or to select alternate character forms. Table 7.7 lists OpenType features for CJK text (additional OpenType layout features for general use are listed in Table 7.1 and features for vertical writing mode in Table 7.6).

Table 7.7 Supported OpenType layout features for Chinese, Japanese, and Korean text

keyword	*name*	*description*
expt	*expert forms*	*Like the JIS78 forms this feature replaces standard Japanese forms with corresponding forms preferred by typographers.*

Table 7.7 Supported OpenType layout features for Chinese, Japanese, and Korean text

keyword	name	description
fwid	*full widths*	*Replace glyphs set on other widths with glyphs set on full (usually em) widths. This may include Latin characters and various symbols.*
hkna	*horizontal Kana alternates*	*Replace standard Kana with forms that have been specially designed for only horizontal writing.*
hngl	*Hangul*	*(Deprecated per ISO 14496-22:2015/Amd.2:2017) Replace hanja (Chinese-style) Korean characters with the corresponding Hangul (syllabic) characters.*
hojo	*Hojo Kanji forms (JIS X 0212-1990)*	*Access the JIS X 0212-1990 glyphs (also called »Hojo Kanji«) if the JIS X 0213:2004 form is encoded as default.*
hwid	*half widths*	*Replace glyphs on proportional widths, or fixed widths other than half an em, with glyphs on half-em widths.*
ital	*italics*	*Replace the Roman glyphs with the corresponding Italic glyphs.*
jp04	*JIS2004 forms*	*(Subset of the* nlck *feature) Access the JIS X 0213:2004 glyphs.*
jp78	*JIS78 forms*	*Replace default (JIS90) Japanese glyphs with the corresponding forms from JIS C 6226-1978 (JIS78).*
jp83	*JIS83 forms*	*Replace default (JIS90) Japanese glyphs with the corresponding forms from JIS X 0208-1983 (JIS83).*
jp90	*JIS90 forms*	*Replace Japanese glyphs from JIS78 or JIS83 with the corresponding forms from JIS X 0208-1990 (JIS90).*
locl	*localized forms*	*Enable localized forms of glyphs to be substituted for default forms. This feature requires the* script *and* language *options.*
nalt	*alternate annotation forms*	*Replace default glyphs with various notational forms (e.g. glyphs placed in open or solid circles, squares, parentheses, diamonds or rounded boxes).*
nlck	*NLC Kanji forms*	*Access the new glyph shapes defined in 2000 by the National Language Council (NLC) of Japan for a number of JIS characters.*
pkna	*proportional Kana*	*Replace glyphs, Kana and Kana-related, set on uniform widths (half or full-width) with proportional glyphs.*
pwid	*proportional widths*	*Replace glyphs set on uniform widths (typically full or half-em) with proportionally spaced glyphs.*
qwid	*quarter widths*	*Replace glyphs on other widths with glyphs set on widths of one quarter of an em.*
ruby	*Ruby notation forms*	*Replace default Kana glyphs with smaller glyphs designed for use as (usually superscripted) Ruby.*
smpl	*simplified forms*	*Replace traditional Chinese or Japanese forms with the corresponding simplified forms.*
tnam	*traditional name forms*	*Replace simplified Japanese Kanji forms with the corresponding traditional forms. This is equivalent to the* trad *feature, but limited to the traditional forms considered proper for use in personal names.*
trad	*traditional forms*	*Replace simplified Chinese Hanzi or Japanese Kanji forms with the corresponding traditional forms.*
twid	*third widths*	*Replace glyphs on other widths with glyphs set on widths of one third of an em.*

7.5.5 Unicode Variation Selectors and Variation Sequences

Unicode characters can be represented by a wide variety of glyphs. Generally, such visual differences are realized by using suitable fonts (e.g. regular vs. italic). In some situations the choice of glyph is semantically relevant and must be made explicit even in plain text without any font-related formatting information. Unicode offers the mechanism of variation selectors for this purpose.

Variation sequences. A variation sequence consists of a base Unicode character followed by a variation selector. The sequence is called a *variant* of the base character. The Unicode standard comprises two kinds of sequences:

- Standardized variation sequences are defined in the file *StandardizedVariants.txt*[1] in the Unicode Character Database. They use one of 16 variation selectors in the range *U+FE00 - U+FE0F*. Standardized variation sequences are used for selecting alternate mathematical glyphs, Emoji variants, and Mongolian text.
- Ideographic Variation Sequences (IVS) are defined by the registration process according to Unicode Technical Standard #37, »Unicode Ideographic Variation Database«, and are listed in the Ideographic Variation Database[2]. An IVS consists of a unified ideographic character as base character and one of 240 variation selectors in the range *U+E0100 - U+E01EF*. IVSes are mainly used for selecting appropriate glyphs for person and place names.

If the variation selector for the base character cannot be honored, e.g. because the font does not contain the required glyph, it will be ignored.

Creating variant glyphs with PDFlib. Suitable glyphs for Unicode Variation Sequences (UVS) must be provided by the font. Currently OpenType is the only font format capable of storing UVSes (using a format 14 cmap table). PDFlib interprets the UVS table in an OpenType font unless this has been disabled with the *readselectors* font option. Since a font is available only for content strings, but not for hypertext and name strings, variation selectors will be ignored for these string types.

Assuming you know that a font contains the required glyphs, working with variation sequences is as simple as providing the sequence to PDFlib's text output functions. The following code fragments print the default glyph of a Unicode base character plus a variant chosen by a selector.

```
p.fit_textline("&#x2268; &#x2268;&#xFE00;", 50, 700,
        "fontname={Cambria Math} encoding=unicode fontsize=24 charref=true");

p.fit_textline("&#x3402;&#xE0100; &#x3402;&#xE0101;", 50, 650,
        "fontname={KozMinPr6N-Regular} encoding=unicode fontsize=24 charref=true");
```

The resulting output is shown in Figure 7.4; note the differences within the glyph pairs.

Note *Variation sequences are not supported in the* `forcechars` *suboption of the* `fallbackfonts` *option.*

1. See www.unicode.org/Public/UNIDATA/StandardizedVariants.html
2. See www.unicode.org/ivd

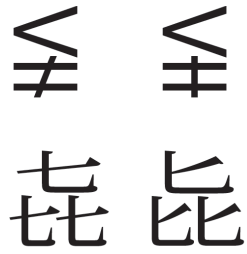


Fig. 7.4 Default glyph and variant glyph

Querying variation selectors in a font. Using *PDF_info_font( )* you can check whether a font contains variation selectors at all. The *selector* keyword along with the *index* option can be used to retrieve a list of all variation selectors available in the font:

```
for (i = 0; i < 256; i++)
{
        selectors[i] = (int) p.info_font(font, "selector", "index=" + i);
        if (selectors[i] == -1)
        {
            selectorcount = i;
            break;
        }
}
```

The following code fragment can be used to check whether the font contains a variant glyph for a particular sequence:

```
if (p.info_font(font, "unicode", "unicode=" + uv + "selector=" + s) == -1)
{
        /* no variant glyph available in the font for this sequence */
}
```

This query is only intended to check whether a variant is available. The resulting Unicode (if a variant is available) itself is unlikely to be useful because PDFlib assigns PUA Unicode values to variants.

7.5.6 Standard CJK Fonts

Note The concept of standard CJK fonts is deprecated; use externally configured font files with or without embedding instead.

Acrobat supports various standard fonts for CJK text. These fonts are supplied with the Acrobat installation and don't have to be embedded in the PDF file. The standard CJK fonts support horizontal and vertical writing modes. The names of the standard CJK fonts are listed in Table 7.8 along with applicable CMaps (see Section 5.5, »Chinese, Japanese, and Korean CMaps«, page 116, for more details on CJK CMaps).

Keeping native CJK legacy codes. If *keepnative=true*, native legacy character codes (e.g. Shift-JIS) according to the selected CMap are written to the PDF output; otherwise the text is converted to Unicode. The advantage of *keepnative=true* is that such fonts can be used for form fields without embedding (see description of the *keepnative* font loading option for in the PDFlib API Reference). If *keepnative=false* legacy code sequences are converted to CID values which are written to the PDF output. The advantage is that OpenType features and the Textflow formatter can be used.

Table 7.8 Acrobat's standard fonts and CMaps (encodings) for Japanese, Chinese, and Korean text

locale	font name	supported CMaps (encodings)
Simplified Chinese	AdobeSongStd-Light[2]	GB-EUC-H, GB-EUC-V, GBpc-EUC-H, GBpc-EUC-V, GBK-EUC-H, GBK-EUC-V, GBKp-EUC-H, GBKp-EUC-V, GBK2K-H, GBK2K-V, UniGB-UTF16-H[1], UniGB-UTF16-V[1]
Traditional Chinese	AdobeMingStd-Light[2]	B5pc-H, B5pc-V, HKscs-B5-H, HKscs-B5-V, ETen-B5-H, ETen-B5-V, ETenms-B5-H, ETenms-B5-V, CNS-EUC-H, CNS-EUC-V, UniCNS-UTF16-H[1], UniCNS-UTF16-V[1]
Japanese	KozMinPro-Regular-Acro KozGoPro-Medium[2] KozMinProVI-Regular[2]	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-UTF16-H[1], UniJIS-UTF16-V[1]
Korean	AdobeMyungjoStd-Medium[2]	KSC-EUC-H, KSC-EUC-V, KSCms-UHC-H, KSCms-UHC-V, KSCms-UHC-HW-H, KSCms-UHC-HW-V, KSCpc-EUC-H, UniKS-UTF16-H[1], UniKS-UTF16-V[1]

1. Only available when generating PDF 1.5 or above
2. Only available when generating PDF 1.6 or above

8 Importing Images, SVG Graphics and PDF Pages

8.1 Raster Images

8.1.1 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 analyzes the image characteristics and copies the pixel data to the PDF output. *PDF_load_image()* 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 options:

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

p.fit_image(image, 0.0, 0.0, "");
p.close_image(image);
```

The last parameter of *PDF_fit_image()* function is an option list which supports a variety of options for positioning, scaling, and rotating the image. Details regarding these options are discussed in Section 8.4, »Placing Images, Graphics, and imported PDF Pages«, page 213.

If an image file cannot be imported successfully *PDF_load_image()* returns an error code. If you need to know more details about the image failure, call *PDF_get_errmsg()* to retrieve a detailed error message.

Cookbook *Code samples for image handling can be found in the* `images` *category of the PDFlib Cookbook.*

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

Scaling and dpi calculations. PDFlib never changes the number of pixels in an imported image. Scaling either blows up or shrinks image pixels, but doesn't do any downsampling. A scaling factor of 1 results in a pixel size of 1 unit in user coordinates. In other words, the image is 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).

Cookbook *A full code sample can be found in the Cookbook topic* `images/image_dimensions`. *It shows how to get the dimensions of an image and how to place it with various sizes.*

Color space of imported images. Except for adding or removing ICC profiles and applying spot or DeviceN color according to the options provided in *PDF_load_image()*, PDFlib generally preserves the native color space of an imported image. However, this is not possible for certain rare combinations.

PDFlib does not perform any conversion between RGB and CMYK. If such a conversion is required it must be applied to the image data before loading the image in PDFlib.

Multi-page images. PDFlib supports GIF, TIFF and JBIG2 images with more than one image, also known as multi-page image files. In order to use multi-page images use the *page* option in *PDF_load_image():*

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

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

Cookbook *A full code sample for converting all images in a multi-image TIFF file to a multi-page PDF file can be found in the Cookbook topic* `images/multi_page_tiff`.

Inline images. As opposed to reusable images, which are written to the PDF output as image XObjects, inline images are written directly into the respective content stream (page, pattern, template, or glyph description). This results in some space savings, but should only be used for small amounts of image data (up to 4 KB). The primary use of inline images is for bitmap glyph descriptions in Type 3 fonts; inline images are not recommended for other situations.

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

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

Inline images are only supported for *imagetype=ccitt, jpeg,* and *raw*. For other image types the inline option is silently ignored.

XMP metadata in images. Image files may contain XMP metadata. By default PDFlib ignores image metadata for images in the TIFF, JPEG, and JPEG 2000 image formats to reduce the output file size. However, the XMP metadata can be attached to the generated image in the output PDF document with the following option of *PDF_load_image():*

```
metadata={keepxmp=true}
```

8.1.2 Supported Image File Formats

PNG images. PDFlib supports all flavors of PNG images (ISO 15948). If a PNG image contains transparency information, the transparency is retained in the generated PDF (see Section 8.1.4, »Image Transparency«, page 191).

If a PNG image contains an sRGB chunk, the sRGB ICC profile is attached to the image unless the *honoriccprofile* option is *false* or another ICC profile has been assigned to the image with the *iccprofile* option. The rendering intent in the sRGB chunk will be used unless the *renderingintent* option has been supplied.

JPEG images. PDFlib supports the following flavors of JPEG images (ISO 10918-1):

- Grayscale, RGB (usually encoded as YCbCr, but direct RGB is also supported), and CMYK color
- Baseline and progressive JPEG compression

These conditions cover all JPEG images in practical use. The restrictions above imply that images using some uncommon JPEG features cannot be imported, in particular arithmetic coding, lossless compression, bit depths other than 8 bits per component as well as non-standard features such as SmartScale and reversible RGB color transform. Note that the similar formats JPEG-LS (ISO 14495), sometimes called »lossless JPEG« and JPEG-XR (ISO 29199-2), formerly called HD Photo, are not supported.

JPEG images can be packaged in several file formats. PDFlib supports all common JPEG file formats and features:

- JFIF 1, which is generated by a wide variety of imaging applications.
- JPEG files written by Adobe Photoshop and other Adobe applications.
- PDFlib reads clipping paths from JPEG images created with Adobe Photoshop.
- PDFlib honors embedded ICC profiles in JPEG images unless the *honoriccprofile* option is set to *false*.
- If a JPEG image contains an Exif marker the color space information in the Exif marker is interpreted. If it indicates sRGB color space the sRGB ICC profile is attached to the image (unless the image contains an explicit embedded ICC profile, the *honoriccprofile* option is *false* or another ICC profile has been assigned to the image with the *iccprofile* option).
- The orientation entry in an Exif marker which specifies the desired image orientation is honored. It can be ignored (as many applications do) with the *ignoreorientation* option.

JPEG 2000 images. JPEG 2000 images (ISO 15444-2) require PDF 1.5 or above. PDFlib accepts JPEG 2000 images according to the following conditions:

- JP2 and JPX baseline images (usually **.jp2 or *.jpf)* are supported, subject to the color space conditions below. All color depth values in the range 1-38 bits are supported. The following color spaces are supported: sRGB, sRGB-grey, ROMM-RGB, sYCC, e-sRGB, e-sYCC, CIELab, ICC-based color spaces, and CMYK. PDFlib does not alter the original color space in the JPEG 2000 image file.
- (Unsupported) Raw JPEG 2000 code streams without JPX wrapper (often **.j2k)* with 1, 3, or 4 color components can be imported.
- Restricted or full ICC profiles embedded in the JPEG 2000 image file are kept, i.e. the *honoriccprofile* option is always *true*.

*Note JPM compound image files according to ISO 15444-6 (usually *.jpm) are not supported.*

Additional JPEG 2000 restrictions for PDF/X-4/5 (JPEG 2000 is not allowed in PDF/X-3 which is based on PDF 1.4):

- The number of color channels must be 1, 3, or 4.
- The bit depth of each color channel must be 1, 8, or 16.

- All color channels must have the same bit depth.
- Exactly one color space definition must be present in the JPEG 2000 image file.
- CMYK images can only be used if the output condition is a CMYK device or the *defaultcmyk* option has been supplied.

Additional JPEG 2000 restrictions for PDF/A-2 (JPEG 2000 is not allowed in PDF/A-1 which is based on PDF 1.4):

- The number of color channels must be 1, 3, or 4.
- All color channels must have the same bit depth.
- If the number of color space specifications in the JPEG2000 image file is larger than one, there must be exactly one color space specification that has the value 0x01 in the *APPROX* field.

JBIG2 images. PDFlib supports single- and multi-page flavors of JBIG2 images (ISO 14492). JBIG2 images always contain black/white pixel data.

Due to the nature of JBIG2 compression, several pages in a multi-page JBIG2 stream may refer to the same global segments. If more than one page of a multi-page JBIG2 stream is converted the global segments can be shared among the generated PDF images. Since the calls to *PDF_load_image()* are independent from each other you must inform PDFlib in advance that multiple pages from the same JBIG2 stream will be converted. This works as follows:

- When loading the first page all global segments are copied to the PDF. Use the following option list for *PDF_load_image():*

```
page=1 copyglobals=all
```

- When loading subsequent pages from the same JBIG2 stream the image handle<*N*> for page 1 must be provided so that PDFlib can create references to the global segments which have already been copied with page 1. Use the following option list for *PDF_load_image():*

```
page=2 imagehandle=<N>
```

The client application must make sure that the *copyglobals/imagehandle* mechanism is only applied to pages which are extracted from the same JBIG2 image stream. Without the *copyglobals* options PDFlib will automatically copy all required data for the current page.

GIF images. PDFlib supports all GIF flavors and all palette sizes. GIF images are always recompressed with Flate compression. GIF files may contain more than one image (see »Multi-page images«, page 186); use the *page* option to select a specific image within a GIF file.

TIFF images. PDFlib processes all relevant flavors of TIFF images:

- Compression schemes: uncompressed, CCITT (group 3, group 4, and RLE), ZIP (=Flate), PackBits (=RunLength), LZW, old-style and new-style JPEG, as well as some other rare compression schemes;
- Color space: black and white, grayscale, RGB, CMYK, CIELab, and YCbCr images; the color space in imported TIFF images is retained unmodified with the following exception: LZW-compressed TIFF images with CIELab color are converted to RGB, and do not retain the CIELab color space.

- Color depth must be 1, 2, 4, 8, or 16 bits per color component. 16-bit images require PDF 1.5.
- The BigTIFF format which extends the original TIFF format beyond 4GB.

The following TIFF features are processed when importing an image:

- TIFF files containing more than one image (see »Multi-page images«, page 186); use the *page* option to select a specific image within a TIFF file.
- Alpha channels or masks (see Section 8.1.4, »Image Transparency«, page 191) are honored unless the *ignoremask* option is set. You can explicitly select an alpha channel with the *alphachannelname* option.
- PDFlib honors clipping paths in TIFF images created with Adobe Photoshop and compatible programs unless the *ignoreclippingpath* option is set.
- PDFlib honors embedded ICC profiles in TIFF images unless the *honoriccprofile* option is set to *false*.
- If a TIFF image contains an Exif marker the color space information in the Exif marker is interpreted. If it indicates sRGB color space the sRGB ICC profile is attached to the image (unless the image contains an explicit embedded ICC profile, the *honoriccprofile* option is *false* or another ICC profile has been assigned to the image with the *iccprofile* option).
- The orientation tag which specifies the desired image orientation is honored. It can be ignored (as many applications do) with the *ignoreorientation* option.

BMP images. 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 are treated as 5+5+5 plus one unused bit. 32-bit images are treated as 3 x 8 bit images (the remaining 8 bits are ignored).
- black and white or RGB color (indexed and direct color);
- uncompressed as well as 4-bit and 8-bit RLE compression;
- PDFlib does 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 passed through without uncompressing. Note that this format actually means plain 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.

In order to supply CCITT image data the *width* and *height* options must be supplied since PDFlib cannot deduce the image dimensions from the data. The specific type of CCITT compression can be supplied with the *K* option.

Raw images. Uncompressed raw image data may be useful for special applications. Note that this format is unrelated to Camera Raw files created by Adobe applications.

In order to supply raw image data the *width, height* and *bpc* options must be supplied since PDFlib cannot derive the image dimensions from the image data. The color space can be derived from the *components* option: 1 component implies a grayscale image, 3 components an RGB image, and 4 components a CMYK image. Alternatively a spot or DeviceN color can be applied to the image via the *colorize* option. In this case the color space and number of color components are derived from the color space handle.

The length of the supplied image data must be equal to

```
[width x components x bpc / 8] x height
```

bytes, with the bracketed term adjusted upwards to the next integer. Image samples are expected in top to bottom and left to right ordering (assuming no coordinate transformations have been applied). 16-bit samples must be provided with the most significant byte first (big-endian byte order). If *bpc* is smaller than 8, each pixel row begins on a byte boundary, and color values must be packed from left to right within a byte. Color channels are always interleaved, i.e. all color values for the first pixel must be supplied first, followed by the color values for the second pixel, and so on. The user is responsible for supplying option values which match the image. Otherwise corrupt PDF output may be generated, and Acrobat may respond with the message *Insufficient data for an Image*.

The polarity of the color values is the same as for color-related options (see Chapter 4, »Color Spaces«, page 77). Note that Adobe Photoshop uses the expected PDF polarity only for grayscale and RGB images, but inverse polarity for raw CMYK images and additional color channels: while PDF and PDFlib expect 0=no intensity, Photoshop assumes 0=maximum intensity. You can adjust the polarity by applying the *invert* option when loading the image.

8.1.3 Clipping Paths

PDFlib supports clipping paths in TIFF and JPEG images created with Adobe Photoshop or compatible programs. 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 remain invisible. This is useful to separate background and foreground, eliminate unwanted portions of an image, etc.

Alternatively, an image file may specify a default clipping path. If PDFlib finds a default clipping path in an image file it will automatically apply it to an image (see Figure 8.1). 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.

Fig. 8.1
Using a clipping path to separate foreground and background

Cookbook *A full code sample can be found in the Cookbook topic* `images/integrated_clipping_path`.

The vector operations for describing the clipping path are written to the PDF output at each call to *PDF_fit_image()*. If an image with clipping path is placed in the document more than once it is strongly recommended to wrap the image within a template in order to reduce the output file size. This can be achieved with the *templateoptions* option of *PDF_load_image()*.

8.1.4 Image Transparency

Image transparency is useful for a variety of artistic and other effects. For example, you can ignore the irrelevant parts of an image and display only the interesting person or object. PDFlib supports several methods for image transparency:

- An alpha channel (also called soft mask) specifies transparency values for each image pixel. The alpha channel may be part of the imported image file or can be specified as a separate grayscale image. Soft masks with more than 1 bit per pixel are not allowed in PDF/A-1 and PDF/X-1a/3.
- Chroma key masking specifies a single color value or a range of color value as transparent. The transparent color value(s) may come from the imported image file or can be specified via the *chromakey* option.
- Stencil masking uses a separate bitmap image which specifies transparent areas. The remaining image areas are painted with the current color or the main image.

Table 8.1 summarizes the available methods for masking an image with an alpha channel, chroma key value or stencil mask. The mask polarity differs for the alpha vs. stencil methods: the background shines through black areas of an alpha mask, but through white areas of a stencil mask.

Additional image color effects can be achieved with the *colorize* option (see Section 8.1.5, »Colorize Images with Spot or DeviceN Color«, page 194), blend modes and gstate soft masks (see Section 4.9, »Changing the Color of Objects«, page 98). For comparison the effects of colorizing an image with a spot or DeviceN color are also listed in Table 8.1.

Cookbook *Full code samples can be found in the Cookbook topic* `images/image_mask`.

Images with an internal alpha channel. PDFlib reads an internal alpha channel from the following image formats (unless *ignoremask=true):*

- TIFF images may contain a single alpha channel which is used by PDFlib. Alternatively, a TIFF image may contain multiple alpha channels which are identified by name. If multiple channels are present in a TIFF image PDFlib by default uses the first alpha channel. You can explicitly select another channel by supplying its name with the *alphachannelname* option:

```
image = p.load_image("tiff", filename, "alphachannelname={apple}");
```

- PNG images may contain an alpha channel which is used by PDFlib.
- JPEG 2000 images may contain an alpha channel which is used by PDFlib.

Note *As an alternative to a full alpha channel Photoshop can create transparent backgrounds in TIFF images using a proprietary format which is not supported in PDFlib. In order to use such transparent images with PDFlib you must save them in Photoshop in the TIFF file format and select* `Save Transparency` *in the TIFF options dialog box.*

Table 8.1 Supported methods for alpha masking, chroma key masking stencil masking and colorizing images

masking/colorizing method	***base image***	***auxiliary mask image***	***effect of pixels in the mask or base image***
alpha channel (soft mask) from image or `masked` ***option***			
image with internal alpha channel	*TIFF, PNG or JPEG 2000 image with internal alpha channel*	*(from image file)*	*(transparency according to the TIFF/ PNG/JPEG 2000 image format)*
image plus separate alpha channel image	*any image; option* `masked` *refers to soft mask in auxiliary image*	*bitmap or grayscale image*	*black mask = background gray mask = semi transparent white mask = base image*[1]
chroma key masking from image or `chromakey` ***option***			
image with internal chroma key value	*GIF or PNG image with internal chroma key entry*	–	*chroma key color = transparent other image colors = base image*
image plus separate chroma key value or range	*any image except JPEG and JPEG 2000; option* `chromakey` *specifies transparent color value(s)*	–	*chroma key color(s) = transparent other image colors = base image*
stencil masking with `mask` ***option***			
paint through mask image as stencil	–	*bitmap image loaded with option* `mask`	*black mask = fill color white mask = background*[1]
mask base image with stencil image	*any image; option* `masked` *refers to stencil mask in auxiliary image*	*bitmap image loaded with option* `mask`	*black mask = base image white mask = background*[1]
colorizing images with `colorize` ***option***			
colorize image with spot color	*grayscale image (>=1 bit per pixel); option* `colorize` *refers to spot color*	–	*black image = full spot color gray image = tinted spot color white image = white*
colorize image with DeviceN color	*n-channel image; option* `colorize` *refers to DeviceN color*	–	*0 in image = white intermediate = DeviceN tint 1 in image = full DeviceN color*

1. The effect of the mask can be inverted by applying the `invert` *option to the stencil or alpha image.*

Using a separate grayscale image as alpha channel. As an alternative to an internal alpha channel in the image file you can use a second image as source of an alpha channel. All kinds of grayscale images are suitable for use as alpha channel. If the mask contains an embedded ICC profile this must be ignored with *honoriccprofile=false*. For TIFF images the *nopassthrough* option for *PDF_load_image()* is recommended to avoid multi-strip images. BMP images are oriented differently than other image types. For this reason BMP images must be mirrored along the *x* axis before they can be used as a mask.

White pixels in the alpha image result in the corresponding area of the base image being painted, while black mask pixels result in the background shining through. If the mask uses more than one bit per pixel, intermediate values blend the foreground image against the background, providing a transparency effect.

After loading the mask it is applied to the base image with the *masked* option:

```
// load the mask image which serves as alpha channel
mask = p.load_image("png", maskfilename, "");
```

```
if (mask == -1)
        throw new Exception("Error: " + p.get_errmsg());

// load base image and mask it
image = p.load_image(type, filename, masked=" + mask)
if (image == -1)
        throw new Exception("Error: " + p.get_errmsg());

p.fit_image(image, x, y, "");
```

Visualizing an image alpha channel. Sometimes you may want to display an image alpha channel as grayscale image. This may be useful for debugging or to repurpose an alpha channel. It can be achieved by retrieving an image handle for the mask from the main image, using *PDF_info_image()* with the keyword *imagemask:*

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

alpha = (int) p.info_image(image, "imagemask", "");
if (alpha != -1)
        p.fit_image(alpha, 0.0, 0.0, "");
```

The alpha channel is treated as grayscale image: transparent areas appear black.

Chroma key masking. Images may specify a single color or a range of colors to be masked out. Pixels with colors in the specified range are not painted so that the background shines through. In video technology this technique is called chroma key or blue screen masking. It is supported with the *chromakey* image option. This option expects *n* or *2xn* integers in the range *0* to $2^{bitspercomponent}-1$, where *n* is the number of components in the image colorspace.

Each pair in the list contains the inclusive lower and upper bounds of a color component range. Pixels where all color components (before applying any *decode* values or color inversion with the *invert* option) fall in the specified ranges are treated as transparent, i.e. they are not painted but allow the background to shine through.

If *n* values instead of *n* pairs are supplied each component range contains only a single color value, i.e. each list value describes both lower and upper bound for a color component.

Table 8.2 lists various examples of the *chromakey* option.

Table 8.2 Examples for the chromakey *image option*

chromakey option	*image color space*	*effect*
`chromakey={255 255 255}`	*RGB*	*treat white pixels as transparent*
`chromakey={0 255 128 255 0 255}`	*RGB*	*treat all pixels with more than 50% green as transparent*
`chromakey={242 255 242 255 242 255}`	*RGB*	*treat all colors lighter than 95% as transparent*
`chromakey={242 255 242 255 242 255}` `decode={0 0.95 0 0.95 0 0.95}`	*RGB*	*treat all colors lighter than 95% as transparent and spread the remaining colors to avoid sudden truncation at 95%*

Chroma key masking is applied automatically in the following cases (unless *ignoremask=true)*:

- GIF images may contain a single transparent color value (palette entry) which is respected by PDFlib.
- PNG images may contain a single transparent color value which is respected by PDFlib. If multiple color values with a corresponding alpha value are present, only the first one with an alpha value below 50 percent is used.

Stencil masks. Stencil masks are bitmap images with a bit depth of 1 where white pixels are treated as transparent: whatever content already exists on the page shines through the transparent parts of the image. The areas with 0 pixel values can be painted with the current fill color, or can be used to show parts of another image.

In order to colorize a stencil mask with the current fill color you must load the image with the *mask* option and set the current fill color before placing the image: Black pixels are colorized with the current fill color, while white areas remain unchanged:

```
mask = p.load_image("tiff", maskfilename, "mask");
if (mask == -1)
        throw new Exception("Error: " + p.get_errmsg());

p.set_graphics_option("fillcolor=red);
p.fit_image(mask, x, y, "");
```

If the stencil image is used to mask another image with the *masked* option, the pixels of that base image are visible in black areas of the stencil mask, and the background in white areas.

8.1.5 Colorize Images with Spot or DeviceN Color

Similarly to stencil masks, where a color is applied to the non-transparent parts of a bitmap image PDFlib supports colorizing images with a spot or DeviceN color.

Colorize a grayscale image with a spot color. Black and white or grayscale images can be colorized with a spot color. To colorize an image with a spot color you must supply the *colorize* option when loading the image. This option contains a spot color handle created with *PDF_makespotcolor( ):*

```
spot = p.makespotcolor("PANTONE Reflex Blue CV");

String optlist = "colorize=" + spot;
image = p.load_image("tiff", "image.tif", optlist);
```

As explained in Section 4.4, »Pantone, HKS, and custom Spot Colors«, page 84, spot color spaces usually expect *0=no color=white* in contrast to the grayscale color space where *0=black*. In order to ensure that white remains white PDFlib inverts the image polarity when colorizing an image with a spot color, i.e. dark image areas result in full spot color intensity. You can revert the color polarity with the *invert* image option.

Colorize an n-channel image with a DeviceN color. A DeviceN color handle created with *PDF_create_devicen( )* can be supplied in the *colorize* image option to apply DeviceN color to an n-colorant image. In order to colorize an image with a DeviceN color the image data must contain the appropriate number N of color channels.

As explained in Section 4.5, »DeviceN Colors«, page 88, DeviceN color spaces expect *0=no color=white* in contrast to the grayscale color space where *0=black*. Unlike when colorizing an image with spot colors, PDFlib does *not* invert the image polarity when colorizing an image with a DeviceN color. As a result, colorizing with a spot color and colorizing with a DeviceN color with *N=1* expect different color polarities. You can invert the color polarity with the *invert* image option.

A code fragment for colorizing an image with a DeviceN color space can be found in »DeviceN color space based on CMYK process colors«, page 89.

Note *Only raw images can be colorized with DeviceN color.*

Cookbook *A code sample can be found in the Cookbook topic* `images/colorize_image_with_DeviceN_color`.

8.1.6 Modifying Color Values with a Decode Array

The color values of an image can be further modified with a linear decoding function. This method is supported with the *decode* image option. It expects *2xn* float or percentage values, where *n* is the number of components in the image colorspace. Applying decode arrays therefore requires knowledge of the image color space. Each pair of numbers in a decode array describes the target color values to which the lowest and highest component values *0* and $2^{bitspercomponent}-1$ are mapped; intermediate values are interpolated linearly. By default, image component values are mapped to the range 0..1 for most colorspaces. Decode arrays can be used to apply certain color effects by compacting, spreading or clamping the color values. This technique may also be useful if the image is colorized or used as a mask.

Decode values have different effects on additive vs. subtractive colorspaces. For example, increasing the values results in lighter RGB colors, but darker CMYK colors. For most colorspaces (except Lab and Indexed) typical values are in the range 0..1. Values outside this range are allowed, but the viewer clips the resulting color values to the appropriate range for the colorspace. This can be used for certain effects, e.g. to force a range of light colors to white.

Applying the *decode* option to an image which is used as a soft mask to another image can modify the effect of the mask, e.g. soften the mask.

The following image option shifts the black target values from the default range 0..1 to the new range 0.2..1.2, i.e. it increases the black channel of a CMYK image by 20%, resulting in a stronger image:

```
decode={0 1  0 1  0 1  0.2 1.2}
```

Table 8.3 lists various examples of the *decode* option.

Table 8.3 Examples for the `decode` *image option*

decode option	*image color space*	*effect*
`decode={1 0 1 0 1 0}`	*RGB*	*invert image colors; same as* `invert`
`decode={-0.5 1.5 -0.5 1.5 -0.5 1.5}`	*RGB*	*increase contrast*
`decode={0.5 1.5 0.5 1.5 0.5 1.5}`	*RGB*	*lighten all colors by 50%*
`decode={-0.2 0.8 -0.2 0.8 -0.2 0.8}`	*RGB*	*darken all colors by 20% (since 100%=white)*

Table 8.3 Examples for the decode *image option*

decode option	***image color space***	***effect***
`decode={-0.2 0.8 -0.2 0.8 -0.2 0.8 -0.2 0.8}`	*CMYK*	*lighten all colors by 20% (since 100%=dark)*
`decode={0 1 0 1 0 1 0 0}`	*CMYK*	*remove black channel, resulting in a duller image*
`decode={0 1 0 1 0 1 0.2 1.2}`	*CMYK*	*increase black channel by 20%, resulting in a stronger image*
`decode={1 0 0 1 0 1 0 1}`	*CMYK*	*invert Cyan channel while keeping other channels unchanged; unlikely to be useful except for artistic purposes*

8.2 SVG Graphics

8.2.1 Supported SVG Flavors

PDFlib is a »Conforming High-Quality Static SVG Viewer« according to W3C nomenclature. PDFlib accepts SVG graphics as follows:

- PDFlib implements SVG 1.1 (Second Edition) as published by the W3C.
- The following Unicode formats and encodings are supported: UTF-8, UTF-16, ISO 8859-1 ... ISO 8859-15, ASCII
- CSS styling is available, but some CSS elements are unsupported.
- In addition to SVG files in plain text format Flate-compressed SVG files *(*.svgz)* are supported.
- Fonts in the CEF format are supported. CEF fonts are not part of the SVG specification, but are embedded in SVG graphics by some Adobe applications.
- Colors can be specified with additional color spaces according to the SVG Color 1.2 draft including a PDFlib-specific extension for the tint value of spot colors (see Section 8.2.6, »SVG Color Extension«, page 203).

See Section 8.2.8, »Unsupported SVG Features«, page 206, for restrictions.

8.2.2 SVG Processing Considerations

Basic SVG handling. Embedding vector graphics with PDFlib is easy to accomplish. First, the graphics file has to be opened with a PDFlib function which interprets the graphics and stores an internal representation in memory. The *PDF_load_graphics()* function returns a handle which serves as a graphics descriptor. This handle can be used in a call to *PDF_fit_graphics()*, along with positioning and scaling options:

```
graphics = p.load_graphics("auto", "graphics.svg", "");
if (graphics == -1)
        throw new Exception("Error: " + p.get_errmsg());

if (p.info_graphics(graphics, "fittingpossible", optlist) == 1)
        p.fit_graphics(graphics, 0.0, 0.0, "");
else
        System.err.println("Cannot place graphics: " + p.get_errmsg());

p.close_graphics(graphics);
```

The last parameter of *PDF_fit_graphics()* is an option list which supports a variety of options for positioning, scaling, and rotating. Details regarding these options are discussed in Section 8.4, »Placing Images, Graphics, and imported PDF Pages«, page 213.

Cookbook *Code samples for SVG handling can be found in the* `graphics` *category of the PDFlib Cookbook.*

SVG content as inline graphics or Form XObject (template). PDFlib supports the following methods for importing vector graphics:

- By default, the graphics data is written inline in the content stream of the page, pattern, template, or glyph description. This is the default behavior, which is recommended for situations where the graphics is placed exactly once in the document and where you don't want to change the opacity of the imported graphics. If *PDF_fit_*

graphics( ) is called more than once, the graphics data is written again and again to the PDF output which increases output file size.

- If the graphics is intended to be placed multiply in the document, the *template-options* option of *PDF_load_graphics( )* is recommended. The option *templateoptions={transparencygroup={isolated=true}}* is required if you intend to change the opacity of the imported graphics. It creates a PDF Form XObject (template), i.e. the graphics data is stored in the PDF document as a separate entity which can be referenced an arbitrary number of times. The graphics data for the template is written to the PDF output at the end of the document or when *PDF_close_graphics( )* is called. This way the output file size is optimized. However, links within graphics are no longer converted to PDF annotations.

Using the same graphics in multiple documents. Graphics can be loaded and closed independently from the current output document. When *PDF_load_graphics( )* is called an internal representation of the graphics is created. It is kept in memory until the corresponding call to *PDF_close_graphics( )*. Keeping graphics in memory across documents has performance advantages in situations where the same graphics are placed in many output documents since the graphics must be loaded only once. For example, an application may load graphics with symbols, background artwork or company stationery once and call *PDF_fit_graphics( )* in each document where the graphics is required.

Checking for SVG processing problems. While *PDF_load_graphics( )* loads the SVG graphics, full processing and analysis is done only later in *PDF_fit_graphics( )*, *PDF_close_graphics( )* or *PDF_end_document( )* depending on Form XObject creation and scope of loading. Since some error situations can be detected only during full processing, these functions could throw an exception if a problem was found (since they cannot return any error values). In order to avoid such an exception the graphics can be checked with the *fittingpossible* keyword of *PDF_info_graphics( )*. It performs all processing steps and doesn't create any output, but reports the success (or not) of SVG processing. If this check succeeds, *PDF_fit_graphics( )* will not throw an exception when the image is placed. If an error occurs during the *fittingpossible* check, *PDF_info_graphics( )* returns 0 regardless of *errorpolicy*. To summarize:

- The *fittingpossible* check prevents later exceptions in *PDF_fit_graphics( )*, *PDF_close_graphics( )* or *PDF_end_document( )*. Since the PDF output would be unusable after an exception this is the recommended approach.
- Skipping the *fittingpossible* check speeds up SVG loading, but exceptions triggered by SVG data may occur later. This setting can be used to speed up SVG loading if exceptions in *PDF_fit_graphics( )* are acceptable. For example, if the application converts a single SVG graphics file to a single PDF document without any additional page content this would be an acceptable approach.

The *fittingpossible* check uses the currently active global, document, and page options as well as the current output intent. It is therefore recommended to run this check only immediately before the actual call to *PDF_fit_graphics( )*.

8.2.3 Visible Size of SVG Graphics

SVG graphics specify the width and height in the *svg* element which defines the mapping of the SVG graphics to the target viewport (e.g. the browser window or some part of a PDF page). Often the size of the viewport is specified in absolute units, e.g.

```
<svg xmlns="http://www.w3.org/2000/svg" width="640mm" height="480mm">
```

PDFlib converts the *width* and *height* attributes to points and makes them available via the *graphicswidth* and *graphicsheight* keywords of *PDF_info_graphics()*. If the size is specified in pixels *(px)* PDFlib uses 1pt=1px. These values are also used to calculate the object box for fitting operations. The *svg* element may also contain the *viewBox* attribute which specifies a window inside the viewport.

You can override the size values specified in the SVG graphics file with the *forcedwidth* and *forcedheight* options. The suboption *clipping* of the *matchbox* option can be used to clip the graphics to some part of the object box.

SVG graphics without absolute size information. Some SVG graphics do not contain absolute size information since *width* and *height* are missing or contain only relative values such as in the following example:

```
<svg xmlns="http://www.w3.org/2000/svg" width="100%" height="100%">
```

This is used for SVG graphics which are not intended for standalone use. PDFlib supports this case if the user has specified a fitbox with the options *boxsize* and *fitmethod=nofit*. The fitbox is used as object box. The SVG graphics will be clipped at the border of the fitbox. If no fitbox is specified PDFlib uses the *viewBox* attribute (if present) as object box.

If no absolute size information is available in the SVG graphics you can supply the *fallbackwidth* and *fallbackheight* options of *PDF_load_graphics()*. In the absence of these options PDFlib calculates the bounding box of the SVG graphics in the first call of *PDF_fit_graphics()* or *PDF_info_graphics()*. The calculated bounding box may be too small because line widths and oversized glyphs are not taken into account. In this case the calculated box can be enlarged with the *bboxexpand* option. By default the calculated bounding box is moved such that the graphics is located at the original position in the coordinate system (i.e. the box does not necessarily use the origin as corner).

8.2.4 Font Selection

Font selection algorithm. Font selection in SVG is controlled by the following properties:

```
font-family
font-style
font-weight

font-stretch
font-variant
font-size
font-size-adjust
```

Only the first three of these properties are relevant for selecting an external font.

In order to select a suitable font, PDFlib constructs the following font names:

```
<font-family>,<font-weight>,<font-style>
<font-family>-<font-weight><font-style>
<font-family>,<font-normweight>,<font-style>
<font-family>,<font-weight>,<font-normstyle>
<font-family>,<font-normweight>,<font-normstyle>
```

where <*font-normweight*> is one of

```
Regular, Thin, Extralight, Light, Medium, Semibold, Bold, Extrabold, Black
```

and <*font-normstyle*> is

```
Italic
```

For example, with the following SVG font specification:

```
font-family="Tahoma" font-weight="Bold" font-style="Italic"
```

PDFlib will search for the font *Tahoma,Bold,Italic,* using the comma-separated PDFlib syntax for specifying font styles which can also be used to address Windows host fonts.

PDFlib then tries to load the font names listed above one after the other until the process is successful and a font could be loaded. The font names in this list can be used in font resource specifications, e.g.

```
p.set_option("FontOutline={<fontname>=<filename>}")
p.set_option("FontNameAlias={<fontname>=ArialMT}")
```

If all attempts fail, PDFlib tries to load the font with the name <*font-family*> and simulates the *Bold* and *Italic* properties if required.

Some browsers ignore the font selection properties if the specified font family cannot be found. Since PDFlib doesn't do this you must take care to make suitable fonts available via PDFlib's font configuration mechanism (see Section 6.4, »Loading Fonts«, page 133).

The *font-family* property may also contain multiple font family names, e.g.

```
font-family="Georgia, 'Minion Web', 'Times New Roman', Times, 'MS PMincho', serif"
```

In this case PDFlib tries to load the next font in the list if a particular font family could not be loaded. If some font for a *font-family* list could be loaded, PDFlib attempts to load the remaining *font-families* in the list as fallback fonts for the first loaded font (the master font, see Section 6.4.6, »Fallback Fonts«, page 146). If the master font already has fallback fonts because it has been loaded earlier, the new fallback fonts will be appended to the list of existing fallback fonts.

When PDFlib attempts to load a font for SVG graphics the following options are used by default:

```
embedding subsetting skipembedding={fstype latincore}
```

These options can be overridden with the option *defaultfontoptions* of *PDF_load_graphics( )*.

Font configuration. On Windows systems PDFlib can access all fonts which are installed on the system (see Section 6.4.5, »Host Fonts on Windows and macOS«, page 144). For example, the SVG font specification

```
font-family="Verdana" font-weight="bold"
```

results in the PDFlib font name *Verdana,Bold*. On other operating systems PDFlib will find a font if a *FontOutline* resource has been specified in the following way:

```
<fontnamepattern>=<filename.xxx>
```

where *<fontnamepattern>* is one of the font name patterns above and *xxx* is the corresponding file name extension of the font outline file.

Suitable *FontOutline* resources with font names which match one or more of the font name patterns above can be created automatically with the *enumeratefonts* option of *PDF_set_option()*. Refer to Section 6.4.4, »Searching for Fonts«, page 140, for more information on font configuration.

Mapping generic SVG font families to PDF core fonts. PDFlib automatically maps the generic SVG font families *monospace*, *sans-serif* and *serif* to the Latin core fonts at the first occurrence of a generic SVG font family, using *FontNameAlias* resources of the following form:

```
p.set_option("FontNameAlias={monospace=Courier}")
p.set_option("FontNameAlias={monospace,Bold=Courier-Bold}")
```

The full list of generic font name mappings is as follows (there are no default mappings for the generic font families *cursive* and *fantasy):*

```
monospace               Courier
monospace,Bold          Courier-Bold
monospace,Italic        Courier-Oblique
monospace,Bold,Italic   Courier-BoldOblique

sans                    Helvetica
sans,Bold               Helvetica-Bold
sans,Italic             Helvetica-Oblique
sans,Bold,Italic        Helvetica-BoldOblique

sans-serif              Helvetica
sans-serif,Bold         Helvetica-Bold
sans-serif,Italic       Helvetica-Oblique
sans-serif,Bold,Italic  Helvetica-BoldOblique

serif                   Times-Roman
serif,Bold              Times-Bold
serif,Italic            Times-Italic
serif,Bold,Italic       Times-BoldItalic
```

These mappings are executed only if no other suitable resource has been specified by the user before.

8.2.5 Dealing with missing Fonts and missing Glyphs

Missing fonts and the default font. If all font loading attempts fail, PDFlib attempts to load a default font which has the *font-family* name defined in the *defaultfontfamily* option of *PDF_load_graphics()*. By default, the *Arial Unicode MS* font is used if available, and Helvetica otherwise. It is highly recommended to either make the *Arial Unicode MS* font available in PDFlib's font configuration, or to specify another font with a large glyph complement in *defaultfontfamily*. For example, to configure the *NotoSans-Regular* font as last resort font use the following option:

```
defaultfontfamily={NotoSans-Regular}
```

Replacing individual fonts. In order to avoid a particular font which is not available or not desirable (e.g. because it does not contain enough glyphs, see below) you can map it to another font which is more suitable. Use the font name alias feature and *PDF_set_option()* for this purpose (see »Font name aliasing«, page 140, for details). For example, if you have Chinese text which is inappropriately set with the *Trebuchet MS* font which doesn't contain any Chinese glyphs, you can map it to *Arial Unicode MS* as follows:

```
p.set_option("FontnameAlias={ {Trebuchet MS}={Arial Unicode MS} }");
```

Keep in mind that font attributes are not automatically added. For example, if the *Trebuchet MS* font is used with the attribute *font-weight="bold"* you must create an alias for the bold version of the font:

```
p.set_option("FontnameAlias={ {Trebuchet MS,Bold}={Arial Unicode MS} }");
```

Visualizing missing glyphs. If a selected font does not contain a required glyph, the default replacement glyph is used instead, which means that no text will be visible at all. In order to visualize missing glyphs you can specify a visible replacement glyph using the *defaultfontoptions* option. For example, the following option for *PDF_load_graphics()* will display a question mark for all missing glyphs:

```
defaultfontoptions={replacementchar=?}
```

Specifying a particular fallback font for missing glyphs. If the font specified in SVG does not contain suitable glyphs for the text, no text will be visible. As a common example consider Chinese text which is attempted to be shown with a Western font which doesn't contain any Chinese glyphs. Of course the best solution would be to use suitable fonts in the SVG in the first place. However, if you have to deal with unsuitable fonts in SVG you can specify a fallback font in PDFlib. This fallback font is used whenever the original font does not provide a particular glyph.

The following option for *PDF_load_graphics()* calls for Arial Unicode MS as fallback font:

```
defaultfontoptions={fallbackfonts={{fontname={Arial Unicode MS} encoding=unicode}}}
```

Note that the font specified in the *defaultfontfamily* option shown earlier is used when a font cannot be found, while the fallback technique applies when a font is available, but doesn't contain all required glyphs.

Specifying a global fallback font family. The *fallbackfontfamily* and *fallbackfontoptions* options can be used to specify a family of fallback fonts and corresponding options. While the *fallbackfonts* option within *defaultfontoptions* selects a single font for use as fallback font, *fallbackfontfamily* can be used to specify a family of fallback fonts, i.e. style attributes will be applied to this font family name, assuming that style variants of the specified font family are actually available. Example:

```
fallbackfontfamily={Arial} fallbackfontoptions={encoding=unicode}
```

8.2.6 SVG Color Extension

Cookbook *A code samples for non-sRGB color in SVG can be found in the Cookbook topic* `color/svg_color_extension`.

SVG by default uses the sRGB color space for text and vector graphics. PDFlib therefore treats SVG graphics as device-independent color for PDF/X and PDF/A. Embedded images in SVG may use other color spaces, though. For example, an embedded or referenced JPEG image may use the CMYK color space with or without an ICC profile. The default sRGB profile for RGB colors can be overridden with the *iccprofilergb* option of *PDF_load_graphics()*.

In addition to sRGB PDFlib supports additional color spaces using syntax extensions according to the SVG Color 1.2 draft specification. Table 8.4 lists supported SVG color spaces along with syntax examples. See Chapter 4, »Color Spaces«, page 83, for general information about these color spaces and their treatment in PDFlib.

SVG syntax requires sRGB fallback colors for all variants. PDFlib uses the fallback color in case of syntax errors, missing spot color definitions and missing ICC profiles or if the option *forcesrgb* has been supplied.

You can colorize or decolorize SVG images with the help of blend modes (see Section 4.9.1, »Changing the Color with Blend Modes«, page 98).

ICC-based colors. The *icc-based* color specifications apply the specified ICC profile which is searched according to the *ICCprofile* resource category unless it references a local file. ICC profiles are ignored if the option *honoriccprofile=false* is supplied. Use the options *iccprofilegray/iccprofilergb/iccprofilecmyk* to override the profiles specified in the SVG graphics with other profiles.

Gray, RGB and CMYK device color spaces. The device-specific color spaces *device-gray, device-rgb* and *device-cmyk* are not recommended for general use as their rendering depends on the specific output device unless controlled by a PDF/A or PDF/X output intent ICC profile or suitable default color space. It is strongly recommended to work with ICC-based color spaces instead. Particular exceptions may be the use of a CMYK control strip to directly control tint values on a printing machine, or printing registration marks.

Device-dependent colors created by *device-gray/rgb/cmyk* attributes or referenced images can be mapped to ICC-based colors via the *iccprofile* suboption of *defaultimageoptions* or the *defaultgray/rgb/cmyk* suboptions of *templateoptions*.

Spot colors and DeviceN colors. The *icc-named-color* syntax can be used to specify spot colors. However, since named color ICC profiles are not supported the spot color must be known to PDFlib; otherwise the sRGB fallback color is used as alternate color. Custom

spot colors must therefore be defined in your code (see »Custom spot colors«, page 86) before calling *PDF_load_graphics( )*, e.g.

```
// Define custom spot color "CompanyRed" with Lab alternate values
p.set_graphics_option("fillcolor={spotname {CompanyRed} 1.0 {lab 60 65 65}}");
```

The *device-nchannel* syntax can be used to select DeviceN colors which must be created with *PDF_create_devicen( )* (see Section 4.5, »DeviceN Colors«, page 88) supplied to *PDF_load_graphics( )* via the *devicencolors* option. This implies that only a single DeviceN color space can be used for each value of *N*.

Shadings. The color spaces listed in Table 8.4 can also be used in SVG gradients. However, PDF requires all stop colors for a gradient to be taken from the same color space. If a gradient uses stop colors from different color spaces the sRGB fallback colors for all stop colors are used.

Shadings created from spot colors retain the spot colors provided these have been created with Lab alternate values (which is true for all spot colors known internally to PDFlib). Spot colors created with other alternate color spaces such as CMYK will not be retained, but result in sRGB shadings in the PDF output.

Note *Since non-sRGB colors in SVG have not been fully standardized yet the syntax may change in the future.*

Table 8.4 Extended color spaces in SVG (can be disabled with `forcesrgb`*)*

color space	*SVG syntax examples*	*notes*
Device-independent color spaces		
sRGB	`#FF0000`	*SVG 1.1 default color space*

ICC-based color	`#7F7F7F icc-color(gray.icc, 0.5)`
	`#CC6633 icc-color(rgb.icc, 0.8, 0.4, 0.2)`
	`#2B7FAB icc-color(cmyk.icc, 0.8, 0.4, 0.2, 0.0)`
		One, three or four color values must be supplied for Grayscale, RGB or CMYK profiles, respectively.

CIELab	`#598237 cielab(50, -25, 35)`	
CIELch	`#FF007E cielch(50, 127, 0)` `#FF007E cielchab(50, 127, 0)`	*Lightness, chroma and hue (in degrees) are converted to CIELab.*

well-known spot color	`#FFB12D icc-named-color(nmcl.icc, 'PANTONE 123 U')`
	`#FFCE4C icc-named-color(nmcl.icc, 'PANTONE 123 U', 0.5)`
	`#994F5B icc-named-color(nmcl.icc, 'HKS 18 Z')`
		The profile name is ignored. As an extension to the SVG syntax PDFlib supports an optional tint value. If no tint value is present, 1.0 is assumed.

custom spot color	`#FFE560 icc-named-color(nmcl.icc, 'CompanyColor')` `#FFF36D icc-named-color(nmcl.icc, 'CompanyColor', 0.5)`	*The sRGB fallback color is used as alternate color unless the spot color has already been defined earlier; the profile name is ignored. If no tint value is present, 1.0 is assumed.*
n-colorant	`#FFE560 device-nchannel(0, 0.7, 0.2)` `#FFF36D device-nchannel(1 0 0 0 0.5 0.2 0)`	*DeviceN color space handles must be supplied via the* `devicencolors` *option. If no DeviceN color space handle for* `n` *has been supplied the sRGB fallback color is used.*

Table 8.4 Extended color spaces in SVG (can be disabled with `forcesrgb`)

color space	*SVG syntax examples*	*notes*
Device-dependent color spaces		
DeviceGray	`#7F7F7F device-gray(0.5)`	*PDF/A and PDF/X requirements for device color spaces must be obeyed.*
DeviceRGB	`#0000FF device-rgb(0, 0, 255)`	*PDF/A and PDF/X requirements for device color spaces must be obeyed.*
DeviceCMYK	`#00AEEF device-cmyk(1, 0, 0, 0)`	*PDF/A and PDF/X requirements for device color spaces must be obeyed.*

8.2.7 SVG Contents beyond Vector Graphics and Text

Embedded images in SVG. PDFlib processes the *image* element in SVG and accepts all of the image formats discussed in Section 8.1, »Raster Images«, page 185, and nested SVG graphics. Image data may be embedded in the SVG file or located in an external file.

Images in SVG graphics are processed automatically. However, in some situations you may want to supply certain image processing options. This can be achieved with the *defaultimageoptions* option of *PDF_load_graphics( )*. For example, use the following option to apply antialiasing which improves the appearance of low-resolution images (this matches the SVG display of most browsers which apply anti-aliasing to images):

```
defaultimageoptions={interpolate}
```

If an image is not available (e.g. because the referenced external image file is missing) and the option *fallbackimage={ }* (i.e. an empty option list) has been supplied, PDFlib creates a transparent gray checkerboard pattern. Suboptions of the option *fallbackimage* can be used to customize this pattern or to supply a custom image or template as fallback visualization. Use the following option if you want to avoid that missing images are silently replaced:

```
errorconditions={references={image}}
```

SVG links. Links in SVG graphics are generally converted to interactive Link annotations in the generated PDF output; however, there are a few conditions which disable link creation (see PDFlib API Reference). Links which are located outside the graphics are ignored. The *contents* option of the PDF annotation is populated with the *xlink:title* attribute of the SVG link if present, and the target URI otherwise. In Tagged PDF mode a *Link* element and an associated OBJR element are created for the generated link unless the currently active item is an Artifact or pseudo element.

Conversion of SVG links to PDF links can be disabled with the *convertlinks* option of *PDF_fit_graphics( )*. Note that links cannot be created when a template (Form XObject) is created for the graphics or the graphics is placed on a template.

Metadata in SVG. SVG graphics may contain XMP metadata. By default PDFlib ignores SVG metadata in graphics to reduce the output file size. However, if a template is created from SVG the XMP metadata can be attached to the generated XObject with the following option of *PDF_load_graphics( )*:

```
templateoptions={metadata={keepxmp=true}}
```

The contents of the *metadata*, *desc*, and *title* elements of an SVG graphic can be retrieved with *PDF_info_graphics( )* according to the following pattern:

```
idx = (int) p.info_graphics(svg, "description", "");
if (idx != -1)
        description = p.get_string(idx, "");
```

8.2.8 Unsupported SVG Features

Treatment of unsupported features. By default, unsupported SVG features are ignored. As a result, output is created but some aspects of the graphics may be missing or wrong. This behavior can be changed with the *errorconditions* option of *PDF_load_graphics( )*. Its suboptions specify conditions which trigger an error instead of being ignored. For example, with the following option list *PDF_load_graphics( )* fails if the SVG graphics contain animated or scripted elements, e.g.:

```
errorconditions = {elements={animate script}}
```

General restriction. The following restriction affects various elements:

- references to external URLs are not resolved (image, font, etc.)

Unsupported SVG elements. The following SVG elements are not supported and will be ignored:

- elements for animation and scripting:

  ```
  animate, animateColor, animateMotion, animateTransform, script, mpath, set
  ```

- elements for SVG filters:

  ```
  feBlend, feColorMatrix, feComponentTransfer, feComposite, feConvolveMatrix,
  feDiffuseLighting, feDisplacementMap, feDistantLight, feFlood, feFuncA, feFuncB,
  feFuncG, feFuncR, feGaussianBlur, feImage, feMerge, feMergeNode, feMorphology,
  feOffset, fePointLight, feSpecularLighting, feSpotLight, feTile, feTurbulence, filter
  ```

- other elements:

  ```
  cursor, foreignObject, vkern
  ```

Restricted SVG elements, attributes and properties. The following attributes and properties are subject to restrictions:

- Some CSS rules are not supported, including *@import* and *@font-face*.
- The font selection property *font-variant* is supported only with the keyword *small-caps*, and only for fonts containing the OpenType feature *smcp*.
- Combination of values of the text presentation property *text-decoration* are not supported. PDFlib doesn't draw the decoration elements as areas with separate fill and stroke color, but draws the decorations as lines. The lines are drawn with the fill color if present, otherwise with the stroke color.
- The attribute *rotate* for the *textPath* element is not supported.
- The attribute *xlink:href of the use* element is supported only for local IRI references. For example, a reference to an external SVG file within a *use* element is not supported.
- The property *unicode-bidi* is honored only for TrueType/OpenType fonts containing the required tables for Bidi text layout. PDFlib sets the options *shaping* and *script=_auto* in the option list of *PDF_fit_textline( )*.

- The property *glyph-orientation-vertical* is supported according to SVG 1.1 except the angles 180° and 270° which are not supported.
- The attribute *preserveAspectRatio* for the *view* element is ignored.
- The property *overflow* for the *svg* element does not support the value *visible* if a Form XObject (template) is created with the option *templateoptions*.

Unsupported SVG properties. The following SVG properties are not supported and are ignored:

```
alignment-baseline, color-interpolation, color-interpolation-filters,
color-rendering, cursor, dominant-baseline, enable-background, filter, flood-color,
flood-opacity, glyph-orientation-horizontal,
image-rendering, lighting-color, pointer-events, shape-rendering, text-rendering
```

Unsupported attributes of supported SVG elements. The following attributes of supported SVG elements are not supported and are ignored:

```
baseProfile (svg)
contentScriptType (svg)
contentStyleType (svg)
externalResourcesRequired (all elements)
method (textPath)
on* (all elements)
spacing (textPath)
textLength and lengthAdjust for vertical text (text, textpath, tref, tspan)
version (svg)
zoomAndPan (svg)
xlink:role (all elements)
xlink:show (all elements)
xlink:type (all elements)
```

8.3 Importing PDF Pages with PDI

Note *All functions described in this section require PDFlib+PDI or the PDFlib Personalization Server PPS (which includes PDI). The PDF import library (PDI) is not contained in the PDFlib base product. Although PDI is integrated in all binary editions of PDFlib, a license key for PDFlib+PDI or PPS is required to use it.*

8.3.1 PDI Features and Applications

If PDI (PDF import library) is available, pages from existing PDF documents can be imported. PDI prepares pages from existing PDF documents for use with PDFlib. Conceptually, imported PDF pages are treated similarly to imported raster images: you open a PDF document, choose a page to import, and place it on an output page. You can apply any of PDFlib's transformation functions for translating, scaling, rotating, or skewing the imported page. Imported pages can be combined with new content by using PDFlib's text or graphics functions after placing the imported PDF page on the output page (think of the imported page as the background for new content). Using PDFlib and PDI you can easily accomplish the following tasks:

- overlay two or more pages from multiple PDF documents (e.g., add stationary to existing documents in order to simulate preprinted paper stock);
- place PDF ads in existing documents;
- clip the visible area of a PDF page in order to get rid of unwanted elements (e.g., crop marks), or scale pages;
- impose multiple pages on a single sheet for printing;
- process multiple PDF/X or PDF/A documents to create a new PDF/X or PDF/A file;
- copy the PDF/X or PDF/A output intent of a file;
- add some text (e.g., headers, footers, stamps, page numbers) or images (e.g., company logo) to existing PDF pages;
- copy all pages from an input document to the output document, and place barcodes on the pages;
- use the pCOS interface to query arbitrary properties of a PDF document (see pCOS Path Reference for details).

In order to place a PDF background page and populate it with dynamic data (e.g., mail merge, personalized PDF documents on the Web, form filling) we recommend using PDI along with PDFlib blocks (see Chapter 13, »PPS and the PDFlib Block Plugin«, page 355).

8.3.2 Using PDFlib+PDI

Cookbook *Code samples regarding PDF import issues can be found in the* `pdf_import` *category of the PDFlib Cookbook.*

General considerations. It is important to understand that PDI only imports 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.

The following items can optionally be imported:

- Structure element tags can be imported (see »Importing pages from Tagged PDF documents«, page 211, for details).

- Layer definitions can be imported (see »Importing PDF pages with layers«, page 211, for details).
- (PPS only) PDFlib Blocks can be imported with *PDF_process_pdi()* and the option *action=copyallblocks* or *copyblock* (see Section 13.9.2, »Importing PDFlib Blocks«, page 397).

You cannot re-use individual elements of imported pages with other PDFlib functions. For example, re-using fonts from imported documents for some other content is not possible. Instead, all required fonts must be configured in PDFlib. If multiple imported documents contain embedded font data for the same font, PDI will not remove any duplicate font data. On the other hand, if fonts are missing from some imported PDF, they will also be missing from the generated PDF output file. As an optimization you should keep the imported document open as long as possible in order to avoid the same fonts to be embedded multiple times in the output document.

PDFlib+PDI uses the template feature (Form XObjects) for placing imported PDF pages on the output page. Documents which contain imported pages from other PDF documents can be processed with PDFlib+PDI again.

Code fragment 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, pagecount, 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());

/* Number of pages in the input document; useful for importing all pages */
pagecount = (int) p.pcos_get_number(doc, "length:pages");

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 contents to the page using PDFlib functions...
p.end_page_ext("");
p.close_pdi_document(doc);
```

The last parameter to *PDF_fit_pdi_page()* is an option list which supports a variety of options for positioning, scaling, and rotating the imported page. Details regarding these options are discussed in Section 8.4, »Placing Images, Graphics, and imported PDF Pages«, page 213.

8.3.3 Document and Page-related Checks

Document-related checks. PDFlib+PDI happily processes all kinds of PDF documents which can be opened with Acrobat, regardless of PDF version number or features used within the file.

PDFlib+PDI implements a repair mode for damaged PDFs so that even certain kinds of damaged documents can be opened. However, in rare cases a PDF document or a particular page of a document may be rejected by PDI.

If a PDF document or page can't be imported successfully *PDF_open_pdi_document()* and *PDF_open_pdi_page()* return an error code. If you need to know more details about the failure you can query the reason with *PDF_get_errmsg()*. Alternatively, you can set the *errorpolicy* option to *exception*, which will result in an exception if the document cannot be opened.

Page-related checks. The following checks are done in *PDF_open_pdi_page()*:

- Pages from PDF documents which use a higher PDF version number than the PDF output document that is currently being generated can not be imported. The reason is that PDFlib can no longer make sure that the output will actually conform to the requested PDF version after a PDF with a higher version number has been imported. Solution: set the version of the output PDF to the required level using the *compatibility* option in *PDF_begin_document()*.
  PDF 1.7ext 3 (Acrobat 9) and *PDF 1.7ext8* (Acrobat X/XI) documents are compatible with PDF 1.7 as far as PDI is concerned.
  In PDF/A mode the input PDF version number is not relevant since the PDF version header must be ignored in PDF/A.
  If a document uses a higher PDF version header although it is known to conform to an older PDF version you can use the *ignorepdfversion* option of *PDF_open_pdi_document()*.
- PDF/A, PDF/X, PDF/VT or PDF/UA documents which are incompatible to the corresponding PDF/A, PDF/X, PDF/VT or PDF/UA status of the current output document are rejected. See the following sections for more information:
  - Section 12.3.7, »Importing PDF/A Documents with PDI«, page 326;
  - Section 12.4.6, »Importing PDF/X Documents with PDI«, page 338;
  - Section 12.5.7, »Importing PDF/X and PDF/VT Documents with PDI«, page 347;
  - Section 12.6.4, »Importing PDF/UA Documents with PDI«, page 352.
- If the document contains an inconsistent PDF/A or PDF/X output intent no pages can be imported.

8.3.4 Specific Aspects of imported PDF Documents

Dimensions of imported PDF pages. Imported PDF pages are handled similarly to imported raster images, and can be placed on the output page using *PDF_fit_pdi_page()*. By default, PDI imports 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 65);
- rotation which has been applied to the page is retained.

The *cloneboxes* option instructs PDFlib+PDI to copy all page boxes of the imported page to the generated output page, effectively cloning all page size aspects.

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

Finally, you can use the *transform* option of *PDF_open_pdi_page( )* to apply arbitrary transformations to the imported page, e.g. scale it up or down or rotate it.

Color handling. PDFlib+PDI does not change the color of imported PDF documents in any way. For example, if a PDF contains ICC color profiles these are retained in the output document. If a page contains a transparency group entry it will be copied to the generated Form XObject unless the *transparencygroup* option of *PDF_open_pdi_page( )* asks for different treatment. If required, a suitable transparency group is created for the imported page, taking into account all applicable PDF/A, PDF/X, and PDF/VT requirements.

Importing pages from Tagged PDF documents. By default, tags are imported if both input and output document are tagged. However, tag import can be disabled with the *usetags* options of *PDF_open_pdi_document( )* and *PDF_open_pdi_page( )*. For more information refer to Section 11.4.5, »Importing Tagged PDF Pages with PDI«, page 310.

Importing PDF pages with layers. PDI always imports the contents of all layers (technically known as *optional content)* on a page. The layer definitions including the layers' visibility state are also imported provided the layer is used on any of the imported pages. However, import of layer definitions can be disabled with the *uselayers* options of *PDF_open_pdi_document( )*. In order to work with *uselayers=false* the generated document must not contain any layers at all, i.e. all PDF documents with layers must be opened with *uselayers= false* and *PDF_define_layer( )* must not be called.

For more control for arranging the imported layers you can use the *parenttitle* of *PDF_open_pdi_document( )* which creates a hierarchical title layer in the layer list on top of the imported layers (e.g. for supplying a file name). The *parentlayer* option works similarly but expects a handle to a user-defined layer.

Importing georeferenced PDF. When importing georeferenced PDF with PDI the geospatial information is kept if it has been created with one of the following methods (image-based geospatial reference):

- with PDFlib and the *georeference* option of *PDF_load_image( )*
- by importing an image with geospatial information in Acrobat.

The geospatial information is lost after importing a page if it has been created with one of the following methods (page-based geospatial reference):

- with PDFlib and the *viewports* option of *PDF_begin/end_page_ext( )*
- by manually geo-registering a PDF page in Acrobat.

Optimization across multiple imported documents. While PDFlib itself creates highly optimized PDF output, imported PDF may contain redundant data structures which in some cases can be optimized. In addition, importing multiple PDFs may bloat the output file size if multiple files contain identical resources, e.g. fonts. In this situation you can use the *optimize* option of *PDF_begin_document( )*. It detects redundant objects in imported files and removes them without affecting the visual appearance or quality of the generated output.

Encrypted PDF documents and the »shrug« feature. In order to import pages from encrypted documents (i.e., files with permission settings or password) the corresponding master password must be supplied. Encrypted PDF documents without the master password are rejected by default. However, they can be opened for querying information with pCOS (as opposed to importing pages) by setting the *infomode* option of *PDF_open_pdi_document( )* to *true* (exception to the *infomode* rule: documents created with the Distiller setting *Object Level Compression: Maximum* cannot be opened even in info mode).

With the *shrug* feature pages from protected documents can be imported without master password, assuming the user accepts responsibility for respecting the document author's rights. By using the *shrug* feature the user asserts that he or she does not violate any document authors' rights. PDFlib GmbH's terms and conditions require that users respect document author's rights.

If all of the following conditions are true, the *shrug* feature is enabled:

- The *shrug* option has been supplied to *PDF_open_pdi_document( )*.
- The document requires a master password but it has not been supplied to *PDF_open_pdi_document( )*.
- If the document requires a user (open) password, it must have been supplied to *PDF_open_pdi_document( )*.

The *shrug* feature has the following effects:

- Pages can be imported although the master password has not been supplied.
- The pCOS pseudo object *shrug* is set to *true*/1.
- pCOS runs in full mode (instead of restricted mode), i.e. the *pcosmode* pseudo object is set to 2.

8.4 Placing Images, Graphics, and imported PDF Pages

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

All samples in this section work the same for raster images, templates, graphics 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( )*, *PDF_load_graphics( )* or *PDF_open_pdi_document( )* and *PDF_open_pdi_page( )* must be issued. For the sake of simplicity these calls are not repeated here.

Cookbook Code samples regarding images, graphics and imported PDF pages can be found in the `images`, `graphics`, *and* `pdf_import` *categories of the PDFlib Cookbook.*

8.4.1 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 *(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.

Cookbook A full code sample can be found in the Cookbook topic `images/starter_image`.

8.4.2 Placing an Object at a Point or Line or in a Box

In order to position an object an additional box with specified width and height can be used. The gray box or line in the figures below is depicted for visualizing the box size only; it is not part of the actual output.

Placing an object in a box doesn't make sense for *fitmethod=nofit* since the object is only positioned in this case, but not scaled. The *boxsize* option can be used to specify a horizontal line, vertical line or real box for object placement:

```
boxsize={100 0}          horizontal line
boxsize={0 100}          vertical line
boxsize={100 200}        box
```

In the examples below we will fit the object into the box with various fitting methods.

Automatic fitting in a box. With *fitmethod=auto* PDFlib scales the image to fit into the box without distorting it: if it fits into the box no scaling is applied. Otherwise the size is reduced while preserving the aspect ratio of width and height. Figure 8.2a, Figure 8.2b, and Figure 8.2c demonstrate how PDFlib reduces the image size as the size of the fitbox decreases from initially *boxsize={70 45}* to *boxsize={70 30}* and further to *boxsize={30 30}*.

Fig. 8.2 Fitting an image into a box subject to various fit methods

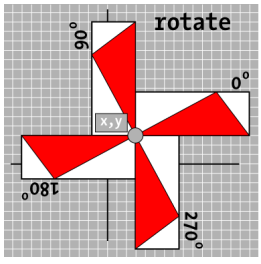
Generated output	*Option list for PDF_fit_image()*
a) 	`boxsize={70 45} position=center fitmethod=auto` `(no scaling necessary)`
b) 	`boxsize={70 30} position=center fitmethod=auto` `(scaled down to adjust for smaller box height)`
c) 	`boxsize={30 30} position=center fitmethod=auto` `(scaled down to adjust for smaller box height and width)`
d) 	`boxsize={70 45} position=center fitmethod=meet`
e) 	`boxsize={35 45} position=center fitmethod=meet`
f) 	`boxsize={70 45} position=center fitmethod=entire`
g) 	`boxsize={30 30} position=center fitmethod=clip`
h) 	`boxsize={30 30} position={right top} fitmethod=clip`

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 8.2d).

Decreasing the box width from 70 to 35 units forces PDFlib to scale down the image until its width completely fits into the box (see Figure 8.2e).

With *fitmethod=meet* it is guaranteed that the image is not distorted and that it is placed in the box as large as possible.

Fig. 8.3
The rotate option



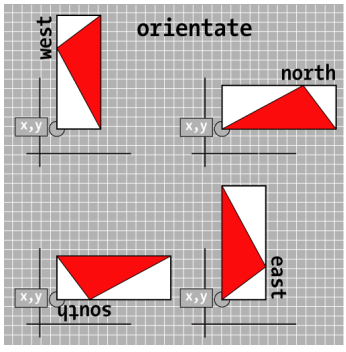


Fig. 8.4
The orientate option

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 may be distorted (see Figure 8.2f).

Clipping an image when fitting it into a 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 8.2g).

By positioning the image at the center of the box, the image will be cropped evenly on all sides. Similarly, to completely show the upper right part of the image you can position it with *position={right top}* (see Figure 8.2h).

8.4.3 Orientating an Object

Placing an image with orientation. In our next example we orientate an image towards western direction *(orientate=west)*. This means that the image is rotated by 90° counterclockwise and then the lower left corner of the rotated object is translated to the reference point *(0, 0)*. The object will be rotated in itself (see Figure 8.5a). Since we have not specified any fit method the image is output in its original size and exceeds the box.

Fitting an image proportionally into a box with orientation. Our next goal is to orientate the image to the west with a predefined size. We define a box of the desired size and fit the image into the box with the image's proportions being unchanged *(fitmethod=meet)*. The orientation is specified as *orientate=west*. By default, the image will be placed in the lower left corner of the box (see Figure 8.5b). Figure 8.5c shows the image orientated to the east, and Figure 8.5d the orientation to the south.

The *orientate* option supports the direction keywords *north, east, west,* and *south* as demonstrated in Figure 8.4.

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

Fig. 8.5 Orientating an image

Generated output	*Option list for PDF_fit_image()*
a) 	`boxsize={70 45} orientate=west`[1]
b) 	`boxsize={70 45} orientate=west fitmethod=meet`
c) 	`boxsize={70 45} orientate=east fitmethod=meet`
d) 	`boxsize={70 45} orientate=south fitmethod=meet`
e) 	`boxsize={70 45} position={center bottom} orientate=east fitmethod=clip`

1. The `boxsize` *option isn't actually required because of the default* `fitmethod=nofit`.

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 8.5e).

8.4.4 Rotating an Object

The *rotate* option rotates an object by rotating the coordinate system at the reference point. As a result the fitbox is also rotated. Figure 8.3 demonstrates the general behavior of the *rotate* option.

Placing an image with rotation. Our first goal is to rotate an image by 90° counterclockwise. Before placing the object the coordinate system is rotated at the reference point by 90° counterclockwise. The rotated object's lower right corner (which is the unrotated object's lower left corner) ends up at the reference point. This case is shown in Figure 8.6a.

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 8.6b).

Fitting an image with rotation. Our next goal is to fit the image rotated by 90° counterclockwise into the box while maintaining its proportions. This is accomplished using *fitmethod=meet* (see Figure 8.6c). Similarly, we can rotate the image by 30° counterclockwise and proportionally fit the image into the box (see Figure 8.6d).

Fig. 8.6 Rotating an image

Generated output	*Option list for PDF_fit_image()*
a) 	`boxsize={70 45} rotate=90`[1]
b) 	`boxsize={70 45} rotate=30`[1]
c) 	`boxsize={70 45} rotate=90 fitmethod=meet`
d) 	`boxsize={70 45} rotate=30 fitmethod=meet`

1. The `boxsize` *option isn't actually required because of the default* `fitmethod=nofit`.

8.4.5 Adjusting the Page Size

Adjusting the page size to an image. In the next example we will automatically adjust the page size to the object's size. This can be useful, for example, for archiving images in the PDF format. The reference point *(x, y)* can be used to specify whether the page will have exactly the object's size, or somewhat larger or smaller. When enlarging the page size (see Figure 8.7) some border will be kept around the image. If the page size is smaller than the image some parts of the image will be clipped. Let's start with exactly matching the page size to the object's size:

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

The next code fragment increases the page size by 40 units in *x* and *y* direction, creating a white border around the object:

```
p.fit_image(image, 40, 40, "adjustpage");
```

The next code fragment decreases the page size by 40 units in *x* and *y* direction. The object will be clipped at the page borders, and some area within the object (with a width of 40 units) will be invisible:

```
p.fit_image(image, -40, -40, "adjustpage");
```

In addition to placing by means of *x* and *y* coordinates (which specify the object's distance from the page edges, or the coordinate axes in the general case) you can also spec-

Fig. 8.7 Adjusting the page size. Left to right: exact, enlarge, shrink

ify a target box. This is a rectangular area in which the object will be placed subject to various formatting rules. These can be controlled with the *boxsize*, *fitmethod* and *position* options.

Cloning the page boxes of an imported PDF page. You can copy all page boxes (MediaBox, CropBox) etc. of an imported PDF page to the current output page. The *cloneboxes* option must be supplied to *PDF_open_pdi_page( )* to read all relevant box values, and again in *PDF_fit_pdi_page( )* to apply the box values to the current page:

```
/* Open the page and clone the page box entries */
inpage = p.open_pdi_page(indoc, 1, "cloneboxes");
...
/* Start the output page with a dummy page size */
p.begin_page_ext(10, 10, "");
...
/*
* Place the imported page on the output page, and clone all
* page boxes which are present in the input page; this will
* override the dummy size used in begin_page_ext().
*/
p.fit_pdi_page(inpage, 0, 0, "cloneboxes");
```

Using this technique you can make sure that the pages in the generated PDF will have the exact same page size, cropping etc. as the pages of the imported document. This is especially important for prepress applications.

8.4.6 Querying Information about placed Images and PDF Pages

Information about placed images and templates. The *PDF_info_image( )* function can be used to query image and template information. The supported keywords for this function cover general image information (e.g. width and height in pixels) as well as geometry information related to placing the image on the output page (e.g. width and height in absolute values after performing the fitting calculations).

The following code fragment retrieves both the pixel size and the absolute size after placing an image with certain fitting options:

```
String optlist = "boxsize={300 400} fitmethod=meet orientate=west";
p.fit_image(image, 0.0, 0.0, optlist);

imagewidth  = (int) p.info_image(image, "imagewidth", optlist);
imageheight = (int) p.info_image(image, "imageheight", optlist);
System.err.println("image size in pixels: " + imagewidth + " x " + imageheight);

width  = p.info_image(image, "width", optlist);
height = p.info_image(image, "height", optlist);
System.err.println("image size in points: " + width + " x " + height);
```

Information about placed PDF pages. The *PDF_info_pdi_page()* function can be used to query information about placed PDF pages. The supported keywords for this function cover information about the original page (e.g. its width and height) as well as geometry information related to placing the imported PDF on the output page (e.g. width and height after performing the fitting calculations).

The following code fragment retrieves both the original size of the imported page and the size after placing the page with certain fitting options:

```
String optlist = "boxsize={400 500} fitmethod=meet";
p.fit_pdi_page(page, 0, 0, optlist);

pagewidth  = p.info_pdi_page(page, "pagewidth", optlist);
pageheight = p.info_pdi_page(page, "pageheight", optlist);
System.err.println("original page size: " + pagewidth + " x " + pageheight);

width  = p.info_pdi_page(page, "width", optlist);
height = p.info_pdi_page(page, "height", optlist);
System.err.println("size of placed page: " + width + " x " + height);
```

9 Text and Table Formatting

9.1 Placing and Fitting Textlines

The function *PDF_fit_textline( )* for placing a single line of text on a page offers a wealth of formatting options. The most important options will be discussed in this section using some common application examples. A complete description of these options can be found in the PDFlib API Reference. Most options for *PDF_fit_textline( )* are identical to those of *PDF_fit_image( )*. Therefore we will only use text-related examples here; it is recommended to take a look at the examples in Section 8.4, »Placing Images, Graphics, and imported PDF Pages«, page 213, for an introduction to image formatting.

The examples below demonstrate only the relevant call of *PDF_fit_textline( )*, assuming that the required font has already been loaded and set in the desired font size.

PDF_fit_textline( ) uses a hypothetical text box to determine the positioning of the text: the width of the text box is identical to the width of the text, and the box height is identical to the height of capital letters in the font. The text box can be modified by the *matchbox* option.

In the examples below, the coordinates of the reference point are supplied as *x, y* parameters of *PDF_fit_textline( )*. The fitbox for text lines is the area where text will be placed. It is defined as the rectangular area specified with the *x, y* parameters of *PDF_fit_textline( )* and appropriate options *(boxsize, fitmethod, position, rotate)*. The fitbox can be reduced to the left/right or top/bottom with the *margin* option.

Cookbook *Code samples regarding text output issues can be found in the* `text_output` *category of the PDFlib Cookbook.*

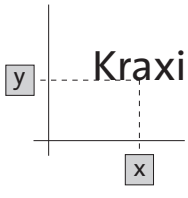
9.1.1 Simple Textline Placement

Positioning text at the reference point. By default, the text will be placed with the lower left corner at the reference point. However, in this example we want to place the text with the bottom centered at the reference point. The following code fragment places the text box with the bottom centered at the reference point *(30, 20)*.

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

Figure 9.1 illustrates centered text placement. Similarly, you can use the *position* option with another combination of the keywords *left, right, center, top*, and *bottom* to place text at the reference point.

Fig. 9.1
Centered text



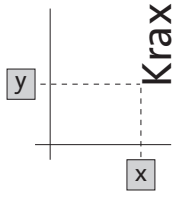


Fig. 9.2
Simple text with orientation west

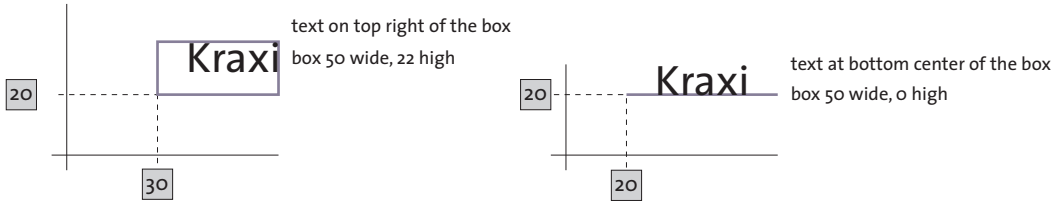


Fig. 9.3 Positioning text in a box

Placing text with orientation. Our next goal is to rotate text while placing its lower left corner (after the rotation) at the reference point. The following code fragment orientates the text to the west (90° counterclockwise) and then translates the lower left corner of the rotated text to the reference point *(0, 0)*.

```
p.fit_textline(text, 0, 0, "orientate=west");
```

Figure 9.2 illustrates simple text placement with orientation.

9.1.2 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 9.3 illustrates the general behavior

Positioning text in the box. We define a rectangular box and place the text within this box on the top right. The following code fragment defines a box with a width of 50 units and a height of 22 units at reference point *(30, 20)*. In Figure 9.4a, 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 9.4b.

To achieve some distance between the text and the box we can add the *margin* option (see Figure 9.4c).

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

Fig. 9.4 Placing text in a box subject to various positioning options

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

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

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 9.4d 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 9.4e.

9.1.3 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 9.5a).

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 9.5b).

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 option *fitmethod=auto*. In Figure 9.5c the box is wide enough to keep the text in its original size completely so that the text is placed 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 tries to condense the text horizontally, subject to the *shrinklimit* option (default: 0.75). Figure 9.5d shows the text shrunk to 75 percent of its original length.

When reducing the box width further to 30 units the text will not fit even with shrinking. Then the *meet* method is applied. It decreases the font size until the text fits completely into the box. This case is shown in Figure 9.5e.

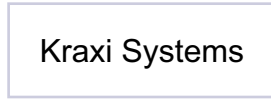
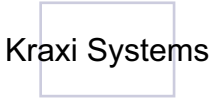
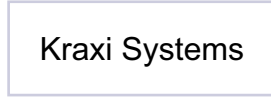
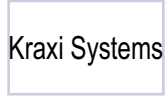
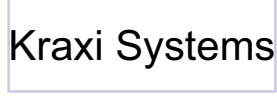
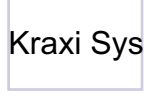
Fitting the text into the box with increased font size. You might want to fit the text so that it covers the whole width (or height) of the box but maintains its proportions. Using *fitmethod=meet* with a box larger than the text, the text will be increased until its width matches the box width. This case is illustrated in Figure 9.5f.

Completely fitting text into a box. We can further fit the text so that it completely fills the box. In this case, *fitmethod=entire* is used. However, this combination will rarely be used since the text will most probably be distorted (see Figure 9.5g).

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 9.5h 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. 9.5 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`

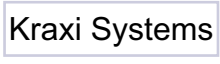
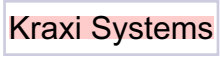
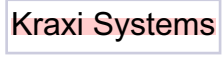
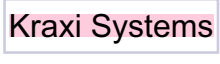
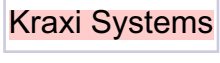
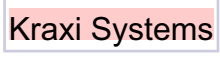
Vertically centering text. The text height in *PDF_fit_textline( )* is the capheight, i.e. the height of the capital letter *H*, by default. If the text is positioned in the center of a box it will be vertically centered according to its capheight (see Figure 9.6a).

To specify another height for the text box we can use the Matchbox feature (see also Section 9.4, »Matchboxes«, page 267). 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 9.6b). Figure 9.6c vertically centers the text according to the *xheight* by defining a matchbox with a corresponding box height.

Figure 9.6d–f shows the matchbox (red) with various useful *boxheight* settings to determine the height of the text to be centered in the box.

Fig. 9.6 Fitting text proportionally into a box according to different box heights

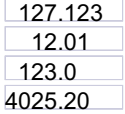
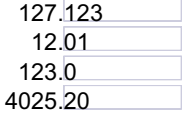
Generated output		*Option list for PDF_fit_textline( )*
a)		`boxsize={80 20} position=center fitmethod=auto`
b)		`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={capheight none} fillcolor=mistyrose}`
c)		`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={xheight none} fillcolor=mistyrose}`
d)		`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={ascender none} fillcolor=mistyrose}`
e)		`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={ascender descender} fillcolor=mistyrose}`
f)		`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={fontsize none} fillcolor=mistyrose}`

9.1.4 Aligning Text at a Character

You might want to align text at a certain character, e.g. at the decimal point in a number. As shown in Figure 9.7a, the text is positioned at the center of the fitbox. Using *PDF_fit_textline( )* with the *alignchar*=. option the numbers are aligned at the dot character.

You can omit the *position* option which places the dots in the center of the box. In this case, the default *position={left bottom}* will be used which places the dots at the reference point (see Figure 9.7b). In general, the alignment character will be placed with the lower right corner at the reference point.

Fig. 9.7 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=.`

9.1.5 Placing a Stamp

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

As an alternative to rotated text, the stamp feature offers a convenient method for placing text diagonally in a box. The stamp function will automatically perform some sophisticated calculations to determine a suitable font size and rotation so that the text covers the box. To place a diagonal stamp, e.g. in the page background, use *PDF_fit_textline( )* with the *stamp* option. With *stamp=ll2ur* the text is placed from the lower left to the upper right corner of the fitbox. However, with *stamp=ul2lr* the text is placed from the upper left to the lower right corner of the fitbox. The *fitbox* option is ignored. As shown in Figure 9.8, *showborder=true* is used to illustrate the fitbox and the bounding box of the stamp.

Fig. 9.8 Fitting a text line like a stamp from the lower left to the upper right

Generated output	***Option list for PDF_fit_textline( )***
	`fontsize=8 boxsize={160 50} stamp=ll2ur showborder=true`

9.1.6 Using Leaders

Leaders can be used to fill the space between the borders of the fitbox and the text. For example, dot leaders are often used as a visual aid between the entries in a table of contents and the corresponding page numbers.

Leaders in a table of contents. Using *PDF_fit_textline( )* with the *leader* option and the *alignment={none right}* suboption, leaders are appended to the right of the text line, and repeated until the right border of the text box. There will be an equal distance between the rightmost leader and the right border, while the distance between the text and the leftmost leader may differ (see Figure 9.9a).

Cookbook *A full code sample demonstrating the usage of dot leaders in a text line can be found in the Cookbook topic* `text_output/leaders_in_textline`.

Cookbook *A full code sample demonstrating the usage of dot leaders in a Textflow can be found in the Cookbook topic* `textflow/dot_leaders_with_tabs`.

Leaders in a news ticker. In another use case you might want to create a news ticker effect. In this case we use a plus and a space character »+ « as leaders. The text line is placed in the center, and the leaders are printed before and after the text line *(alignment={left right})*. The left and right leaders are aligned to the left and right border, and might have a varying distance to the text (see Figure 9.9b).

Fig. 9.9 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={+ }}`

9.1.7 Text on a Path

Instead of placing text on a straight line you can also place text on an arbitrary path. PDFlib will place the individual characters along the path so that the text follows the curvature of the path. Use the *textpath* option of *PDF_fit_textline( )* to create text on a path. The path must have been created earlier and is represented by a path handle. Path handles can be created by explicitly constructing a path with *PDF_add_path_point( )* and related path object functions, or by retrieving a handle for the clipping path in an existing raster image. The following code fragment creates a simple path and places text on the path (see Figure 9.10):

```
/* Define the path in the origin */
path = p.add_path_point(  -1,   0,   0, "move", "");
path = p.add_path_point(path, 100, 100, "control", "");
path = p.add_path_point(path, 200,   0, "circular", "");

/* Place text on the path */
p.fit_textline("Long Distance Glider with sensational range!", x, y,
    "textpath={path=" + path + "} position={center bottom}");

/* We also draw the path for demonstration purposes */
p.draw_path(path, x, y, "stroke strokecolor=dodgerblue");
```

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

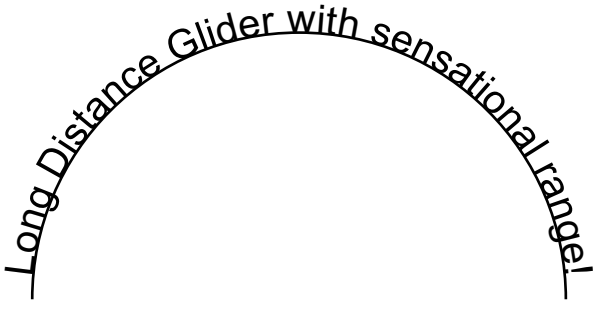


Fig. 9.10
Text on a path

Using an image clipping path for placing text. As an alternative to manually constructing a path object with the path functions you can extract the clipping path from an image and place text on the resulting path. The image must have been loaded with the *honorclippingpath* option, and the *clippingpathname* option must also be supplied to *PDF_load_image()* if the target path is not the image's default clipping path:

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

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

/* Place text on the path */
p.fit_textline("Long Distance Glider with sensational range!", x, y,
    "textpath={path=" + path + "} position={center bottom}");
```

Creating a gap between path and text. By default, PDFlib places individual characters directly on the path, i.e. there is no space between the glyphs and the path. If you want to create a gap between the path and the text you can increase the character boxes. This can be achieved with *boxheight* suboption of the *matchbox* option which specifies the vertical extension of the character boxes. The following option list takes the descenders into account (see Figure 9.11):

```
p.fit_textline("Long Distance Glider with sensational range!", x, y,
    "textpath={path=" + path + "} position={center bottom} " +
    "matchbox={boxheight={capheight descender}}");
```

9.1.8 Shadowed Text

The *shadow* option can be used to create a shadow effect for text. You can specify the shadow color as well as its horizontal and vertical distance from the main text in suboptions:

```
p.fit_textline("Long Distance Glider", x, y,
    "fillcolor=rosybrown shadow={offset={3, -3}}");
```

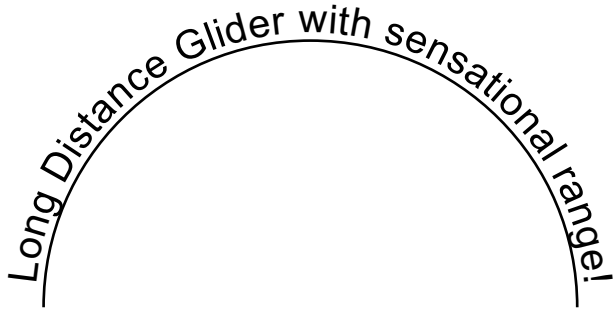


Fig. 9.11
Text on a path with an additional gap between text and path

9.1.9 Watermarks which can be edited in Acrobat

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

The Textline functionality can be used to apply a variety of formatting options to text. The text becomes an integral part of the page and cannot easily be modified in the final PDF document. However, Acrobat offers a »Watermark« feature for page content which can later be modified or deleted in Adobe Acrobat (but not the free Reader). Note that this feature is not part of the PDF standard ISO 32000, but is a private extension implemented in Acrobat. This feature is therefore not guaranteed to work in all standard-conforming PDF viewers.

Acrobat watermarks work by wrapping one or more lines of text, an image, or a PDF page within a template (Form XObject) and adding an XML description of the watermark's formatting properties. Proceed as follows to edit an existing watermark in Acrobat:

- Acrobat DC: click *Tools, Edit PDF, Watermark, Update...* or *Remove...* .
- Acrobat XI: click *Tools, Pages, Edit Page Design, Watermark, Update...* or *Remove...* .

This brings up the Watermark dialog (see Figure 9.12) where the watermark text, its placement and appearance as well as the set of target pages can be specified. The text appearance may change after using this dialog since the dialog does not support the full range of text formatting options, e.g. character spacing, underline, text rendering mode etc.

The watermark can be specified to appear on the screen display, printed page, or both *(Appearance Options...* in the Acrobat dialog). This is achieved via a layer called *Watermark* with the appropriate screen and print settings. Note that layers and layer options are subject to various restrictions in PDF/A, PDF/X and PDF/UA.

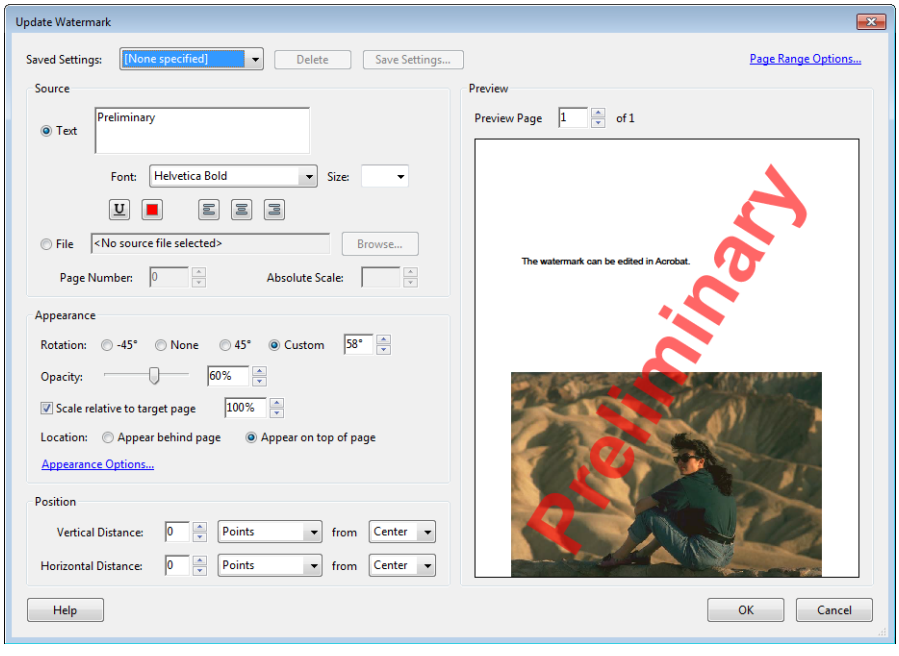


Fig. 9.12 Acrobat's Watermark dialog

PDFlib supports the watermark feature with an extension to the template functionality. When a template is created with *PDF_begin_template_ext( )* it can be marked with the *watermark* option. In this case the template must only contain a single line of text which has been created with *PDF_fit_textline( )*. The resulting watermark text is automatically added to all pages (or optionally a subset of pages), and can later be modified or removed with Acrobat.

Note Watermarks containing multiple lines, images, or PDF pages are not currently supported.

The following code fragment demonstrates the definition of an editable watermark. The watermark is automatically added to all pages which are created after defining the watermark. Note that the specified font size is irrelevant since the resulting template is fitted in the page anyway:

```
p.begin_template_ext(0, 0, "watermark={location=ontop opacity=60%}");

p.fit_textline("Preliminary", 0, 0,
        "fontsize=12 fontname=Helvetica-Bold encoding=unicode fillcolor=red " +
        "boxsize={595 842} stamp=ll2ur");

p.end_template_ext(0, 0);
```

9.2 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 9.13 and Figure 9.14 demonstrate how various parts of an invoice can be placed on the page using the Textflow feature. We will discuss the options for controlling the output in more detail in the following sections.

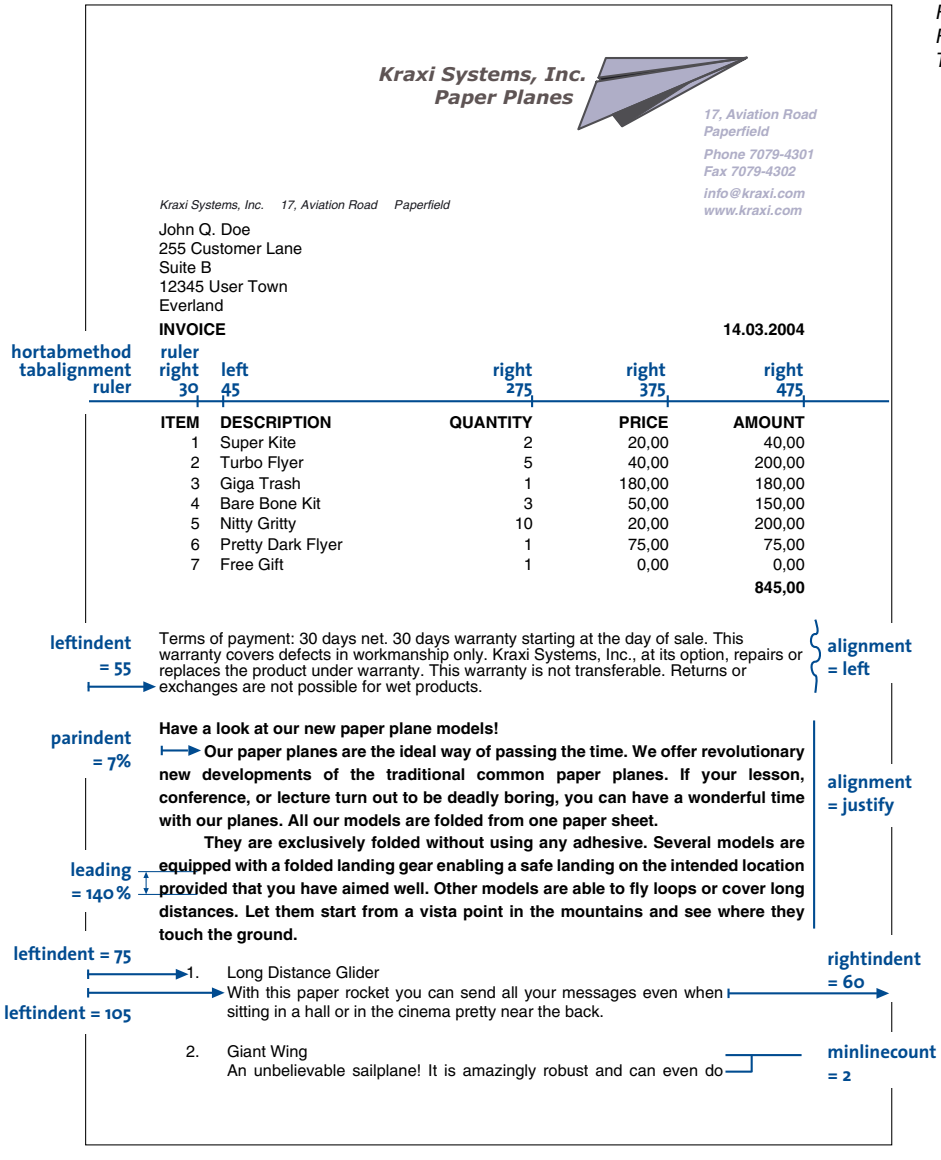


Fig. 9.13 Formatting Textflows

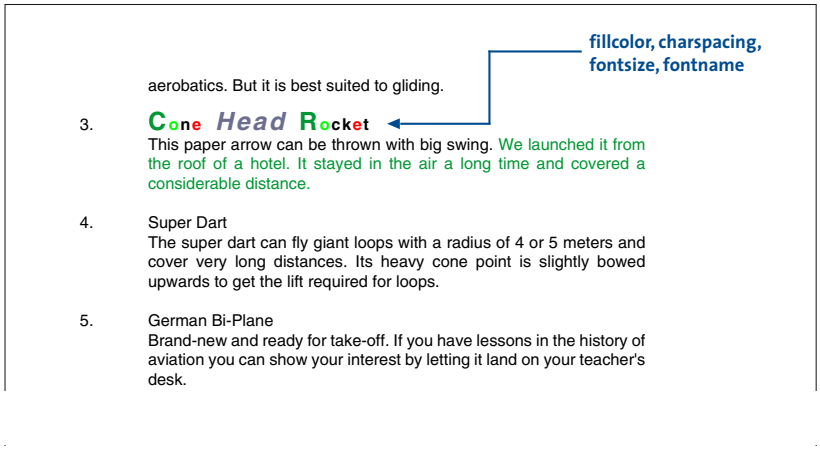


Fig. 9.14 Formatting Textflows

A multi-line Textflow can be placed into one or more rectangles (so-called fitboxes) on one or more pages. The following steps are required for placing a Textflow on the page:

- The function *PDF_add_textflow( )* accepts portions of text and corresponding formatting options, creates a Textflow object, and returns a handle. As an alternative, the function *PDF_create_textflow( )* analyzes the complete text in a single call, where the text may contain inline options for formatting control. These functions do not place any text on the page.
- The function *PDF_fit_textflow( )* places all or parts of the Textflow in the supplied fitbox. To completely place the text, this step must possibly be repeated several times where each of the function calls provides a new fitbox which may be located on the same or another page.
- The function *PDF_delete_textflow( )* deletes the Textflow object after it has been placed in the document.

The functions *PDF_add/create_textflow( )* for creating Textflows support a variety of options for controlling the formatting process. These options can be provided in the function's option list, or embedded as *inline* options in the text when using *PDF_create_textflow( )*. *PDF_info_textflow( )* can be used to query formatting results and many other Textflow details. We will discuss Textflow placement using some common application examples. A complete list of Textflow options can be found in the *PDFlib API Reference*.

Many of the options supported by *PDF_add/create_textflow( )* are identical to those of *PDF_fit_textline( )*. It is therefore recommended to familiarize yourself with the examples in Section 9.1, »Placing and Fitting Textlines«, page 221. In the below sections we will focus on options related to multi-line text.

Cookbook *Code samples regarding text output issues can be found in the* `text_output` *category of the PDFlib Cookbook.*

9.2.1 Placing Textflows in the Fitbox

The fitbox for Textflow is the area where text will be placed. It is defined as the rectangular area specified with the *llx, lly, urx, ury* parameters of *PDF_fit_textflow( )*.

Placing text in a single fitbox. Let's start with an easy example. The following code fragment uses two calls to *PDF_add_textflow( )* to assemble a piece of bold text and a

piece of normal text. Font, font size, and encoding are specified explicitly. In the first call to *PDF_add_textflow( )*, -1 is supplied, and the Textflow handle will be returned to be used in subsequent calls to *PDF_add_textflow( )*, if required. *text1* and *text2* are assumed to contain the actual text to be printed.

With *PDF_fit_textflow( )*, the resulting Textflow is placed in a fitbox on the page using default formatting options.

```
/* Add text with bold font */
tf = p.add_textflow(-1, text1, "fontname=Helvetica-Bold fontsize=9 encoding=unicode");
if (tf == -1)
        throw new Exception("Error: " + p.get_errmsg());

/* Add text with normal font */
tf = p.add_textflow(tf, text2, "fontname=Helvetica fontsize=9 encoding=unicode");
if (tf == -1)
        throw new Exception("Error: " + p.get_errmsg());

/* Place all text */
result = p.fit_textflow(tf, left_x, left_y, right_x, right_y, "");
if (!result.equals("_stop"))
        { /* ... */}

p.delete_textflow(tf);
```

Placing text in two fitboxes on multiple pages. If the text placed with *PDF_fit_textflow( )* doesn't completely fit into the fitbox, the output will be interrupted and the function will return the string *_boxfull*. PDFlib will remember the amount of text already placed, and will continue with the remainder of the text when the function is called again. In addition, it may be necessary to create a new page. The following code fragment demonstrates how to place a Textflow in two fitboxes per page on one or more pages until the text has been placed completely (see Figure 9.15).

Cookbook *A full code sample can be found in the Cookbook topic* `textflow/starter_textflow`.

```
/* Loop until all of the text is placed; create new pages as long as more text needs
 * to be placed. Two columns will be created on all pages.
*/
```

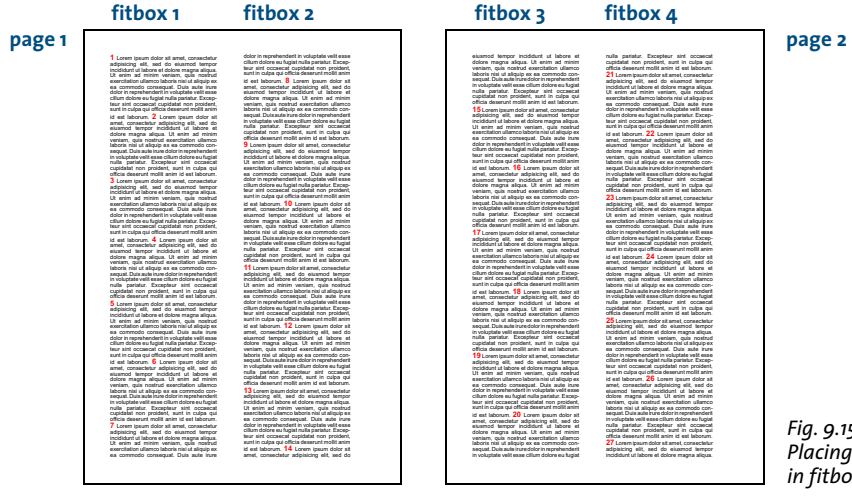


Fig. 9.15
Placing a Textflow in fitboxes

```
do
{
        String optlist = "verticalalign=justify linespreadlimit=120%";

        p.begin_page_ext(0, 0, "width=a4.width height=a4.height");

        /* Fill the first column */
        result = p.fit_textflow(tf, llx1, lly1, urx1, ury1, optlist);

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

        p.end_page_ext("");

        /* "_boxfull" means we must continue because there is more text;
         * "_nextpage" is interpreted as "start new column"
         */
} while (result.equals("_boxfull") || result.equals("_nextpage"));

/* Check for errors */
if (!result.equals("_stop"))
{
        /* "_boxempty" happens if the box is very small and doesn't hold any text at all.
        */
        if (result.equals( "_boxempty"))
            throw new Exception("Error: " + p.get_errmsg());
        else
        {
            /* Any other return value is a user exit caused by the "return" option;
             * this requires dedicated code to deal with.
             */
        }
}
p.delete_textflow(tf);
```

9.2.2 Paragraph Formatting Options

In the previous example we used default settings for the paragraphs. For example, the default alignment is left-justified, and the leading is 100% (which equals the font size).

In order to fine-tune the paragraph formatting we can feed more options to *PDF_add_textflow()*. For example, we can indent the text 15 units from the left and 10 units from the right margin. The first line of each paragraph should be indented by an additional 20 units. The text should be justified against both margins, and the leading increased to 140%. Finally, we'll reduce the font size to 8 points. To achieve this, extend the option list for *PDF_add_textflow()* as follows (see Figure 9.16):

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

9.2.3 Inline Option Lists and Macros

The text in Figure 9.16 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.

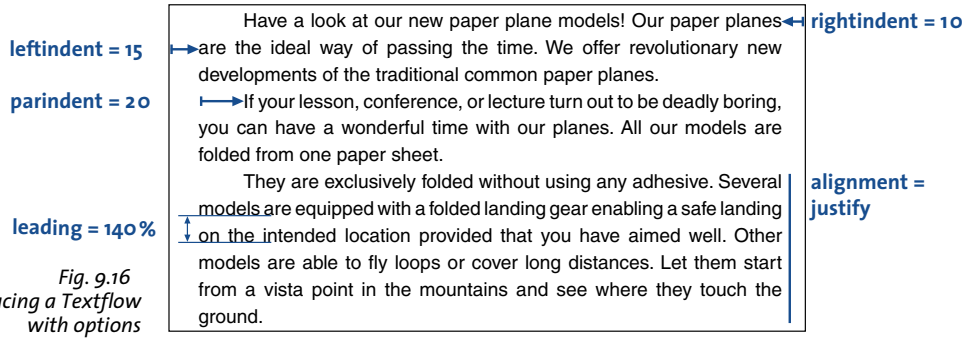


Fig. 9.16 Placing a Textflow with options

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
```

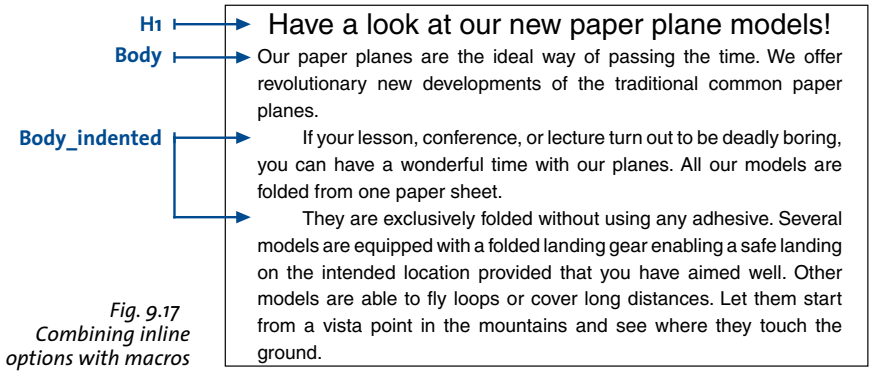


Fig. 9.17 Combining inline options with macros

```
landing on the intended location provided that you have aimed well.
Other models are able to fly loops or cover long distances. Let them
start from a vista point in the mountains and see
where they touch the ground.
```

The characters for bracketing option lists can be redefined with the *begoptlistchar* and *endoptlistchar* options. Supplying the keyword *none* for the *begoptlistchar* option completely disables the search for option lists. This is useful if the text doesn't contain any inline option lists, and you want to make sure that »<« and »>« will be processed as regular characters.

Symbol characters and inline option lists. Symbolic characters can be used for Textflow even in combination with inline option lists. The code for the character which introduces an inline option list (by default: '<' U+003C) will not be interpreted as a symbol code within text for a font with *encoding=builtin*. In order to select the symbol glyph with the same code, the same workarounds can be used which are available for text fonts, i.e. by redefining the start character with the *begoptlistchar* option or by using the *textlen* option to specify the number of symbolic glyphs. Note that character references (e.g. *<)* can not be used as a workaround.

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 9.17 we define three macros called *H1* for the heading, *Body* for main paragraphs, and *Body_indented* for indented paragraphs. In order to use a macro we place the & character in front of its name and put it into an option list. The following code fragment defines three macros according to the previously used inline options and uses them in the text:

```
<macro {
H1 {leftindent=15 rightindent=10 alignment=center
fontname=Helvetica fontsize=12 encoding=winansi}

Body {leftindent=15 rightindent=10 alignment=justify leading=140%
fontname=Helvetica fontsize=8 encoding=winansi}

Body_indented {parindent=20 leftindent=15 rightindent=10 alignment=justify
leading=140% fontname=Helvetica fontsize=8 encoding=winansi}
}>
<&H1>Have a look at our new paper plane models! ↵
<&Body>Our paper planes are the ideal way of passing the time. We offer
revolutionary new developments of the traditional common paper planes. ↵
<&Body_indented>If your lesson, conference, or lecture
turn out to be deadly boring, you can have a wonderful time
with our planes. All our models are folded from one paper sheet. ↵
They are exclusively folded without using any adhesive. Several
models are equipped with a folded landing gear enabling a safe
landing on the intended location provided that you have aimed well.
Other models are able to fly loops or cover long distances. Let them
start from a vista point in the mountains and see
where they touch the ground.
```

Explicitly setting options. Note that all options which are not set in macros will retain their previous values. In order to avoid side effects caused by unwanted »inheritance« of options you should explicitly specify all settings required for a particular macro. This way you can ensure that the macros will behave consistently regardless of their ordering or combination with other option lists.

On the other hand, you can take advantage of this behavior for deliberately retaining certain settings from the context instead of supplying them explicitly. For example, a macro could specify the font name without supplying the *fontsize* option. As a result, the font size will always match that of the preceding text.

Inline options or options passed as function parameters? When using Textflows it makes an important difference whether the text is contained literally in the program code or comes from some external source, and whether the formatting instructions are separate from the text or part of it. In most applications the actual text will come from some external source such as a database. In practise there are two main scenarios:

- Text contents from external source, formatting options in the program: An external source delivers small text fragments which are assembled within the program, and combined with formatting options (in the function call) at runtime.
- Text contents and formatting options from external source: Large amounts of text including formatting options come from an external source. The formatting is provided by inline options in the text, represented as simple options or macros. When it comes to macros a distinction must be made between macro definition and macro call. This allows an interesting intermediate form: the text content comes from an external source and contains macro calls for formatting. However, the macro definitions are only blended in at runtime. This has the advantage that the formatting can easily be changed without having to modify the external text. For example, when generating greeting cards one could define different styles via macros to give the card a romantic, technical, or other touch.

9.2.4 Tab Stops

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

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

To place that simple table use the following option list in *PDF_add/create_textflow( )*. The *ruler* option defines the tab positions, *tabalignment* specifies the alignment of tab stops, and *hortabmethod* specifies the method used to process tab stops (the result can be seen in Figure 9.18):

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

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. 9.18 Placing text as a table

Cookbook A full code sample can be found in the Cookbook topic `textflow/tabstops_in_text`.

Note PDFlib's table feature is recommended for creating complex tables (see Section 9.3, »Table Formatting«, page 251).

9.2.5 Numbered Lists and Paragraph Spacing

The following example demonstrates how to format a numbered list using the inline option *leftindent* (see Figure 9.19):

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

Cookbook Full code samples for bulleted and numbered lists can be found in the Cookbook topics `textflow/bulleted_list` *and* `textflow/numbered_list`.

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

```
<macro {
indent {25}

list {parindent=-&indent leftindent=&indent hortabsize=&indent
hortabmethod=ruler ruler={&indent}}
}>
<&list>1. →  Long Distance Glider: With this paper rocket you can send all your messages
even when sitting in a hall or in the cinema pretty near the back. ↵
2. →  Giant Wing: An unbelievable sailplane! It is amazingly robust and can even do
aerobatics. But it is best suited to gliding. ↵
```

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

Fig. 9.19 Numbered list

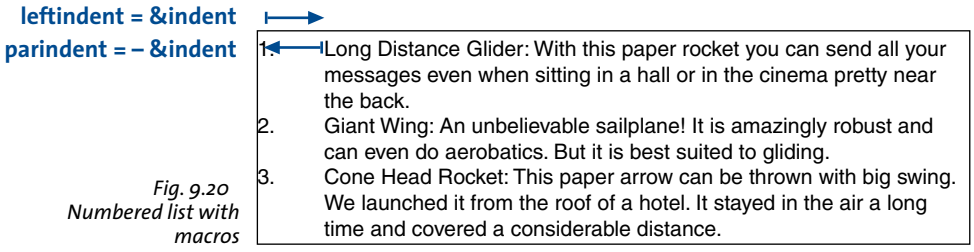


Fig. 9.20 Numbered list with macros

```
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 amount specified in *leftindent*. Figure 9.20 shows the *parindent* and *leftindent* options at work.

Setting the distance between two paragraphs. In many cases more distance between adjacent paragraphs is desired than between the lines within a paragraph. This can be achieved by inserting an extra empty line (which can be created with the *nextline* option), and specifying a suitable leading value for this empty line. This value is the distance between the baseline of the last line of the previous paragraph and the baseline of the empty line. The following example will create 80% additional space between the two paragraphs (where 100% equals the most recently set value of the font size):

```
1. → Long Distance Glider: With this paper rocket you can send all your messages
even when sitting in a hall or in the cinema pretty near the back.
<nextline leading=80%><nextparagraph leading=100%>2. → Giant Wing: An unbelievable
sailplane! It is amazingly robust and can even do aerobatics. But it is best suited to
gliding.
```

Cookbook *A full code sample can be found in the Cookbook topic* `textflow/distance_between_paragraphs`.

9.2.6 Control Characters and Character Mapping

Control characters in Textflows. Various characters are given special treatment in Textflows. PDFlib supports symbolic character names which can be used instead of the corresponding character codes in the *charmapping* option (which replaces characters in the text before processing it, see below). Table 9.1 lists all control characters which are evaluated by the Textflow functions along with their symbolic names, and explains their meaning. An option must only be used once per option list, but multiple option lists can be provided one after the other. For example, the following sequence will create an empty line:

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

Table 9.1 Control characters and their meaning in Textflows

Unicode character	*entity name*	*equiv. Textflow option*	*meaning within Textflows*
U+0020	*SP, space*	*space*	*align words and break lines*
U+00A0	*NBSP, nbsp*	*(none)*	*(no-break space) space character which will not break lines*
U+202F	*NNBSP, nnbsp*	*(none)*	*(narrow no-break space) fixed-width space character which will not break lines, and will not change its width according to formatting options*
U+0009	*HT, hortab*	*(none)*	*horizontal tab: will be processed according to the* `ruler`*,* `tabalignchar`*, and* `tabalignment` *options*
U+002D	*HY, hyphen*	*(none)*	*separator character for hyphenated words*
U+00AD	*SHY, shy*	*(none)*	*(soft hyphen) hyphenation opportunity, only visible at line breaks*
U+000B *U+2028*	*VT, 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( ) stops and returns the string* `_nextpage`.

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

```
charmapping={hortab space}
```

This command uses the symbolic character names *hortab* and *space*. 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:

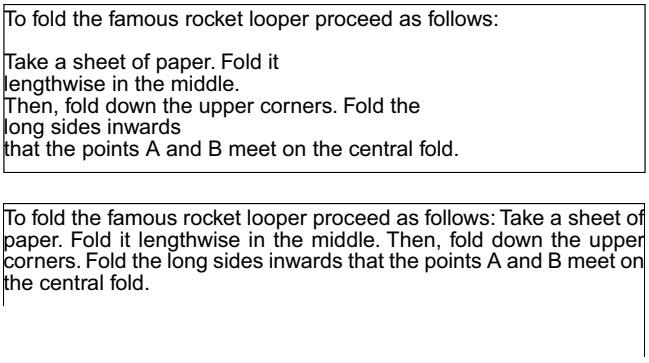


Fig. 9.21
Top: text with redundant line breaks

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

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

9.2.7 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)*: **

The glyph *U+00AD* will be used as hyphenation character if it is available in the font, otherwise U+002D. 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 9.22 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 9.23, the output looks okay even without explicit soft hyphens. The option list in both cases looks as follows:

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

Fig. 9.22
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. 9.23
Justified text without soft hyphens, 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.

9.2.8 Widow and Orphan Lines

If the first line (or lines) of a paragraph appears by itself at the bottom of a column or page it is called an orphan. Similarly, if the last line (or lines) of a paragraph appears at the beginning of the next column or page it is called a widow. Isolated orphan or widow lines are considered as undesirable in high-quality typesetting.

Orphan control. The Textflow option *minlinecount* specifies the minimum number of lines in the last paragraph of the fitbox. If there are fewer lines they are placed in the next fitbox. The value *minlinecount=2* can be used to prevent single orphan lines of a paragraph at the end of a fitbox.

Widow control. Since PDFlib doesn't know anything about future Textflow placements it cannot directly control widow lines. However, you can implement widow control in the client code according to the following scheme:

- Fit the first part of the Textflow into the first fitbox in blind mode (option *blind=true)*, i.e. without creating any real output.
- Fit the next part of the Textflow into the second fitbox in blind mode. Call *PDF_info_textflow( )* with the keyword *firstparalinecount* to query the number of lines in the first paragraph of the second fitbox. If the result is 1 you found a single-line widow.
- Now you can return to the first fitbox with the *rewind* option, and your algorithm must adjust the fitting options to avoid the widow line. For example, this can be achieved by reducing the number of lines in the first fitbox with the *maxlines* option.

Cookbook A full code sample can be found in the Cookbook topic `textflow/widows_and_orphans`.

9.2.9 Controlling the standard Linebreak Algorithm

PDFlib implements a sophisticated line-breaking algorithm. Table 9.2 lists Textflow options which control the line-breaking algorithm.

Line-breaking rules. When a word or other sequence of text surrounded by space characters doesn't fully fit into a line, it must be moved to the next line. In this situation the line-breaking algorithm decides after which characters a line break is possible.

For example, a formula such as *-12+235/8*45* will never be broken, while the string *PDF-345+LIBRARY* may be broken to the next line at the minus character. If the text contains soft hyphen characters it can also be broken after such a character.

For parentheses and quotation marks it depends on whether we have an opening or closing character: opening parentheses and quotations marks do not offer any break 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.

Table 9.2 Options for controlling the line-breaking algorithm

option	explanation
adjustmethod	*(Keyword) The method used to adjust a line when a text portion doesn't fit into a line after compressing or expanding the distance between words subject to the limits specified by the* `minspacing` *and* `maxspacing` *options. Default:* `auto` ***auto*** *The following methods are applied in order:* `shrink`, `spread`, `nofit`, `split`. ***clip*** *Same as* `nofit` *(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.*
advancedlinebreak	*(Boolean) Enable the advanced line breaking algorithm which is required for complex scripts. This is required for linebreaking in scripts which do not use space characters for designating word boundaries, e.g. Thai. The options* `locale` *and* `script` *will be honored. Default:* `false`
avoidbreak	*(Boolean) If* `true`*, avoid any line breaks until* `avoidbreak` *is reset to* `false`*. Default:* `false`
charclass	*(List of pairs, where the first element in each pair is a keyword, and the second element is either a unichar or a list of unichars) The specified unichars will be classified by the specified keyword to determine the line breaking behavior 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*
locale	*(Keyword) The locale which will be used for localized linebreaking methods if* `advancedlinebreak= true`*. The keywords consists of one or more components, where the optional components are separated by an underscore character '_' (the syntax slightly differs from NLS/POSIX locale IDs):* ► *A required two- or three-letter lowercase language code according to ISO 639-2 (see* `www.loc.gov/standards/iso639-2`*), e.g.* `en`*, (English),* `de` *(German),* `ja` *(Japanese). This differs from the* `language` *option.* ► *An optional four-letter script code according to ISO 15924 (see* `www.unicode.org/iso15924/iso15924-codes.html`*), e.g.* `Hira` *(Hiragana),* `Hebr` *(Hebrew),* `Arab` *(Arabic),* `Thai` *(Thai).* ► *An optional two-letter uppercase country code according to ISO 3166 (see* `www.iso.org/iso/country_codes/iso_3166_code_lists`*), e.g.* `DE` *(Germany),* `CH` *(Switzerland),* `GB` *(United Kingdom)* *The keyword* `_none` *specifies that no locale-specific processing will be done.* *Specifying a locale is required for advanced line breaking for some scripts, e.g. Thai. Default:* `_none` *Examples: Thai,* `de_DE`, `en_US`, `en_GB`

Table 9.2 Options for controlling the line-breaking algorithm

option	*explanation*
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*

Preventing linebreaks. You can use the *charclass* option to prevent Textflow from breaking a line after specific characters. For example, the following option will prevent line breaks immediately after the / character:

```
charclass={letter /}
```

In order to prevent a sequence of text from being broken across lines you can bracket it with *avoidbreak*...*noavoidbreak*.

Cookbook A full code sample can be found in the Cookbook topic `textflow/avoid_linebreaking`.

Formatting CJK text. The textflow engine is prepared to deal with CJK text, and properly treats CJK characters as ideographic glyphs as per the Unicode standard. As a result, CJK text will never be hyphenated. For improved formatting the following options are recommended when using Textflow with CJK text; they will disable hyphenation for inserted Latin text and create evenly spaced text output:

```
hyphenchar=none
alignment=justify
shrinklimit=100%
spreadlimit=100%
```

Vertical writing mode is not supported in Textflow.

Justified text in a narrow fitbox. The narrower the fitbox, the more important are the options for controlling justified text. Figure 9.24 demonstrates the results of the various methods for justifying text in a narrow fitbox. The option settings in Figure 9.24 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

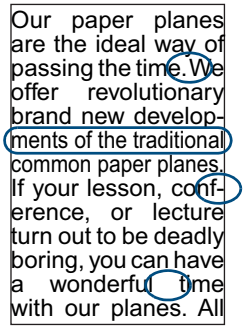


Fig. 9.24 Justified text in a narrow fitbox with default settings

the *shrink* method. Figure 9.25a shows how to avoid forced hyphenation by compressing text down to *shrinklimit=50%*.

Fig. 9.25 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%`
b)	Our paper planes are the ideal way of passing the time. We offer revolutionary b r a n d n e w 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`

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 9.25b 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 9.25c shows the generated text output with a minimum text width of 50%.

9.2.10 Advanced script-specific Line Breaking

PDFlib implements an additional line breaking algorithm on top of the standard line breaking algorithm. This advanced line breaking algorithm is required for some scripts, and improves line breaking behavior for some other script/locale combinations even if

it is not required. It can be enabled with the *advancedlinebreak* option. Since line breaking depends on the language of the text, the advanced line breaking algorithm honors the *script* option (see Table 7.2) and the *locale* option (see PDFlib API Reference). Advanced line breaking determines proper line break opportunities in the following situations:

- For scripts in which line breaking does not rely on the presence of space characters in the text, e.g. Thai. The following Textflow option list enables advanced line breaking for Thai:

  ```
  <advancedlinebreak script=thai locale=tha>
  ```

- In script/locale combinations which require specific treatment of certain punctuation characters, e.g. the « and » guillemet characters used as quotation marks in French text. The following Textflow option list enables advanced line breaking for French text. As a result, the guillemet characters surrounding a word will not be split apart from the word at the end of a line:

  ```
  <advancedlinebreak script=latn locale=fr>
  ```

Note that the *locale* Textflow option is different from the *language* text option (see Table 7.3): although the *locale* option can start with the same three-letter language identifier, it can optionally contain one or two additional parts. However, these will rarely be required for PDFlib.

9.2.11 Wrapping Text around Paths and Images

The wrapping feature can be used to fill arbitrary shapes with text or wrap text around a path. By means of matchboxes, explicit rectangles/polygons/circles/curves or path objects you can specify wrapping areas for the Textflow. If an image contains an integrated clipping path you can wrap text around the image clipping path automatically.

Wrapping text around an image with matchbox. In the first example we will place an image within the Textflow and run the text around the whole image. First the image is loaded and placed in the box at the desired position. To refer to the image by name later, define a matchbox called *img* when fitting the image, and specify a margin of 5 units with the option list *matchbox={name=img margin=-5}* as follows:

```
result = p.fit_image(image, 50, 35,
        "boxsize={80 46} fitmethod=meet position=center matchbox={name=img margin=-5}");
```

The Textflow is added. Then we place it using the *wrap* option with the image's matchbox *img* as the area to run around as follows (see Figure 9.26):

Fig. 9.26
Wrapping text around an image with matchbox

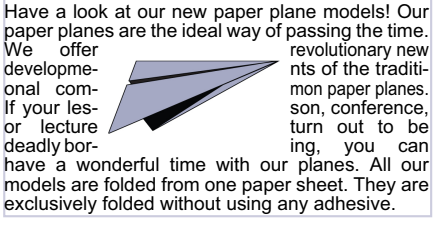
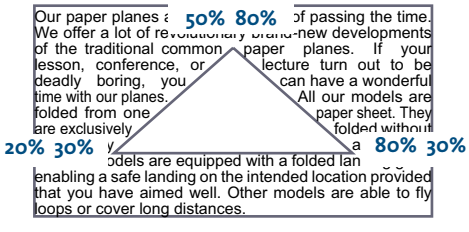


Fig. 9.27
Wrapping text around a triangular shape



```
result = p.fit_textflow(textflow, left_x, left_y, right_x, right_y,
        "wrap={usematchboxes={{img}}}");
```

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

Cookbook *A full code sample can be found in the Cookbook topic* `textflow/wrap_text_around_images`.

Wrapping text around arbitrary paths. You can create a path object (see Section 3.2.3, »Direct Paths and Path Objects«, page 66) and use it as a wrap shape. The following fragment constructs a path object with a simple shape (a circle) and supplies it to the *wrap* option of *PDF_fit_textflow( )*. The reference point for placing the path is expressed as percentage of the fitbox's width and height:

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

/* Visualize the path if desired */
p.draw_path(path, x, y, "stroke");

result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
        "wrap={paths={" +
            "{path=" + path + " refpoint={100% 50%} }" +
        "}}");

p.delete_path(path);
```

Use the *inversefill* option to wrap text inside the path instead of wrapping the text around the path (i.e. the path serves as text container instead of creating a hole in the Textflow):

```
result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
        "wrap={inversefill paths={" +
            "{path=" + path + " refpoint={100% 50%} }" +
        "}}");
```

Wrapping text around an image clipping path. TIFF and JPEG images can contain an integrated clipping path. The path must have been created in an image processing application and will be evaluated by PDFlib. If a default clipping path is found in the image it will be used, but you can specify any other clipping path in the image with the *clippingpathname* option of *PDF_load_image( )*. If the image has been loaded with a clipping path you can extract the path and supply it to the *wrap* option *PDF_fit_textflow( )* as above. We also supply the *scale* option to enlarge the imported image clipping path:

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

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

result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
        "wrap={paths={{path=" + path + " refpoint={50% 50%} scale=2}}}");

p.delete_path(path);
```

Placing an image and wrapping text around it. While the previous section used only the clipping path of an image (but not the image itself), let's now place the image inside the fitbox of the Textflow and wrap the text around it. In order to achieve this we must again load the image with the *clippingpathname* option and place it on the page with *PDF_fit_image()*. In order to create the proper path object for wrapping the Textflow we call *PDF_info_image()* with the same option list as *PDF_fit_image()*. This ensures that the same transformations (e.g. scaling) are applied when the clipping path is retrieved and when the image is actually placed. Finally, the reference point (the *x/y* parameters of *PDF_fit_image()*) must be supplied to the *refpoint* suboption of the *paths* suboption of the *wrap* option:

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

/* Place the image on the page with some fitting options */
String imageoptlist = "scale=2";
p.fit_image(image, x, y, imageoptlist);

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

result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
        "wrap={paths={{path=" + path + " refpoint={" + x + " " + y + "} }}}");

p.delete_path(path);
```

You can supply the same *wrap* option in multiple calls to *PDF_fit_textflow()*. This is useful if the placed image overlaps multiple Textflow fitboxes, e.g. for multi-column text.

Wrapping text around non-rectangular shapes. As an alternative to creating a path object as wrap shape you can specify path elements directly in Textflow options.

In addition to wrapping text around a rectangle specified by a matchbox you can define arbitrary graphical elements as wrapping shapes. For example, the following option list will wrap the text around a triangular shape (see Figure 9.27):

```
wrap={ polygons={ {50% 80%   20% 30%   80% 30%   50% 80%} } }
```

Note that the *showborder=true* option has been used to illustrate the margins of the shapes. The *wrap* option can contain multiple shapes. The following option list will wrap the text around two triangle shapes:

```
wrap={ polygons={ {50% 80%   20% 30%   80% 30%   50% 80%}
                  {20% 90%   10% 70%   30% 70%   20% 90%} } }
```

Instead of percentages (relative coordinates within the fitbox) absolute coordinates on the page can be used.

Note *It is recommended to set* `fixedleading=true` *when using shapes with segments which are neither horizontally nor vertically oriented.*

Cookbook *A full code sample can be found in the Cookbook topic* `textflow/wrap_text_around_polygons`.

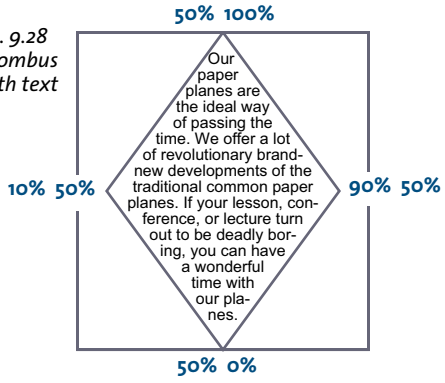


Fig. 9.28
Filling a rhombus shape with text

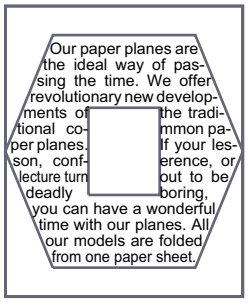


Fig. 9.29
Filling overlapping shapes

Filling non-rectangular shapes. The wrap feature can also be used to place the contents of a Textflow in arbitrarily shaped areas. This is achieved with the *addfitbox* or *inversefill* suboptions of the *wrap* option. Instead of wrapping the text around the specified shapes the text will be placed within one or more shapes. The following option list can be used to flow text into a rhombus shape, where the coordinates are provided as percentages of the fitbox rectangle (see Figure 9.28):

```
wrap={ addfitbox polygons={ {50% 100%   10% 50%   50% 0%   90% 50%   50% 100%} } }
```

Note that the *showborder=true* option has been again used to illustrate the margins of the shape. Without the *addfitbox* option the rhombus shape will remain empty and the text will be wrapped around it.

Filling overlapping shapes. In the next example we will fill a shape comprising two overlapping polygons, namely a hexagon with a rectangle inside. Using the *addfitbox* option the fitbox itself will be excluded from being filled, and the polygons in the subsequent list will be filled except in the overlapping area (see Figure 9.29):

```
wrap={ addfitbox polygons=
       { {20% 10%   80% 10%   100% 50%   80% 90%   20% 90%   0% 50%   20% 10%}
         {35% 35%   65% 35%   65% 65%   35% 65%   35% 35%} } }
```

Without the *addfitbox* option you will get the opposite effect: the previously filled area will remain empty, and the previously empty areas will be filled with text.

Cookbook *A full code sample can be found in the Cookbook topic* `textflow/fill_polygons_with_text`.

9.3 Table Formatting

The table formatting feature can be used to automatically format complex tables. Table cells may contain single- or multi-line text, images, SVG graphics or PDF pages. Tables are not restricted to a single fitbox, but can span multiple pages.

Cookbook *Code samples regarding table issues can be found in the* `table` *category of the PDFlib Cookbook.*

General aspects of a table. The description of the table formatter is based on the following concepts and terms (see Figure 9.30):

- A *table* is a virtual object with a rectangular outline. It is comprised of horizontal *rows* and vertical *columns*.
- A *simple cell* is a rectangular area within a table, defined as the intersection of a row and a column. A *spanning cell* spans more than one column, more than one row, or both. The term *cell* will be used to designate both simple and spanning cells.
- The complete table may fit into one fitbox, or several fitboxes may be required. The rows of the table which are placed in one fitbox constitute a *table instance*. Each call to *PDF_fit_table( )* will place one *table instance* in one fitbox (see Section 9.3.5, »Table Instances«, page 261).
- 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*.
- An optional caption (not shown in Figure 9.30) is an auxiliary element which can be used to place a description of the table. It can be placed on any side of the table.

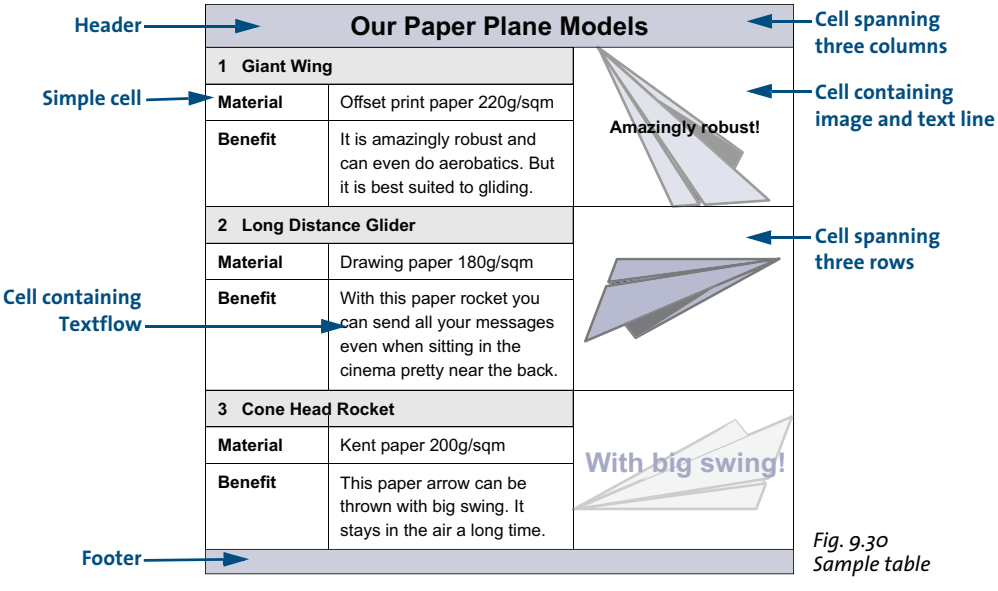


Fig. 9.30
Sample table

As an example, all aspects of creating the table in Figure 9.30 will be explained. A complete description of the table formatting options can be found in the *PDFlib API Reference*. Creating a table starts by defining the contents and visual properties of each

table cell with *PDF_add_table_cell( )*. Then you place the table using one or more calls to *PDF_fit_table( )*.

When placing the table the size of its fitbox and the ruling and shading of table rows or columns can be specified. Use the Matchbox feature for details such as cell-specific shading (see Section 9.4, »Matchboxes«, page 267, for more information).

In this section the most important options for defining the table cells and fitting the table will be discussed. All examples demonstrate the relevant calls of *PDF_add_table_cell( )* and *PDF_fit_table( )* only, assuming that the required font has already been loaded.

Note *Table processing is independent from the current graphics state. Table cells can be defined in* `document` *scope while the actual table placement must be done in* `page` *or* `pattern/template/glyph` *scope.*

Cookbook *A full code sample can be found in the Cookbook topic* `table/starter_table`.

Analyzing table-related formatting problems. Depending on the number of cells and the table formatting options the results of PDFlib's table formatter may sometimes not match your expectations. In almost all cases this can be rectified with suitable options. However, it may be difficult to identify a problematic cells or group of cells with the wrong options. In order to facilitate debugging of table-related formatting problems PDFlib offers the following options in *PDF_fit_table( )*:

- The option *showcells* visualizes the border of each inner cell box. If the function is called in page scope and PDF/A mode is not active, an annotation with details about the cell contents is placed in the center of each table cell.
- If the option *debugshow* is *true*, all errors for tables which are too high, too wide, or where the cells get too small are suppressed and are logged instead. The resulting table instance is created as a debugging aid although the table is damaged.
- If the option *showgrid* is *true*, the vertical and horizontal boundary of all columns and rows are stroked, i.e. the underlying tabular grid is visualized.

9.3.1 Placing a Simple Table

Before we describe the table concepts in more detail, we will demonstrate a simple example for creating a table. The table contains six cells which are arranged in three rows and two columns. Four cells contain text lines, and one cell contains a multi-line Textflow. All cell contents are horizontally aligned to the left, and vertically aligned to the center with a margin of 4 points.

To create the table we first prepare the option list for the text line cells by defining the required options *font* and *fontsize* and a position of *{left center}* in the *fittextline* suboption list. In addition, we define cell margins of 4 points. Then we add the text line cells one after the other in their respective column and row, with the actual text supplied directly in the call to *PDF_add_table_cell( )*.

In the next step we create a Textflow, use the Textflow handle to assemble the option list for the Textflow table cell, and add that cell to the table.

Finally we place the table with *PDF_fit_table( )* while visualizing the table frame and cell borders with black lines. Since we didn't supply any column widths, they will be calculated automatically from the supplied text lines plus the margins.

Cookbook *A full code sample can be found in the Cookbook topic* `table/vertical_text_alignment`.

The following code fragment shows how to create the simple table. The result is shown in Figure 9.31a.

```
/* Text for filling a table cell with multi-line Textflow */
String tf_text = "It is amazingly robust and can even do aerobatics. " +
                 "But it is best suited to gliding.";

/* Define the column widths of the first and the second column */
int c1 = 80, c2 = 120;

/* Define the lower left and upper right corners of the table instance (fitbox) */
double llx=100, lly=500, urx=300, ury=600;

/* bail out on errors */
p.set_option("errorpolicy=exception");

/* Load the font */
font = p.load_font("Helvetica", "unicode", "");

/* Define the option list for the text line cells placed in the first column */
optlist = "fittextline={position={left center} font=" + font + " fontsize=8} margin=4" +
          colwidth=" + c1;

/* Add a text line cell in column 1 row 1 */
tbl = p.add_table_cell(tbl, 1, 1, "Our Paper Planes", optlist);

/* Add a text line cell in column 1 row 2 */
tbl = p.add_table_cell(tbl, 1, 2, "Material", optlist);

/* Add a text line cell in column 1 row 3 */
tbl = p.add_table_cell(tbl, 1, 3, "Benefit", optlist);

/* Define the option list for a text line placed in the second column */
optlist = "fittextline={position={left center} font=" + font + " fontsize=8} " +
          "colwidth=" + c2 + " margin=4";

/* Add a text line cell in column 2 row 2 */
tbl = p.add_table_cell(tbl, 2, 2, "Offset print paper 220g/sqm", optlist);

/* Add a Textflow */
optlist = "font=" + font + " fontsize=8 leading=110%";
tf = p.add_textflow(-1, tf_text, optlist);

/* Define the option list for the Textflow cell using the handle retrieved above */
optlist = "textflow=" + tf + " margin=4 colwidth=" + c2;

/* Add the Textflow table cell in column 2 row 3 */
tbl = p.add_table_cell(tbl, 2, 3, "", optlist);

p.begin_page_ext(0, 0, "width=200 height=100");

/* Define the option list for fitting the table with table frame and cell ruling */
optlist = "stroke={{line=frame linewidth=0.8} {line=other linewidth=0.3}}";

/* Place the table instance */
result = p.fit_table(tbl, llx, lly, urx, ury, optlist);

/* Check the result; "_stop" means all is ok. */
```

```
if (!result.equals("_stop")) {
        if (result.equals( "_error"))
            throw new Exception("Error: " + p.get_errmsg());
        else {
            /* Any other return value requires dedicated code to deal with */
        }
}
p.end_page_ext("");

/* This also deletes Textflow handles used in the table */
p.delete_table(tbl, "");
```

Fine-tuning the vertical alignment of cell contents. When we vertically center contents of various types in the table cells, they will be positioned with varying distance from the borders. In Figure 9.31a, the four text line cells have been placed with the following option list:

```
optlist = "fittextline={position={left center} font=" + font +
          " fontsize=8} colwidth=80 margin=4";
```

The Textflow cell is added without any special options. Since we vertically centered the text lines, the *Benefit* line will move down with the height of the Textflow.

Fig. 9.31 Aligning text lines and Textflow in table cells

Generated output

a)

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

b)

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

As shown in Figure 9.31b, we want all cell contents to have the same *vertical* distance from the cell borders regardless of whether they are Textflows or text lines. To accomplish this we first prepare the option list for the text lines. We define a fixed row height of 14 points, and the position of the text line to be on the top left with a margin of 4 points.

The *fontsize=8* option which we supplied before doesn't exactly represent the letter height but adds some space below and above. However, the height of an uppercase letter is exactly represented by the *capheight* value of the font. For this reason we use *fontsize={capheight=6}* which will approximately result in a font size of 8 points and (along with *margin=4)*, will sum up to an overall height of 14 points corresponding to the *rowheight* option. The complete option list of *PDF_add_table_cell( )* for our text line cells looks as follows:

```
/* option list for the text line cells */
optlist = "fittextline={position={left top} font=" + font +
          " fontsize={capheight=6}} rowheight=14 colwidth=80 margin=4";
```

To add the Textflow we use *fontsize={capheight=6}* which will approximately result in a font size of 8 points and (along with *margin=4*), will sum up to an overall height of 14 points as for the text lines above.

```
/* option list for adding the Textflow */
optlist = "font=" + font + " fontsize={capheight=6} leading=110%";
```

In addition, we want the baseline of the *Benefit* text aligned with the first line of the Textflow. At the same time, the *Benefit* text should have the same distance from the top cell border as the *Material* text. To avoid any space from the top we add the Textflow cell using *fittextflow={firstlinedist=capheight}*. Then we add a margin of 4 points, the same as for the text lines:

```
/* option list for adding the Textflow cell */
optlist = "textflow=" + tf + " fittextflow={firstlinedist=capheight} "
          "colwidth=120 margin=4";
```

Cookbook *A full code sample can be found in the Cookbook topic* `table/vertical_text_alignment`.

9.3.2 Contents of a Table Cell

When adding cells to a table with *PDF_add_table_cell( )*, you can specify various kinds of cell contents. Table cells can contain one or more content types at the same time. Additional ruling and shading is available, as well as matchboxes which can be used for placing additional content in a table cell.

For example, the cells of the paper plane table contain the elements illustrated in Figure 9.32.

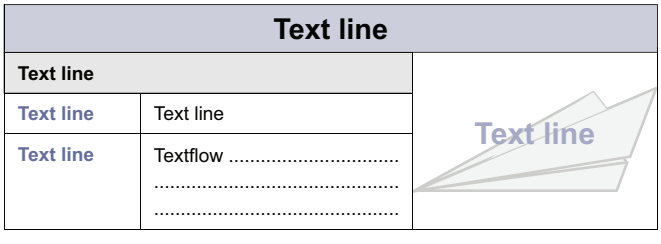


Fig. 9.32 Contents of the table cells

Single-line text with Textlines. The text is supplied in the *text* parameter of *PDF_add_table_cell( )*. In the *fittextline* option list all formatting options of *PDF_fit_textline( )* can be specified. The default fit method is *fitmethod=nofit*. The cell will be enlarged if the text doesn't completely fit into the cell. To avoid this, use *fitmethod=auto* to shrink the text, subject to the *shrinklimit* option. If no row height is specified the formatter assumes twice the text size as height of the table cell (more precisely: twice the *boxheight*, which has the default value *{capheight none}* unless specified otherwise). The same applies to the row width for rotated text.

Use the *fillcolor* option in the *fittextline* option list to apply color to the text.

Multi-line text with Textflow. The Textflow must have been prepared outside the table functions and created with *PDF_create_textflow( )* or *PDF_add_textflow( )* before calling *PDF_add_table_cell( )*. The Textflow handle is supplied in the *textflow* option. In the *fittextflow* option all formatting options of *PDF_fit_textflow( )* can be specified.

The default fit method is *fitmethod=clip*. This means: First it is attempted to completely fit the text into the cell. If the cell is not large enough its height will be increased. If the text do not fit anyway it will be clipped at the bottom. To avoid this, use *fitmethod=auto* to shrink the text subject to the *minfontsize* option.

When the cell is too narrow the Textflow could be forced to split single words at undesired positions. If the *checkwordsplitting* option is *true* the cell width will be enlarged until no word splitting occurs any more.

Images and templates. Images must be loaded with *PDF_load_image( )* before calling *PDF_add_table_cell( )*. Templates must be created with *PDF_begin_template_ext( )*. The image or template handle is supplied in the *image* option. In the *fitimage* option all formatting options of *PDF_fit_image( )* can be specified. The default fit method is *fitmethod=meet*. This means that the image/template will be placed completely inside the cell without distorting its aspect ratio. The cell size will not be changed due to the size of the image/template.

Vector graphics. Graphics must be loaded with *PDF_load_graphics( )* before calling *PDF_add_table_cell( )*. The graphics handle is supplied in the *graphics* option. In the *fitgraphics* option all formatting options of *PDF_fit_graphics( )* can be specified. The default fit method is *fitmethod=meet*. This means that the graphics 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 graphics.

Pages from an imported PDF document. The PDI page must have been opened with *PDF_open_pdi_page( )* before calling *PDF_add_table_cell( )*. The PDI page handle is supplied in the *pdipage* option. In the *fitpdipage* option all formatting options of *PDF_fit_pdi_page( )* can be specified. The default fit method is *fitmethod=meet*. This means that the PDI page will be placed completely inside the cell without distorting its aspect ratio. The cell size will not be changed due to the size of the PDI page.

Path objects. Path objects must have been created with *PDF_add_path_point( ) before calling PDF_add_table_cell( )*. The path handle is supplied in the *path* option. In the *fitpath* option all formatting options of *PDF_draw_path( )* can be specified. The bounding box of the path will be placed in the table cell. The lower left corner of the inner cell box will be used as reference point for placing the path.

Annotations. Annotations in table cells can be created with the *annotationtype* option of *PDF_add_table_cell( )* which corresponds to the *type* parameter of *PDF_create_annotation( )* (but this function does not have to be called). In the *fitannotation* option all options of *PDF_create_annotation( )* can be specified. The cell box will be used as annotation rectangle.

Form fields. Form fields in table cells can be created with the *fieldname* and *fieldtype* options of *PDF_add_table_cell( )* which correspond to the *name* and *type* parameters of

PDF_create_field( ) (but this function does not have to be called). In the *fitfield* option all options of *PDF_create_field( )* can be specified. The cell box will be used as field rectangle.

Positioning cell contents in the inner cell box. By default, cell contents are positioned with respect to the cell box. The *margin* options of *PDF_add_table_cell( )* can be used to specify some distance from the cell borders. The resulting rectangle is called the *inner cell box*. If any margin is defined the cell contents will be placed with respect to the inner cell box (see Figure 9.33). If no margins are defined, the inner cell box is identical to the cell box.
In addition, cell contents may be subject to further options supplied in the content-specific fit options, as described in section Section 9.3.4, »Mixed Table Contents«, page 258.

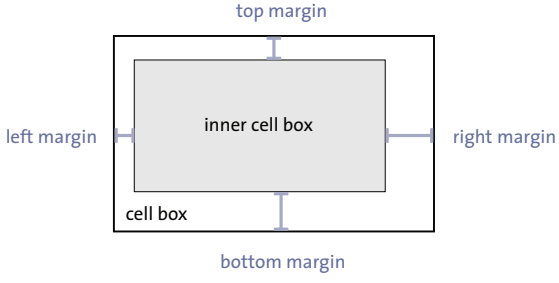


Fig. 9.33
Fitting contents in the inner cell box

9.3.3 Table and Column Widths

When adding a cell to the table, you define the number of columns and/or rows spanned by the cell with the *colspan* and *rowspan* options. By default, a cell spans one column and one row. The total number of columns and rows in the table is implicitly increased by the respective values when adding a cell. Figure 9.34 shows an example of a table containing three columns and four rows.

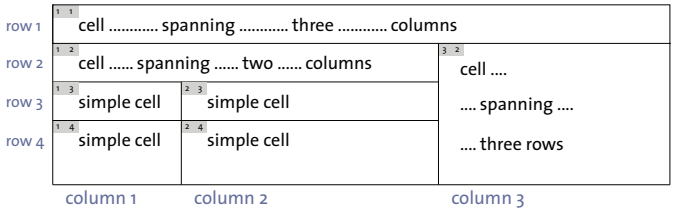


Fig. 9.34
Simple cells and cells spanning several rows or columns

Furthermore you can explicitly supply the width of the first column spanned by the cell with the *colwidth* option. Keep in mind that the supplied *colwidth* affects only the first column, not all columns in the *colspan* group. By supplying each cell with a defined first column width all width values implicitly add up to the total table width. Figure 9.35 shows an example.

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.

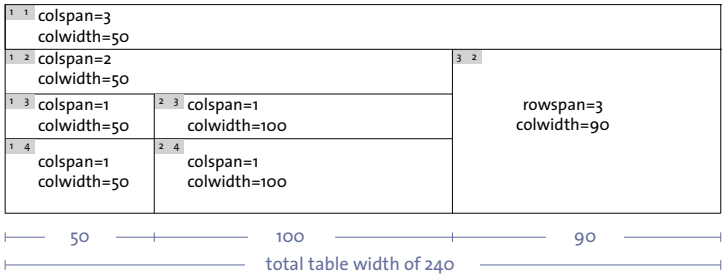


Fig. 9.35
Column widths define the total table width.

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 9.36),

column scaling group

	Max. Load	Range	Weight	Speed
Giant Wing	12g	18m	14g	8m/s
Long Distance Glider	5g	30m	11.2g	5m/s
Cone Head Rocket	7g	7m	12.4g	6m/s

Fig. 9.36
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.

9.3.4 Mixed Table Contents

In the following sections we will create the sample table containing various kinds of contents as shown in Figure 9.37 step by step.

Cookbook *A full code sample can be found in the Cookbook topic* `table/mixed_table_contents`.

As a prerequisite we need to load two fonts. We define the dimensions of the table's fitbox in terms of the coordinates of the lower left and upper right corners and specify the widths of the three table columns. Then, we start a new page with A4 size:

```
double llx = 100, lly = 500, urx = 360, ury = 600; // coordinates of the table

int c1 = 50, c2 = 120, c3 = 90; // widths of the three table columns

boldfont = p.load_font("Helvetica-Bold", "unicode", "");
normalfont = p.load_font("Helvetica", "unicode", "");

p.begin_page_ext(0, 0, "width=a4.width height=a4.height");
```

Step 1: Adding the first cell. We start with the first cell of our table. The cell will be placed in the first column of the first row and will span three columns. The first column has a width of 50 points. The text line is centered vertically and horizontally, with a margin of 4 points from all borders. The following code fragment shows how to add the first cell:

```
optlist = "fittextline={font=" + boldfont + " fontsize=12 position=center} " +
          "fillcolor=red margin=4 colspan=3 colwidth=" + c1;

tbl = p.add_table_cell(tbl, 1, 1, "Our Paper Plane Models", optlist);
```

Step 2: Adding one cell spanning two columns. In the next step we add the cell containing the text line *1 Giant Wing*. It will be placed in the first column of the second row and spans two columns. The first column has a width of 50 points. The row height is 14 points. The text line is positioned on the top left, with a margin of 4 points from all borders. We use *fontsize={capheight=6}* to get a unique vertical text alignment as described in »Fine-tuning the vertical alignment of cell contents«, page 254.

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 9.4, »Matchboxes«, page 267. The following code fragment demonstrates how to add the *Giant Wing* heading cell:

```
optlist = "fittextline={position={left top} font=" + boldfont +
          " fontsize={capheight=6}} rowheight=14 colwidth=" + c1 +
          " margin=4 colspan=2 matchbox={fillcolor={gray .92}}";

tbl = p.add_table_cell(tbl, 1, 2, "1  Giant Wing", optlist);
```

Fig. 9.37 Adding table cells with various contents step by step

Generated table

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

Generation steps

Step 1: Add a cell spanning 3 columns
Step 2: Add a cell spanning 2 columns
Step 3: Add 3 more text line cells
Step 4: Add the Textflow cell
Step 5: Add the image cell with a text line
Step 6: Fitting the table

Step 3: Add three more Textline cells. The following code fragment adds the *Material*, *Benefit* and *Offset print paper...* cells. The *Offset print paper...* cell will start in the second column defining a column width of 120 points. The cell contents is positioned on the top left, with a margin of 4 points from all borders.

```
optlist = "fittextline={position={left top} font=" + normalfont +
          " fontsize={capheight=6}} rowheight=14 colwidth=" + c1 + " margin=4";

tbl = p.add_table_cell(tbl, 1, 3, "Material", optlist);
tbl = p.add_table_cell(tbl, 1, 4, "Benefit", optlist);

optlist = "fittextline={position={left top} font=" + normalfont +
```

```
          " fontsize={capheight=6}} rowheight=14 colwidth=" + c2 + " margin=4";

tbl = p.add_table_cell(tbl, 2, 3, "Offset print paper 220g/sqm", optlist);
```

Step 4: Add the Textflow cell. The following code fragment adds the *It is amazingly...* Textflow cell. To add a table cell containing a Textflow we first create the Textflow. We use *fontsize={capheight=6}* which will approximately result in a font size of 8 points and (along with *margin=4*), will sum up to an overall height of 14 points as for the text lines above.

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

optlist = "font=" + normalfont + " fontsize={capheight=6} leading=110%";

tf = p.add_textflow(-1, tftext, optlist);
```

The retrieved Textflow handle will be used when adding the table cell. The first line of the Textflow should be aligned with the baseline of the *Benefit* text line. At the same time, the *Benefit* text should have the same distance from the top cell border as the *Material* text. Add the Textflow cell using *fittextflow={firstlinedist=capheight}* to avoid any space from the top. Then add a margin of 4 points, the same as for the text lines:

```
optlist = "textflow=" + tf + " fittextflow={firstlinedist=capheight} " +
          "colwidth=" + c2 + " margin=4";

tbl = p.add_table_cell(tbl, 2, 4, "", optlist);
```

Step 5: Add the image cell with a text line. In the fifth step we add a cell containing an image of the Giant Wing paper plane as well as the *Amazingly robust!* text line. The cell will start in the third column of the second row and spans three rows. The column width is 90 points. The cell margins are set to 4 points. For a first variant we place a TIFF image in the cell:

```
image = p.load_image("auto", "kraxi_logo.tif", "");

optlist = "fittextline={font=" + boldfont + " fontsize=9} image=" + image +
          " colwidth=" + c3 + " rowspan=3 margin=4";

tbl = p.add_table_cell(tbl, 3, 2, "Amazingly robust!", optlist);
```

Alternatively, you could import the image as a PDF page. Make sure that the PDI page is closed only after the call to *PDF_fit_table()*:

```
int doc = p.open_pdi("kraxi_logo.pdf", "", 0);

page = p.open_pdi_page(doc, pageno, "");

optlist = "fittextline={font=" + boldfont + " fontsize=9} pdipage=" + page +
          " colwidth=" + c3 + " rowspan=3 margin=4";

tbl = p.add_table_cell(tbl, 3, 2, "Amazingly robust!", optlist);
```

Step 6: Fit the table. In the last step we place the table with *PDF_fit_table()*. Using *header=1* the table header will include the first line. The *fill* option and the suboptions

area=header and *fillcolor={rgb 0.8 0.8 0.87}* specify the header row(s) to be filled with the supplied color. Using the *stroke* option and the suboptions *line=frame linewidth=0.8* we define a ruling of the table frame with a line width of 0.8. Using *line=other linewidth=0.3* a ruling of all cells is specified with a line width of 0.3:

```
optlist = "header=1 fill={{area=header fillcolor={rgb 0.8 0.8 0.87}}} " +
          "stroke={{line=frame linewidth=0.8} {line=other linewidth=0.3}}";

result = p.fit_table(tbl, llx, lly, urx, ury, optlist);

if (result.equals("_error"))
        throw new Exception("Error: " + p.get_errmsg());

p.end_page_ext("");
```

9.3.5 Table Instances

The rows of the table which are placed in one fitbox comprise a table instance. One or more table instances may be required to represent the full table. Each call to *PDF_fit_table()* will place one table instance in one fitbox. The fitboxes can be placed on the same page, e.g. with a multi-column layout, or on several pages.

The table in Figure 9.38 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.

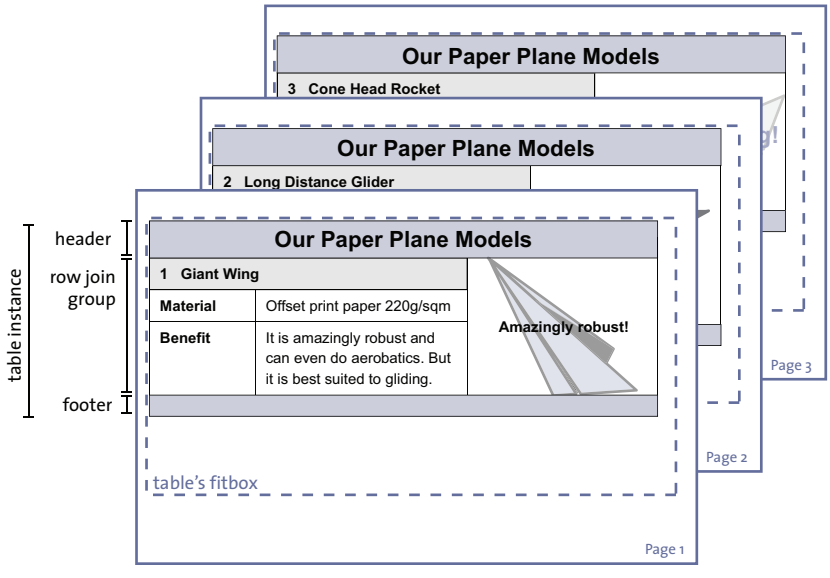


Fig. 9.38
Table broken into several table instances placed in one fitbox each.

The following code fragment shows the general loop for fitting table instances until the table has been placed completely. New pages are created as long as more table instances need to be placed.

```
do {
        /* Create a new page */
        p.begin_page_ext(0, 0, "width=a4.width height=a4.height");

        /* Use the first row as header and draw lines for all table cells */
```

```
        optlist = "header=1 stroke={{line=other}}";

        /* Place the table instance */
        result = p.fit_table(tbl, llx, lly, urx, ury, optlist);
        if (result.equals("_error"))
            throw new Exception("Error: " + p.get_errmsg());

        p.end_page_ext("");

} while (result.equals("_boxfull"));

/* Check the result; "_stop" means all is ok. */
if (!result.equals("_stop")) {
        if (result.equals( "_error"))
            throw new Exception("Error: " + p.get_errmsg());
        else {
            /* Any other return value is a user exit caused by the "return" option;
             * this requires dedicated code to deal with. */
            throw new Exception ("User return found in Textflow");
        }
}
/* This will also delete Textflow handles used in the table */
p.delete_table(tbl, "");
```

Headers and footers. With the *header* and *footer* options of *PDF_fit_table()* you can define the number of initial or final table rows which will be placed at the top or bottom of a table instance. Using the *fill* option with *area=header* or *area=footer*, headers and footers can be individually filled with color. Header rows consist of the first *n* rows of the table definition and footer rows of the last *m* rows.

Headers and footers are specified per table instance in *PDF_fit_table()*. Consequently, they can differ among table instances: while some table instances include headers/footers, others can omit them, e.g. to specify a special row in the last table instance.

Joining rows. In order to ensure that a set of rows will be kept together in the same table instance, they can be assigned to the same row join group using the *rowjoingroup* option. The row join group contains multiple consecutive rows. All rows in the group will be prevented from being separated into multiple table instances.

The rows of a cell spanning these rows don't constitute a join group automatically.

Splitting a cell. If the last rows spanned by a cell don't fit in the fitbox the cell will be split. In case of an image, PDI page, SVG graphics or text line cell, the cell contents will be repeated in the next table instance. In case of a Textflow cell the cell contents will continue in the remaining rows of the cell.

Figure 9.39 shows how the Textflow cell will be split while the Textflow continues in the next row. In Figure 9.40, an image cell is shown which will be repeated in the first row of the next table instance.

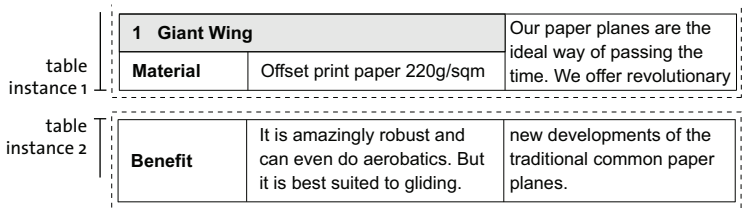


Fig. 9.39
Splitting a cell

Splitting a row. If the last body row doesn't completely fit into the table's fitbox, it will usually not be split. This behavior is controlled by the *minrowheight* option of *PDF_fit_table( )* with a default value of 100%. With this default setting the row will not be split but will completely be placed in the next table instance.

You can decrease the *minrowheight* value to split the last body row with the given percentage of contents in the first instance, and place the remaining parts of the row in the next instance.

Figure 9.40 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 is split and the image is repeated. The *Benefit* cell is repeated as well.

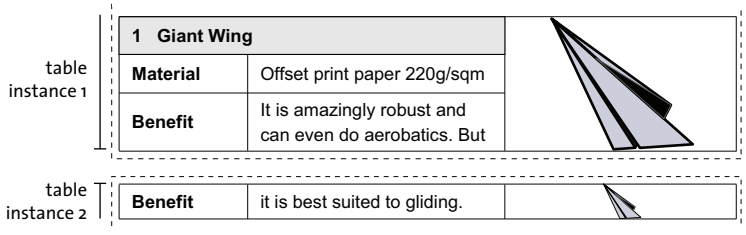


Fig. 9.40
Splitting a row

9.3.6 Table Formatting Algorithm

This section details the steps performed by the table formatter when placing a table. The description below applies to horizontal text. However, if you swap the terms »row height« and »column width« it also applies to vertical or rotated text.

In the first call to *PDF_fit_table( )* the options *colwidth*, *rowheight*, *fittextline*, and *fittextflow* are examined for all cells, and the width and height of the full table is calculated based on column widths, row heights, Textline and Textflow contents and the width of the first fitbox. The height of the fitbox is assumed as infinite. The first table instance (i.e. the placement of the first part of the table in the first fitbox) is calculated according to the *fitmethod* option of *PDF_fit_table( )*.

Calculate the height and width of table cells with Textlines. The table formatter initially determines the size of all those table cells with Textlines which span table columns or rows without *colwidth* or *rowheight*. In order to achieve this it calculates the width of the Textline and therefore the table cell according to the *fittextline* option. It assumes twice the text size as height of the table cell (more precisely: twice the box height,

which has the default value *{capheight none}* unless specified otherwise). For vertical text the width of the widest character will be used as cell width. For text orientated to west or east twice the text height will be used as cell width.

The remaining width and height of the table cell is then distributed among all columns and rows spanned by the cell for which *colwidth* or *rowheight* hasn't been specified.

Calculate a tentative table size. In the next step the formatter calculates a tentative table width and height as the sum of all column widths and row heights, respectively. Column widths and row heights specified as percentages are converted to absolute values based on the width and height of the first fitbox. If there are still columns or rows without *colwidth* or *rowheight* the remaining space is evenly distributed until the tentative table size equals the first fitbox.

The *rowheightdefault* option can be used to completely fill the height of the fitbox (keywords *auto* and *distribute)* or to save space (keyword *minimum).* Explicitly specifying the height of a row with the *rowheight* option always overrides the *rowheightdefault* setting.

Enlarge cells which are too small. Now the formatter determines all inner cell boxes (see Figure 9.33). If the combined margins are larger than the cell's width or height, the cell box is suitably enlarged by evenly enlarging all columns and rows which belong to the cell.

Fit Textlines horizontally. The formatter attempts to increase the width of all cells with Textlines so that the Textline fits into the cell without reducing the font size. If this is not possible, the Textline is automatically placed with *fitmethod=auto*. This guarantees that the Textline will not extend beyond the inner cell box. You can prevent the cell width from being increased by setting *fitmethod=auto* in the *fittextline* option.

You can use the *colscalegroup* option to make sure that all columns which belong to the same column scaling group will be scaled to equal widths, i.e. there widths will be unified and adjusted to the widest column in the group (see Figure 9.36).

Avoid forced hyphenation. If the calculated table width is smaller than the fitbox the formatter tries to increase the width of a Textflow cell so that the text fits without forced hyphenation. This can be avoided with the option *checkwordsplitting=false*. The widths of such cells will be increased until the table width equals the width of the fitbox.

You can query the difference between table width and fitbox width with the *horboxgap* key of *PDF_info_table( )*.

Fit text vertically. The formatter attempts to increase the height of all Textline and Textflow cells so that the Textline or Textflow fits into the inner cell box without reducing the font size. However, the cell height will not be increased if for a Textline or Textflow the suboption *fitmethod=auto* is set, or a Textflow is continued in another cell with the *continuetextflow* option.

This process of increasing the cell height applies only to cells containing a Textline or Textflow, but not for other types of cell contents, i.e. images, graphics, PDI pages, path objects, annotations, and fields.

You can use the *rowscalegroup* option to make sure that all rows which belong to the same row scaling group will be scaled to equal heights.

Continue the table in the next fitbox. If the table's resulting total height is larger than the fitbox (i.e. not all table cells fit into the fitbox), the formatter stops placing rows in the fitbox before it encounters the first row which doesn't fit into the fitbox.

If a cell spans multiple lines and not all of those lines fit into the fitbox, this cell will be split. If the cell contains an image, PDI page, SVG graphics, path object, annotation, form field, or Textline, the cell contents will be repeated in the next fitbox unless *repeatcontent=false* has been specified. Textflows, however, will be continued in the subsequent rows spanned by the cell (see Figure 9.39).

You can use the *rowjoingroup* option to make sure that all rows belonging to a row joining group will always appear together in a fitbox. All rows which belong to the header or footer plus one body line automatically form a row joining group. The formatter may therefore stop placing table rows before it encounters the first line which doesn't fit into the fitbox (see Figure 9.38).

You can use the *return* option to make sure that now more rows will be placed in the table instance after placing the affected line.

Split a row. A row may be split if it is very high or if there is only a single body line. If the last body line doesn't fully fit into the table's fitbox, it will completely be moved to the next fitbox. This behavior is controlled by the *minrowheight* option of *PDF_fit_table( )*, which has a default value of 100%. If you reduce the *minrowheight* value the specified percentage of the content of the last body line will be placed in the current fitbox and the rest of the line in the next fitbox (see Figure 9.40).

You can check whether a row has been split with the *rowsplit* keyword of *PDF_info_table( )*.

Shrink the table to the width of the fitbox. The table width as sum of the supplied column widths may be larger than the fitbox width after one of the previous steps, e.g. after fitting a Textline horizontally. In this case all columns are evenly reduced until the table width equals the width of the fitbox. This shrinking process is limited by the *horshrinklimit* option. In order to avoid any horizontal shrinking use *horshrinklimit=100%*.

You can query the horizontal shrinking factor with the *horshrinking* keyword of *PDF_info_table( )*.

If the *horshrinklimit* threshold is exceeded, i.e. the table would have to be squeezed too much horizontally, the following error message appears:

```
Table width $1 exceeds fitbox width $2 (required shrinking factor $3 is smaller than
'horshrinklimit'
```

Here *$1* designates the calculated table width, *$2* the width of the fitbox as supplied to *PDF_fit_table( )*, and *$3* the calculated shrinking percentage. If you run into this error you must supply a larger value for the width of the fitbox.

Shrink the table to the height of the fitbox. The table height for the header and footer rows plus at least one body row or row join group may be larger than the fitbox height. In this case all rows are evenly reduced until the table height equals the height of the fit-

box. This shrinking process is limited by the *vertshrinklimit* option. In order to avoid any vertical shrinking use *vertshrinklimit= 100%*.

You can query the vertical shrinking factor with the *vertshrinking* keyword of *PDF_info_table( )*.

If the *vertshrinklimit* threshold is exceeded, i.e. the table would have to be squeezed too much vertically, the following error message appears:

```
Table height $1 exceeds fitbox height $2 (required shrinking factor $3 is smaller than
'vertshrinklimit'
```

Here *$1* designates the calculated table height, *$2* the height of the fitbox as supplied to *PDF_fit_table( )*, and *$3* the calculated shrinking percentage. If you run into this error you must supply a larger value for the height of the fitbox.

Subsequent fitboxes with different widths. If subsequent fitboxes are wider than the first fitbox, the table cells are not expanded to completely fill the fitbox width. As a result all table instances whose fitboxes are wider than the first fitbox have the same width by default. To completely fill the fitbox the option *fitmethod=auto* must be supplied. With *fitmethod=meet* the cell heights are suitably scaled to preserve their aspect ratio.

If subsequent fitboxes are narrower than the first fitbox, the widths of the table cells are reduced to fit into the narrower fitbox. In order to calculate the table width anew, call *PDF_fit_table( )* with *rewind=1*.

9.4 Matchboxes

Matchboxes provide access to coordinates calculated by PDFlib as a result of placing some content on the page. Matchboxes are not defined with a dedicated API 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 area 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 calculates the matchbox as a rectangle corresponding to the bounding box which describes the position of the element on the page (as specified by all relevant options). For Textflows and table cells a matchbox may consist of multiple rectangles because of line or row breaking.

The rectangle(s) of a matchbox will be drawn before drawing the element to be placed. As a result, the element may obscure the effect of the matchbox border or filling, but not vice versa. In particular, those parts of the matchbox which overlap the area covered by an image are hidden by the image. If the image is placed with *fitmethod=slice* or *fitmethod=clip* the matchbox borders outside the image fitbox will be clipped as well. To avoid this effect the matchbox rectangle can be drawn using the basic drawing functions, e.g. *PDF_rect( )*, after the *PDF_fit_image( )* call. The coordinates of the matchbox rectangle can be retrieved using *PDF_info_matchbox( )* as far as the matchbox has been provided with a name in the *PDF_fit_image( )* call.

In the following sections some examples for using matchboxes are shown. For details about the functions which support the *matchbox* option list, see the *PDFlib API Reference*.

Note *Matchboxes are not supported in* `blind` *mode, i.e. formatting with the* `blind` *option.*

9.4.1 Decorating a Textline

Let's start with a discussion of matchboxes in text lines. In *PDF_fit_textline( )* the matchbox is the text box of the supplied text. The width of the text box 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 9.41a).

```
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 9.41b).

To increase the box height to match the font size we can use *boxheight={fontsize descender}* (see Figure 9.41c).

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 9.41d).

Cookbook *A full code sample can be found in the Cookbook topic* `textflow/text_on_color`.

Fig. 9.41 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`

9.4.2 Using Matchboxes in a Textflow

Decorating parts of a Textflow. In this section we will decorate some text within a Textflow: The words *very dangerous* will be emphasized similar to a marker pen. To accomplish this the words are enclosed in the *matchbox* and *matchbox=end* inline options (see Figure 9.42).

Fig. 9.42 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=red 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_annotation( )* will be ignored).

Cookbook *A full code sample can be found in the Cookbook topic* `textflow/weblink_in_text`.

```
/* create and fit Textflow with matchbox "kraxi" */
String  tftext =
        "For more information about the Giant Wing Paper Plane see the Web site of " +
        "<underline=true matchbox={name=kraxi boxheight={fontsize descender}}>" +
        "Kraxi Systems, Inc.<matchbox=end underline=false>";

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

```
tflow = p.create_textflow(tftext, optlist);
if (tflow == -1)
        throw new Exception("Error: " + p.get_errmsg());

result = p.fit_textflow(tflow, 0, 0, 50, 70, "fitmethod=auto");
if (!result.equals("_stop"))
        { /* ... */ }

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

/* create Link annotation on matchbox "kraxi" */
optlist = "action={activate " + act + "} linewidth=0 usematchbox={kraxi}";
p.create_annotation(0, 0, 0, 0, "Link", optlist);
```

Even if the text *Kraxi Systems, Inc.* spans several lines the appropriate number of link annotations will be created automatically with a single call to *PDF_create_annotation()*. The result in shown in Figure 9.43.

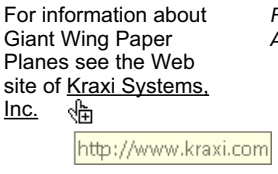


Fig. 9.43
Add Weblinks to parts of a Textflow

9.4.3 Matchboxes and Images

Adding a Web link to an image. To add a Web link to the area covered by an image the image matchbox can be used. The code is similar to »Adding a Web link to the Textflow matchbox«, page 268, above. However, instead of placing the Textflow, fit the image using the following option list:

```
String  optlist = "boxsize={130 130} fitmethod=meet matchbox={name=kraxi}";
p.fit_image(image, 10, 10, optlist);
```

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

Drawing a frame around an image. In this example we want to use the image matchbox to draw a frame around the image. We completely fit the image into the supplied box while maintaining its proportions using *fitmethod=meet*. We use the *matchbox* option with the *borderwidth* suboption to draw a thick border around the image. The *strokecolor* suboption determines the border color, and the *linecap* and *linejoin* suboptions are used to round the corners.

Cookbook *A full code sample can be found in the Cookbook topic* `images/frame_around_image`.

The matchbox is always drawn before the image which means it would be partially hidden by the image. To avoid this we use the *offset* suboptions with 50 percent of the border width to enlarge the frame beyond the area covered by the image. Alternatively, we could increase the border width accordingly. Figure 9.44 shows the option list used with *PDF_fit_image()* to draw the frame.

Fig. 9.44 Using the image matchbox to draw a frame around the image

Generated output	*Option list for PDF_fit_image( )*
	`boxsize={60 60} position={center} fitmethod=meet matchbox={name=kraxi borderwidth=4 offsetleft=-2 offsetright=2 offsetbottom=-2 offsettop=2 linecap=round linejoin=round strokecolor={rgb 0.0 0.3 0.3}}`

Align text at an image. The following code fragment shows how to align vertical text at the right margin of an image. The image is proportionally fit into the supplied box with a fit method of *meet*. The actual coordinates of the fitbox are retrieved with *PDF_info_matchbox( )* and a vertical text line is placed relative to the lower right *(x2, y2)* corner of the fitbox. The border of the matchbox is stroked (see Figure 9.45).

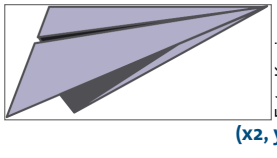
Cookbook A full code sample can be found in the Cookbook topic `images/align_text_at_image`.

```
/* use this option list to load and fit the image */
String  optlist = "boxsize={300 200} position={center} fitmethod=meet " +
        "matchbox={name=giantwing borderwidth=3 strokecolor={rgb 0.85 0.83 0.85}}";

/* load and fit the image */
...

/* retrieve the coordinates of the lower right (second) matchbox corner */
if ((int) p.info_matchbox("giantwing", 1, "exists") == 1)
{
        x2 = p.info_matchbox("giantwing", 1, "x2");
        y2 = p.info_matchbox("giantwing", 1, "y2");
}
/* start the text line at that corner with a small distance of 2 */
p.fit_textline("Foto: Kraxi", x2+2, y2+2, "font=" + font + " fontsize=8 orientate=west");
```

Fig. 9.45 Use the coordinates of the image matchbox to fit a text line

Generated output	*Generation steps*
	*Step 1: Fit image with matchbox* *Step 2: Retrieve matchbox info for coordinates (x2, y2)* *Step 3: Fit text line starting at retrieved coordinates (x2, y2) with option* `orientate=west`

10 Interactive Features

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

10.1 Links, Bookmarks, and Annotations

This section explains how to create interactive elements such as bookmarks, form fields, and annotations. Figure 10.1 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

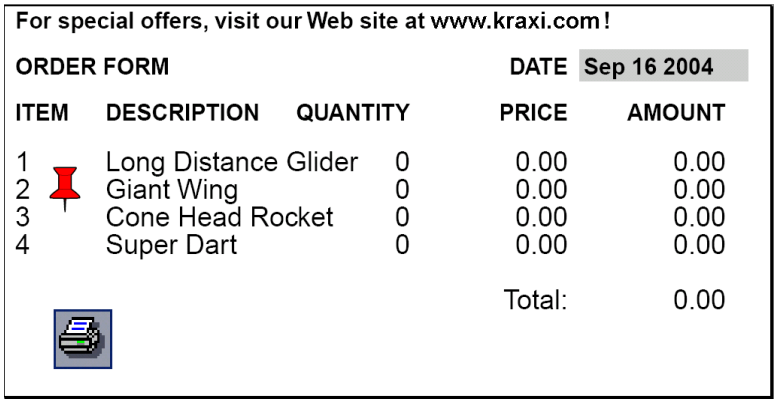


Fig. 10.1
Document with interactive elements

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*.

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

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

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

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

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

p.end_page_ext("");
```

For an example of creating a Web link on an image or on parts of a textflow, see Section 9.4, »Matchboxes«, page 267.

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

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

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

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

In the next step we create the actual bookmark. The *action* option for the bookmark contains the *activate* event which will trigger the action, plus the *goto_action* handle created above for the desired action. The option *fontstyle=bold* specifies bold text, and *textcolor=blue* 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=blue";

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

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

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

Annotation with file attachment. In the next example we create a file attachment. We start by creating an annotation of type *FileAttachment*. The *filename* option specifies the name of the attachment, the option *mimetype image/gif* specifies its type (MIME is a common convention for classifying file contents). The annotation will be displayed as a pushpin *(iconname pushpin)* in red *(annotcolor=red)* 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=red 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.

10.2 Form Fields and JavaScript

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

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

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

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

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

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

```
print_icon = p.load_image("auto", "print_icon.jpg", "template");
if (print_icon == -1)
{
        /* Error handling */
        return;
}
p.begin_page_ext(pagewidth, pageheight, "");
...
String  optlist = "action={up " + print_action + "} icon=" + print_icon +
        " tooltip={Print the document} font=" + button_font;

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

Cookbook A full code sample can be found in the Cookbook topic `form_fields/form_pushbutton`.

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

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

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

Cookbook *Full code samples can be found in the Cookbook topics* `form_fields/form_textfield_layout` *and* `form_fields/form_textfield_height`.

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

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

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

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

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

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

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

Cookbook *A full code sample can be found in the Cookbook topic* `form_fields/form_textfield_fill_with_js`.

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

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

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

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

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

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

Cookbook *A full code sample can be found in the Cookbook topic* `form_fields/form_textfield_input_format`.

In order to specify the various formats which are supported in Acrobat you must use appropriate functions in the JavaScript code. Table 10.1 lists the JavaScript function names for the *keystroke* and *format* actions for all supported formats; the function parameters are described in Table 10.2. These functions must be used similarly to the example above.

Table 10.1 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 10.2 Parameters for the JavaScript formatting functions

parameters	***explanation and possible values***	
nDec	*Number of decimal places*	
sepStyle	*The decimal separator style:*	
	0	*1,234.56*
	1	*1234.56*
	2	*1.234,56*
	3	*1234,56*
negStyle	*Emphasis used for negative numbers:*	
	0	*Normal*
	1	*Use red text*
	2	*Show parenthesis*
	3	*both*
strCurrency	*Currency string to use, e.g.* `\u20AC` *for the Euro sign*	
bCurrency-Prepend	***false***	*do not prepend currency symbol*
	true	*prepend currency symbol*

Table 10.2 Parameters for the JavaScript formatting functions

parameters	*explanation and possible values*	
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*

Validating form field input. The following sample attaches JavaScript to a form field as validate action to check whether the user input for a text field matches the required format *mm/dd/yyyy*:

```
optlist =
"script={" +
        "// JavaScript code for date mask format MM/DD/YYYY\n" +
        "var re = /^[0-9]{2}\\/[0-9]{2}\\/[0-9]{4}$/\n" +
        "if (event.value !=\"\") {\n" +
        " if (re.test(event.value) == false) {\n" +
        " app.alert ({\n" +
        "    cTitle: \"Incorrect Format\",\n" +
        "    cMsg: \"Please enter date using mm/dd/yyyy format\"\n" +
        " });\n" +
        " }\n" +
        "}\n" +
"}";
validate_action = p.create_action("JavaScript", optlist);
textfield_font = p.load_font("Helvetica", "unicode", "");
optlist = "action={validate=" + validate_action + "} " +
```

```
            "backgroundcolor={gray 0.8} font=" + textfield_font;
    p.create_field(llx, lly, urx, ury, "startdate", "textfield", optlist);
```

10.3 Geospatial PDF

Cookbook A full code sample can be found in the Cookbook topic `geospatial/starter_geospatial`.

10.3.1 Using georeferenced PDF in Acrobat

PDF 1.7ext3 allows geospatial reference information (world coordinates) to be added to PDF page contents. Geospatially referenced PDF documents can be used in Acrobat for several purposes. In Acrobat and Adobe Reader DC you must enable *Tools, Measure;* in Acrobat X/XI you have to activate the *Analyze* toolbar using the button at the top of the *Tools* pane; in Adobe Reader X/XI: *Edit, Analysis, Geospatial Location Tool:*

- Display the coordinates of the map point under the mouse cursor: *Geospatial Location Tool*. You can copy the coordinates of the map point under the mouse cursor by right-clicking and selecting *Copy Coordinates to Clipboard;*
- Search for a location on the map: *Geospatial Location Tool*, right-click and select *Find a Location*, and enter the desired coordinates;
- Mark a location on the map: *Geospatial Location Tool*, right-click and select *Mark Location;*
- Measure distance, perimeter and area on geographic maps: *Measuring Tool;*

Only the first two functions mentioned above are available in Adobe Reader. Various settings for geospatial measuring can be changed in *Edit, Preferences, [General...], Measuring (Geo)*, e.g. the preferred coordinate system for coordinate readouts.

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

- One or more georeferenced areas can be assigned to a page with the *viewports* option (and suboption *georeference)* of *PDF_begin/end_page_ext( )*. Viewports allow different geospatial references (specified by the *georeference* option) to be used on different areas of the page, e.g. for multiple maps on the same page. This method works in all PDF viewers with support for georeferenced PDF.

Cookbook A full code sample can be found in the Cookbook topic `geospatial/starter_geospatial`.

- The *georeference* option of *PDF_load_image( )* can be used to assign an earth-based coordinate system to an image.

Cookbook A full code sample can be found in the Cookbook topic `geospatial/georeferenced_image`.

- The *georeference* option of *PDF_open_pdi_page( )*, *PDF_load_graphics( )* and *PDF_begin_template_ext( )* can be used to assign an earth-based coordinate system to a Form XObject. However, this method is not recommended since it is not supported in any known viewer including Acrobat DC.

10.3.2 Geographic and projected Coordinate Systems

A geographic coordinate system describes the earth in geographic coordinates, i.e. angular units of latitude and longitude. A projected coordinate system can be specified on top of a geographic coordinate system and describes the transformation of points in geographic coordinates to a two-dimensional (projected) coordinate system. The resulting coordinates are called Northing and Easting values; degrees are no longer required for projected coordinate systems. While geographic coordinate systems are in use for

GPS and other global applications, projections are required for map-making and other applications with more or less local character.

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

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

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

```
www.epsg.org
```

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

```
www.opengeospatial.org/standards/ct
```

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

```
www.spatialreference.org/ref/epsg
```

10.3.3 Coordinate System Examples

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

```
worldsystem={type=geographic wkt={
GEOGCS["WGS 84",
    DATUM["WGS_1984", SPHEROID["WGS 84", 6378137, 298.257223563]],
    PRIMEM["Greenwich", 0],
    UNIT["degree", 0.01745329251994328]]
}}
```

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

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

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

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

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

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

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

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

10.3.4 Georeferenced PDF Restrictions in Acrobat

We experienced the following shortcomings when working with georeferenced PDF in Acrobat X/XI/DC:

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

readouts.
Workaround: if you need multiple instances of the same image-based map on the same page, open the image multiply.

- The area measurement tool doesn't work correctly for geographic coordinate systems, but only for projected systems.

11 Document Interchange

11.1 XMP Metadata

As an alternative or in addition to document information fields PDFlib supports XMP *(Extensible Metadata Platform)* as a framework for specifying metadata. XMP has been standardized as ISO 16684-1:2012. There are several aspects of XMP support in PDFlib as detailed below.

Cookbook *A simple XMP sample can be found in the Cookbook topic* `interchange/embed_xmp`.

Most commonly XMP is used to attach metadata to the whole document. In addition to document-level metadata, XMP can be supplied for pages, fonts, ICC profiles, images, graphics, templates, and imported PDF pages. This can be achieved with the *metadata* option of various functions, for example:

```
metadata={filename=info.xmp inputencoding=winansi}
```

The *metadata* option expects full or partial XMP metadata streams. PDFlib validates the user-supplied XMP metadata according to XML and XMP/RDF rules. For PDF/A additional rules for custom XMP properties apply; see Section 12.3.8, »XMP Metadata for PDF/A«, page 328.

Internal and reserved XMP properties. PDFlib creates several XMP properties internally, e.g. *CreationDate*. Other XMP properties are required to signal conformance to various PDF standards such as PDF/A or PDF/X. Internal properties as well as standards-related identification properties can not be overridden by user-supplied XMP.

Automatic XMP synchronization for document info fields. If the *autoxmp* option in *PDF_begin/end_document( )* is *true*, PDFlib synchronizes document information fields supplied to *PDF_set_info( )* as well as several internally generated entries (e.g. *CreationDate)* to the corresponding entries in the document-level XMP metadata.

Document info fields which correspond to a well-known property in one of the standard XMP schemas are placed in the appropriate schema. Unknown info fields are generally placed in the extended PDF *(pdfx)* schema, but will be ignored in PDF/A.

Cloning XMP metadata. If most or all pages of a PDF document are imported it is recommended to clone XMP metadata if it is present in the input. XMP metadata can be cloned with the following code fragment:

```
if (p.pcos_get_string(indoc, "type:/Root/Metadata").equals("stream"))
{
        xmp = p.pcos_get_stream(indoc, "", "/Root/Metadata");
        p.create_pvf("/xmp/document.xmp", xmp, "");
        optlist += " metadata={filename=/xmp/document.xmp}";
}

p.end_document(optlist);
p.delete_pvf("/xmp/document.xmp");
```

11.2 Web-Optimized (Linearized) PDF

PDFlib can apply a process called linearization to PDF documents (linearized PDF is also called *Optimized* or *Fast Web View*). Linearization reorganizes the objects within a PDF file and adds supplemental information which can be used for faster access.

While non-linearized PDFs must be fully transferred to the client, a Web server can transfer linearized PDF documents one page at a time using a process called 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. This can be achieved with the *linearize* option in *PDF_begin_document( )* as follows:

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

  In Acrobat you can check whether a file is linearized by looking at its document properties (»Fast Web View: yes«).
- The Web server must support byteserving. The underlying byterange protocol is part of HTTP 1.1 and therefore implemented in all current Web servers.
- The user must use Acrobat as a Browser plugin, and have page-at-a-time download enabled in Acrobat (*Edit, Preferences, [General...,] Internet, Allow fast web view)*. Note that this is enabled by default.

The larger a PDF file (measured in pages or MB), the more it will benefit from linearization when delivered over the Web.

Linearization and file size. Since linearization aims at improving the Web-based display of large PDF documents it doesn't make much sense for single-page documents. Due to a bug in Acrobat small linearized documents are not always treated as linearized. For example, Acrobat regards documents < 4KB as non-linearized, regardless of the actual linearization status.

Acrobat also doesn't regard PDF documents larger than 2 GB as linearized.

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.

11.3 Tagged PDF Basics

Tagged PDF is a requirement for the PDF/UA, PDF/A-1a, PDF/A-2a, and PDF/A-3a ISO standards, Section 508 in the U.S., BITV in Germany and many other regulations. Tagged PDF enhances PDF with document structure information which offers the following advantages:

- Accessibility: Tagged PDF is accessible for users with disabilities, e.g. via Acrobat's built-in Read Aloud feature or more advanced screen reader software (see Figure 11.1).
- Reliable export and conversion to other document formats: converting Tagged PDF to other formats such as RTF, XML or HTML results in more accurate output.

PDF/UA enhances Tagged PDF by specifying requirements for the document tags. If you want to create accessible PDF documents we recommend to follow the additional rules for PDF/UA; see Section 12.6, »PDF/UA for Universal Accessibility«, page 348, for details.

If you cannot obey all PDF/UA requirements (e.g. because you must assemble documents from existing PDFs which themselves don't conform to PDF/UA) we recommend to disable PDF/UA mode and obey as many PDF/UA rules as possible.

Cookbook *Code samples for generating Tagged PDF can be found in the* `pdfua` *category of the PDFlib Cookbook. All Tagged PDF samples in the Cookbook create PDF/UA.*

11.3.1 The Logical Structure Tree (Structure Hierarchy)

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. In order to create Tagged PDF the *tagged* document option must be set to *true*, and the *lang* option is recommended:

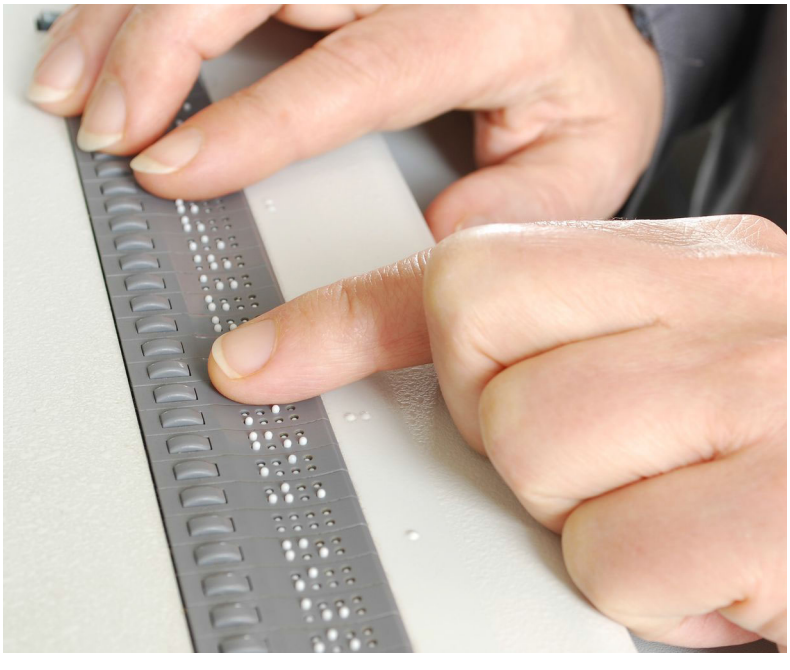


Fig. 11.1
A screen reader captures text on the screen and displays it on a Braille device

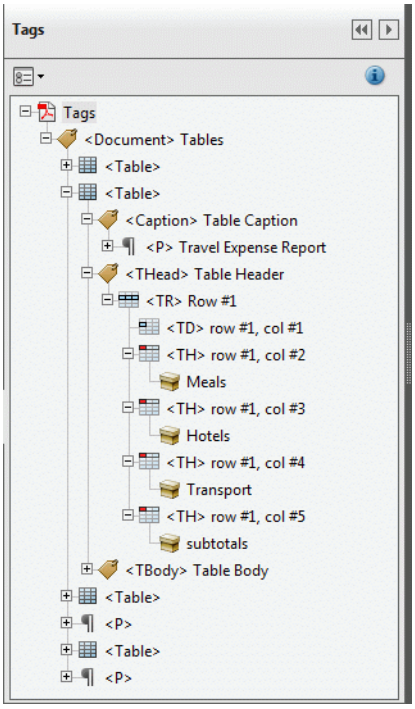


Fig. 11.2
Acrobat's tag pane displays the document's logical structure tree

```
if (p.begin_document("tagged.pdf", "tagged=true lang=en") == -1) {
        throw new Exception("Error: " + p.get_errmsg());
```

The logical structure in a Tagged PDF document is described by a hierarchy of elements, called the structure hierarchy or logical structure tree. Starting at the root level, the structure hierarchy consists of an arbitrary number of levels. On each level an element may contain zero or more items of the following kinds:

- Other structure elements, e.g. the *Document* element may contain multiple *Art* (article) elements. Each *Art* element in turn may contain multiple *P* (paragraph) elements.
- Content items, i.e. sequences of text and graphics on the page, XObjects created from imported images, and OBJR references to annotations and form fields. These items represent the graphical content associated with a structure element.

Structure hierarchy in Acrobat. You can display tag names and structure hierarchy in Acrobat X/XI/DC as follows:

- select *View, Show/Hide, Navigation Panes, Tags* (see Figure 11.2)

Tag nesting functions. Structure elements can be created explicitly with the *PDF_begin_item()* function which must always be paired with a corresponding call to *PDF_end_item()*. For example, the code fragment below creates a hierarchy consisting of a section which contains a heading and a paragraph. The hierarchical relationship is visualized with indentation:

```
/* Create a structure element of type "Sect" (section) */
id_sect = p.begin_item("Sect", "Title={Past and Future}");
```

```
        /* Create a structure element of type "H1" (heading) */
        id_h1 = p.begin_item("H1", "Title={Company History}");
            p.fit_textline(...);
        p.end_item(id_h1);

        /* Create a structure element of type "P" (paragraph) */
        id_p = p.begin_item("P", "");
            p.fit_textline(...);
        p.end_item(id_p);

/* Close "Sect" */
p.end_item(id_sect);
```

By default, structure elements are inserted as a child of the currently active item after all other child items which may already be present. This leads to the correct tree structure if the elements are created in logical order. See Section 11.4.4, »Creating Contents out of Order«, page 309, for more advanced techniques.

Structure elements may carry one or more attributes which are supplied via options, e.g. *lang* for specifying the natural language of a content element or *Alt* for alternative text for images. The set of available options depends on the structure element type.

Abbreviated tagging. In many situations the content of a structure element can be created with a single call to a PDFlib fitting function, resulting in the typical sequence *PDF_begin_item()*/*PDF_fit_*()*/*PDF_end_item()*. This sequence can be reduced with abbreviated tagging. Placing content and supplying tagging information can be combined in a single function call using the *tag* option which is supported by most functions for creating page content. The code fragment above can be simplified by supplying the *tag* option to the text and image placement functions:

```
/* Create a structure element of type "Sect" (section) */
id_sect = p.begin_item("Sect", "Title={Past and Future}");

        /* Create a structure element of type "H1" (heading) */
        p.fit_textline(..., "tag={tagname=H1 Title={Company History}}");

        /* Create a structure element of type "P" (paragraph) */
        p.fit_textline(..., "tag={tagname=P}");

/* Close "Sect" */
p.end_item(id_sect);
```

Since the tag nesting function *PDF_begin_item()* also supports the *tag* option both methods can be combined. This is useful in situations where a tag requires a nested tag which in turn contains multiple content items. For example, a *Caption* element containing a *P* element which in turn contains multiple text items can be created as follows:

```
id_caption = p.begin_item("Caption", "tag={tagname=P}");
        p.fit_textline(...);
        p.fit_textline(...);
p.end_item(id_caption);
```

A second *tag* option can also be nested as suboption in the option list of a *tag* option. Indeed, *tag* options can be nested to an arbitrary level. If a nested structure element con-

tains only a single content item the code fragment above can be further simplified by applying abbreviated tagging directly in the call for placing the text on the page:

```
p.fit_textline(..., "tag={tagname=Caption tag={tagname=P}");
```

Since abbreviated tagging does not expose the handle of the generated structure element it cannot be used with *PDF_activate_item( )* or the *parent* suboption of the *tag* option since these require a structure element id. The detailed operation of the *tag* option is as follows:

- A new structure element is created for the generated content, and closed before returning from the call. The following situations are exceptions from this rule:
  - A structure element created with the *tag* option in *PDF_begin_document( )* will be closed in *PDF_end_document( )*.
  - When multiple tags are supplied with the *tag* option of *PDF_begin_item( )* all of these tags will be closed in the corresponding call to *PDF_end_item( )*.
- *PDF_fit_table( )*: *tagname=Table* or a tag name which is role-mapped to *Table* in *PDF_fit_table( )* instructs PDFlib to create the required table tags (see Section 11.4.1, »Automatic Table Tagging«, page 301). Automatically generated *TH* and *TD* tags for table cells can be further qualified with the *tag* option of *PDF_add_table_cell( )*.
- *PDF_fit_textflow( )*: the complete Textflow instance forms the new structure element.
- The generated element is a child of the currently active item or the item which has been supplied in the *parent* option.
- Grouping elements can only be created with *PDF_begin_item( )*, but not with abbreviated tagging except *PDF_begin_document( )*.
- Except for *PDF_begin_document( )* and *PDF_add_table_cell( )* abbreviated tagging can only be used in *page* scope.

11.3.2 Standard and custom Element Types

Standard element types. PDF supports standard element types which are designed to support a wide range of document classes. PDFlib supports all these standard elements types according to Table 11.1. The descriptions provided in the table are intended to assist in selecting appropriate types. Table 11.1 also contains the BLSE/ILSE distinction which is explained in »Regular elements vs. direct elements«, page 293.

Grouping elements are containers which hold other structure elements. They cannot contain direct page contents. A *Document* element should be used as root of the structure tree if the PDF contains a complete document. *Part* or *Sect* should be used as root of the structure tree if the PDF contains a fragment of a document. The root element can conveniently be provided with the *tag* option of *PDF_begin_document( )*.

Pseudo element types do not create any structure element, but are used for marking up content with certain characteristics. They are mainly used for marking up Artifacts (see Section 11.3.3, »Artifacts«, page 294).

Table 11.1 Standard element types (tags) in Tagged PDF and pseudo element types added by PDFlib

type	*description*
Grouping (container) elements (cannot contain direct page content)	
Document	*Complete document; recommended as root element of the structure tree*

Table 11.1 Standard element types (tags) in Tagged PDF and pseudo element types added by PDFlib

type	description
Part	*Encloses a grouping of structure elements without consideration for their hierarchy (semantic equivalent of* `Div`*).*
Sect	*(Section) Encloses a grouping of structure elements with consideration for their hierarchy.*
Div	*(Division) Generic block-level element or group of elements. It should be used for non-semantic grouping, such as associating a* `lang` *attribute with a sequence of block-level elements, or for role-mapping a custom grouping structure element for which none of the other grouping elements are suitable.*
Caption	*(Caption) Brief portion of text describing a table, list or figure*
Heading and paragraph elements (BLSEs)	
H	*(Heading; in PDF/UA-1 only with* `structuretype=strong`*) Heading for a subdivision of a document's content. It should be the first child of its parent.* `H` *elements are intended for hierarchical nesting. This element is not recommended; weak structuring with* `H1...H6` *should be used instead.*
H1...H6	*Headings with specific levels for weakly structured documents. They should be used if the application cannot hierarchically nest sections and thus cannot determine the level of a heading from its level of nesting.*
P	*(Paragraph) Generic paragraph element, i.e. a low-level division of text which is not a heading*
Label and List elements (BLSEs)	
L	*(List) Sequence of items of like meaning and importance*
LI	*(List item) Individual member of a list*
Lbl	*(Label) Name or number that distinguishes a given item from others in the same list or other group of like items. For example, in a bulleted or numbered list, it contains the bullet character or the number of the list item and associated punctuation.*
LBody	*(List body) Descriptive content of a list item*
Table elements (all table tags can be created automatically, see Section 11.4.1, »Automatic Table Tagging«, page 301)	
Table	*(BLSE) Two-dimensional layout of rectangular cells, possibly having a complex substructure*
TR	*(Table row) Row of headings or data in a table*
TH	*(Table header cell) Table cell containing header text describing one or more rows or columns of the table*
TD	*(Table data cell) Table cell containing data that is part of the table's content*
THead	*(Table header row group; PDF 1.5) Group of rows that constitute the header of a table*
TBody	*(Table body row group; PDF 1.5) Group of rows that constitute the main body portion of a table*
TFoot	*(Table footer row group; PDF 1.5) Group of rows that constitute the footer of a table*
Inline element	
Span	*(Not related to row or column spans in tables) Generic inline portion of text having no particular inherent characteristics.*
Elements for interactive features (see Section 11.4.2, »Tagging Interactive Elements«, page 304)	
Link	*Association between a portion of the element's content and one or more corresponding link annotations*
Annot	*(PDF 1.5) Association between a portion of the element's content and a corresponding PDF annotation. It shall be used for all annotations unless* `Link` *or* `Form` *is more appropriate*
Form	*Interactive form field*

Table 11.1 Standard element types (tags) in Tagged PDF and pseudo element types added by PDFlib

type	description
Illustration elements	
Figure	*Item of graphical content*
Formula	*Mathematical formula. This element type identifies an entire content element as a formula. If the formula is represented as an image, the* `Formula` *element must still be used (and not* `Figure`*).*
Elements for Japanese Ruby and Warichu (PDF 1.5)	
Ruby	*A side note written in a smaller text size and placed adjacent to the base text to which it refers. The* `Ruby` *element serves as wrapper around the entire ruby assembly.*
RB	*(Ruby base text) Full-size text to which the ruby annotation is applied.*
RT	*(Ruby annotation text) The smaller-size text that shall be placed adjacent to the ruby base text.*
RP	*(Ruby punctuation) Punctuation surrounding the ruby annotation text. It is used only when a ruby annotation cannot be properly formatted in a ruby style and instead is formatted as a normal comment, or when it is formatted as a warichu.*
Warichu	*(Warichu) Comment or annotation in a smaller text size and formatted onto two smaller lines within the height of the containing text line and placed following (inline) the base text to which it refers. The* `Warichu` *element serves as wrapper around the entire warichu assembly.*
WT	*(Warichu text) Smaller-size text of a warichu comment that is formatted into two lines and placed between surrounding* `WP` *elements*
WP	*(Warichu punctuation) Punctuation that surrounds the* `WT` *text*
Non-structural	
NonStruct	*(Nonstructural element) Grouping element having no inherent structural significance; it serves solely for grouping purposes*
Pseudo element types	
Artifact	*Artifact, to be distinguished from real page content (see Section 11.3.3, »Artifacts«, page 294).*
ASpan	*(Accessibility span; written to PDF as* `Span`*, but must be distinguished from the inline item* `Span`*) Attach accessibility properties to content which does not belong to a structure element or which resembles a part of a structure element. The* `ASpan` *pseudo element is written as* `Span` *with an accessibility attribute such as* `Alt`, `ActualText`, `Lang`*, or* `E`*. An* `ASpan` *is not associated with any structure element.*
ReversedChars	*(Not recommended) Specifies text in a right-to-left script with reversed characters.*
Clip	*(Not recommended) Specifies a marked clipping sequence. This is a sequence containing only clipping paths or text in text rendering mode 7, but no visible graphics or PDF_save( ) / PDF_restore( ).*

Deprecated standard element types in PDF 2.0. The element types listed in Table 11.2 were available in PDF 1.7/ISO 32000-1, but are deprecated in PDF 2.0/ISO 32000-2 and should not be used.

Table 11.2 Deprecated standard element types (tags) as of PDF 2.0; these types should not be used

type	description
Art	*(Article) A relatively self-contained body of text constituting a single narrative or exposition*
BibEntry	*(Bibliography entry) Reference identifying the external source of some cited content*
BlockQuote	*(Block quotation) Portion of text consisting of one or more paragraphs attributed to someone other than the author of the surrounding text*

Table 11.2 Deprecated standard element types (tags) as of PDF 2.0; these types should not be used

type	*description*
Code	*Fragment of computer program text*
Index	*(Index) Sequence of entries containing identifying text accompanied by* `Reference` *elements that point out occurrences of the specified text in the main body of the document*
Note	*Item of explanatory text, such as a footnote or an endnote, that is referred to from within the body of the document. This element may have* `Lbl` *as a child.*
Private	*(Private element) Grouping element containing private content belonging to the application*
Quote	*(Quotation) Inline portion of text attributed to someone other than the author of the surrounding text. The quoted text should be contained inline within a single paragraph.*
Reference	*Citation to content elsewhere in the document. It should be used for local links.*
TOC	*(Table of contents) List made up of table of contents items (element type* `TOCI`*)*
TOCI	*(Table of contents item) Member of a table of contents. It should contain a suitable* `Link` *element.*

Nesting rules for structure elements. Various rules must be observed for creating structure elements. These rules are summarized in Table 11.3. The rules apply to the listed standard element types as well as to custom element types which are role-mapped to the respective standard types (see »Custom element types and role map«, page 293). Additional rules apply to PDF/UA-1 (see Section 12.6, »PDF/UA for Universal Accessibility«, page 348).

The nesting rules for new structure elements can be disabled or relaxed with the *checktags* option of *PDF_begin_document( )*. However, this is not recommended since it may result in an invalid structure hierarchy. This option is intended as a migration aid for legacy applications. Some rules in Table 11.3 are marked as »strict rule«. The option *checktags=relaxed* enforces all but the strict rules.

Table 11.3 Tag nesting rules for PDF_begin_item( ) and the `tag` *option of various functions*

item	*rule*

content items	*The following elements may contain page content, i.e. text, image or vector graphics:*
	`H, H1, H2, ...`
	`P`
	`Lbl, LBody`
	`TH, TD`
	`Span, Quote`[1]`, Note`[1]`, Reference`[1]`, BibEntry`[1]`, Code`[1]
	`Link, Annot`
	`Figure, Formula`
	`RB, RT, RP, WT, WP`
	`Artifact, ASpan, ReversedChars, Clip`
	All other elements require an intermediate structure element before direct page content can be added. PDFlib throws an exception when attempting to add content items.

grouping elements	*Grouping elements must not contain content items,* `ASpan` *or ILSEs as children, i.e. a BLSE must be created before content can be created:* `Document, Part, Art`[1]`, Sect, Div, BlockQuote`[1]`, Caption, TOC`[1]`, TOCI`[1]`, Index`[1]`, NonStruct, Private`[1] *The option* `Placement=Block` *is recommended for the following elements as children of grouping elements:* `Figure, Formula, Form, Link, Annot.`

Table 11.3 Tag nesting rules for PDF_begin_item( ) and the tag *option of various functions*

item	*rule*
block-level elements	*The following block-level elements must not contain content items, i.e. a suitable grouping element or BLSE must be created before content can be created:* L, LI, Table, TR, THead, TFoot, TBody *Strict rule: the* P *element may not contain grouping elements.*
pseudo and inline elements	*Pseudo elements (i.e.* Artifact, ASpan, ReversedChars, Clip*) and the following ILSEs cannot have any descendants if* direct=true*:* Code[1], BibEntry[1], Note[1], Quote[1], Reference[1], Span *However, these elements may have children if* direct=false. *Artifacts cannot be created within pseudo elements and the following ILSEs with* direct=true*:* Span,Quote[1], Note[1], Reference[1], BibEntry[1], Code[1]
table elements	Table *elements may contain one or more* TR *elements, or an optional* THead *followed by one or more* TBody *elements and an optional* TFoot. TBody *should not be the only child of* Table. *In addition,* Table *elements may have a* Caption *element as its first or last child.* TR *elements may contain* TH *and* TD *elements.* TH *and* TD *elements may not contain* TR, TH, TD, THead, TBody, *or* TFoot THead, TBody, TFoot *elements can contain only* TR *elements, and can only have* Table *as parent.* TR *may only have* Table, THead, TBody, *or* TFoot *as parent.*
list elements	L *elements may optionally contain a* Caption *element and one or more* LI *elements.* LI *elements may contain one or more* Lbl *or* LBody *elements or both.* LI *may only have* L *as parent.* LBody *may only have* LI *as parent.*
table of contents[1]	TOC[1] *elements may contain an optional* Caption *element as first child, and one or more* TOCI *and* TOC *elements (also in combination).* TOCI[1] *elements may contain only* Lbl, Reference, NonStruct, P, *and* TOC *elements.* TOCI *may only have* TOC *as parent.*
label element	Lbl *may only have* Annot, LI, Link, TOCI, BibEntry[1], *or* Note[1] *as parent.*
interactive elements: links, form fields and annotations	*The following element types are not allowed for the* tag *option of PDF_create_field( ) and PDF_create_annotation( ) or as parent for* Link, Annot, *and* Form*: table elements, ILSEs, Ruby and Warichu elements, pseudo elements.* Annot *elements cannot be nested.* Link *elements cannot be nested.* Form *elements must not contain any elements other than the* OBJR *element which is created by PDF_create_field( ) automatically.*
Japanese Ruby and Warichu	Ruby *can have* RB, RT, *and* RP *as children, but not any other element types.* Warichu *can have* WT *and* WP *as children, but not any other element types.*
PDF 2.0 consistency rules	*The following nesting rules for rarely used PDF 1.7 parent/child tag combinations have been introduced for consistency with the PDF 2.0 nesting rules:* *Incompatible changes (stricter nesting rules):* ► RB, RT, *and* RP *can only be children of* Ruby ► WT *and* WP *can only be children of* Warichu ► Document *may not contain* Caption ► Caption *is no longer allowed to contain* Document *and* Caption ► H, H1 *etc. are no longer allowed to contain* Document, Part, *and* Div ► TH *and* TD *are no longer allowed to contain* Document *and* Caption ► *Inline elements with* direct=false *are no longer allowed to contain* L, Table *or* Document

1. Deprecated in PDF 2.0

Regular elements vs. direct elements. Most structure elements are emitted to PDF by marking up the corresponding text or graphics and adding a corresponding entry in the document structure tree. Such elements are displayed in Acrobat's tags pane; they are called regular elements.

In contrast, an element may consist only of marked content without any entry in the structure tree. Such elements are not visible in Acrobat's tags pane. The advantage of such direct elements is that they require fewer bytes in the PDF output. The status of some structure types can be changed with the *direct* option (the default is *true)*. Table 11.4 compares regular structure elements and direct elements. The option *direct=false* is required in cases where an element which is direct by default contains another indirect element as child, e.g. *Reference* contains *Link*.

Table 11.4 Regular and direct items

	regular elements (direct=false)	***direct elements (direct=true)***
affected items	*all other element types*	`Code`, `BibEntry`, `Note`, `Quote`, `Reference`, `Span`, `Lbl` *as child of* `BibEntry`, `TOCI`, *or* `Note` *pseudo items* `ASpan`, `ReversedChars`, `Clip`
part of the structure tree	*yes*	*no*
can extend across page boundaries	*yes*	*no*
can be interrupted by other items	*yes*	*no*
can be activated with `PDF_activate_item()`	*yes*	*no*
nesting rules for child elements	*may contain regular and direct elements*	*may contain only direct elements, and only if the parent can contain direct contain*

Empty structure elements. In general, structure elements which contain neither child elements nor content items should be avoided. However, there are some exceptions to this rule in cases where an empty structure element conveys a semantic role in some other structure as in the following examples:

- Empty *TD* elements are required for empty table cells. This rule is honored by PDFlib's automatic table formatting (see Section 11.4.1, »Automatic Table Tagging«, page 301).
- Empty *LI* elements inside a list structure.

Programming scope and page boundaries. Most structure elements can only be created in *page* scope. Grouping elements may span multiple pages and can also be created in *document* scope. Pseudo items as well as direct items must be closed before ending or suspending a page.

Custom element types and role map. In addition to the predefined standard structure element types listed in Table 11.1 custom element type names can be used. Custom element type names are typically used to localize element type names (e.g. German *Abbildung* maps to *Figure)* or to work with application-specific type names (e.g. *Normal* maps to *P)*. In order to facilitate repurposing documents which contain custom element type names the custom names must be mapped to their exact or approximate equivalent in the set of standard structure element types. You can also remap standard ele-

ment types to other standard types in order to change their semantics. Custom elements types cannot be mapped to inline and pseudo elements. Element mapping is achieved with the *rolemap* document option, e.g.

```
p.begin_document("tagged.pdf",
        "tagged=true lang=en rolemap={ {Heading H1} {Subhead H2} {Paragraph P} }");
```

Role map in Acrobat. You can view and edit the role map in Acrobat X/XI/DC as follows:

- Select *View, Show/Hide, Navigation Panes, Tags,* click the menu button at the top of the *Tags* pane, and select *Edit Role Map* from the drop-down list.

11.3.3 Artifacts

Relevant content and Artifacts. The contents of a page fall into one of the following categories:

- Relevant content has been created by the document author to convey the document's meaning. The document's logical structure tree describes the objects which comprise real content, and may also contain annotations.
- Graphic or text objects which do not contribute relevant page content, but have been created for pagination or layout purposes are called Artifacts. Artifacts are not included in the structure tree and are not read by a screen reader.

Marking Artifacts is strongly recommended in order to improve accessibility and is required in PDF/UA-1. Typical Artifacts are repeated headers and footers, page numbers, background images, and other items which are repeated on each page.

Artifacts in Acrobat. You can check Artifacts in Acrobat X/XI/DC with one of the following methods:

- Select *Tools, Accessibility, Reading Order* (Acrobat DC) or *TouchUp Reading Order* (Acrobat X/XI) to display or edit content elements on the page. Artifacts are called *Background* in Acrobat's *Reading Order* Tool. Unlike structure elements, they are not visualized with a frame and tag name when this tool is activated.
- Identify Artifacts by selecting *View, Show/Hide, Navigation Panes, Content*. The *Content* pane lists all page contents along with the respective structure element type name or *Artifact* as appropriate. Because Artifacts are not read, there is no corresponding numbered block on the page. However, clicking on an Artifact in the list highlights the corresponding content element on the page.
- Search for Artifacts: select *View, Show/Hide, Navigation Panes, Tags,* click the menu button at the top of the *Tags* pane, *Find...*, and select *Artifacts* from the drop-down list. Since Artifacts are not part of the document structure, there is no corresponding entry in the Tags navigation pane nor the Order pane.
- Activate *View, Read Out Loud...* to let Acrobat read the structure elements on the page. Artifacts will not be read.

Designating content as Artifact. Artifacts can be specified in PDFlib with the *Artifact* tag name in *PDF_begin_item()* (despite the fact that Artifacts are not actually tags in the sense of structure elements):

```
id = p.begin_item("Artifact", "");
```

Alternatively, Artifacts can be specified with abbreviated tagging, i.e. with the *tag* option of various functions:

```
p.fit_textline(text, x, y, "tag={tagname=Artifact}");
```

It is recommended to create Artifacts only if no BLSE is currently active. However, since this may not always be possible for applications, PDFlib automatically interrupts the currently active element when an Artifact is created and activates it again after the Artifact. Note that the tag nesting rules (see »Nesting rules for structure elements«, page 291) don't allow Artifacts within ILSEs.

Classifying Artifacts. Irrelevant page content should be identified with the *Artifact* pseudo tag, and classified according to one of the following keywords of the *artifacttype* option:

- *Pagination* Artifacts: page features such as running heads and page numbers. Pagination Artifacts can be further classified with the *artifactsubtype* option and one of the keywords *Header*, *Footer*, *Watermark*.
- *Layout* Artifacts: typographic or design elements such as rules and table shadings;
- *Page* Artifacts: production aids, such as trim marks and color bars;
- *Background* Artifacts: images or colored areas that run the entire width or height of the page or the entire dimensions of a structure element.

Pagination and Background Artifacts support the *Attached* option which specifies to which page edge or edges the Artifact is attached *(Top/Bottom/Left/Right)*.

The following example creates a pagination Artifact with subtype *Header:*

```
id = p.begin_item("Artifact",
        "artifacttype=Pagination artifactsubtype=Header Attached={Top Left}");
```

Automatic Artifact tagging. Since all page contents should be tagged either as structure element or as Artifact PDFlib automatically tags certain graphical elements. The following decorative elements are automatically tagged as *Artifact* with *artifacttype=Layout* in Tagged PDF mode:

- Block decoration: all decorative elements created by *PDF_fill_*block()*, i.e. stroking and filling created according to the *backgroundcolor* and *bordercolor* properties.
- Matchbox decoration: matchbox rectangles created according to the matchbox options *fillcolor*, *shading*, and *strokecolor*
- Table decoration: if automatic table tagging is active (see Section 11.4.1, »Automatic Table Tagging«, page 301), table ruling and shading, i.e. stroking and filling according to the options *fill, stroke, showborder, showgrid*
- Textline Artifacts: *leader, shadow, showborder*
- Textflow Artifacts: *leader, shadow, showborder, showtabs*
- Text decoration: *underline, overline, strikeout*

Like all Tagged PDF features, automatic Artifact tagging works only in *page* scope.

11.3.4 Text Handling

Language specification. In Tagged PDF the natural language of text should be specified explicitly; this allows screen readers to switch to the appropriate language when reading the document. The natural language can be specified on different levels:

- The *lang* option in *PDF_begin_document( )* should be set to specify the primary language, i.e. the natural language of the document as a whole. This specification covers page contents as well as interactive elements such as bookmarks and annotations.
- The document language can be overridden for individual items on any structure level with the *lang* option in *PDF_begin_item( )* or the *tag* option of various functions.
- Hypertext strings may include Unicode escape sequences for specifying the language (see below).

Contents which are encoded as text but are not part of a natural language, such as programming code, musical notes, type samples or mathematical notation, should use an empty language code, e.g. *lang={ }*.

Unicode language identifiers are comprised of the following sequence:

- The Unicode value U+001B (two bytes).
- An ISO 639 language code (two ASCII bytes), e.g. *en, ja, de*.
- Optionally an ISO 3166 country code (two ASCII bytes), e.g. *US, JP*.
- The Unicode value U+001B (two bytes).

Examples in hexadecimal notation:

```
001B656E5553001B          (=enUS)
001B7A68001B              (=zh)
001B6465001B              (=de)
```

In Unicode-capable languages such as Java the two ASCII characters must be packed to form a single Unicode value:

```
\u001B\u6465\u001B        (=de)
```

In the C language the two ASCII characters must be packed into a single Unicode character. With *charref=true* the sequence can be expressed as follows:

```
&#x001B;&#x6465;&#x001B;     (=de)
```

Generating Tagged PDF with Textflow. The Textflow feature (see Section 9.2, »Multi-Line Textflows«, page 231) offers powerful features for text formatting. Since individual text fragments are no longer under client control, but will be formatted automatically by PDFlib, some care must be taken when generating Tagged PDF with Textflows:

- The complete contents of a single Textflow fitbox may be part of a structure element. However, a Textflow box can not contain individual structure elements.
- 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 may be spread over several pages which may contain other structure items, attention should be paid to choosing the proper parent item (rather than using a *parent* option 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 position the annotation correctly in the structure tree.

Separate words with space characters. Words should be separated by space characters (U+0020). The *autospace* option can be used for automatically generating space characters after each call to one of the text output functions.

Hyphenation. Hyphenation, i.e. splitting a word in two parts at the end of a line, must be represented using a soft hyphen character (U+00AD) as opposed to a hard hyphen or minus character (U+002D). The soft hyphen character ensures that Acrobat can correctly recombine (dehyphenate) the hyphenated word when searching for text. When text contains a soft hyphen U+00AD PDFlib's text engine uses the glyph for U+00AD if it is available in the font, and U+002D otherwise. If the font contains two separate glyphs for U+00AD and U+002D using the soft hyphen U+00AD in the text is sufficient for fulfilling Tagged PDF requirements for hyphenation.

If the font doesn't contain a separate glyph for U+00AD (or another hyphenation character is used), the hyphen must be tagged as *Span* or *ASpan* with an *ActualText* attribute containing U+00AD (note that this is not possible in PDF 1.4). When multiline text is created with Textflow the *ActualText* is assigned automatically to the specified *hyphenchar* (and *autospace* is suppressed).

Attaching the required *ActualText* for Textlines can be achieved as follows:

- In *PDF_fit_textline( )* the option *tagtrailinghyphen* results in suitable *ActualText* and suppressed *autospace*. Since the option defaults to U+00AD proper tagging happens automatically if the text uses U+00AD as hyphen.
- If a font does not contain a separate glyph for U+00AD you can amend it with the soft hyphen from a suitable fallback font using the following font loading option:

```
fallbackfonts={{fontname=AuxiliaryFont encoding=unicode embedding forcechars=x00AD}}
```

- Suitable *ActualText* can be assigned manually as follows; keep in mind that this is only required if the font doesn't contain two separate glyphs for U+00AD and U+002D (this code works only with PDF 1.5 or above; add the option *direct=false* for PDF 1.4):

```
p.set_option("charref=true");
p.fit_textline("-", x, y, "tag={tagname=ASpan ActualText=&#x00AD;}");
```

11.3.5 Alternate Description, Replacement Text and Abbreviation Expansion

Tagged PDF provides features which enhance accessibility of images and text which cannot easily be read without additional information.

Alternate Description (Alt). Items which do not translate naturally to text can be assigned an alternate description via the *Alt* option. Examples are images, formulas, and annotations without the *contents* option.

The alternate description should consist of a whole word or phrase which can be read by a screen reader. The description should end with a period or space character as appropriate to ensure that screen readers don't merge it with subsequent text. It is recommended to avoid initial phrases such as *»This image shows...«* in the alternate description. The *Alt* value provides a description of the structure element and all of its children. The *ASpan* pseudo element can be used to assign *Alt* to some part of a structure element.

For example, the image of a company logo could be described via the *Alt* option:

```
p.fit_image(image, x, y, "tag={tagname=Figure Alt={Kraxi company logo }}")
```

Replacement text (ActualText). Items which translate to text, but where the text is represented in some non-standard way can be assigned replacement text via the *ActualText* option. Examples are illustrations with swash characters or drop caps, and images which use pixels to represent a word. On the other hand, OCR text for a scanned page should not be supplied as *ActualText*, but rather as invisible text (i.e. *textrendering=3)*.

The replacement text should contain one or more characters which are equivalent to what a person would see when viewing the content. The *ActualText* value serves as replacement for the structure element and all of its children. The *ASpan* pseudo element can be used to assign *ActualText* to some part of a structure element.

For example, the symbolic *flower* glyph for which no Unicode value is available can be assigned suitable *ActualText* to make it clear that this glyph is actually used as a bullet character U+2022:

```
p.fit_textline("&.flower;", x, y, "tag={tagname=ASpan ActualText={&#x2022;} } ...");
```

Nesting rules for alternate and replacement text. The following rules must be obeyed when using the *Alt* and *ActualText* attributes:

- Since *Alt* and *ActualText* cover the full sub-hierarchy below the affected structure element, both options are not allowed for a structure element if any of its ancestors in the structure hierarchy already contains an *ActualText* or *Alt* attribute.
- An element with an *Alt* or *ActualText* attribute must include content items or one or more *Link* elements; if the attribute is applied to a *Link* element it must contain content items or one or more *OBJR* references created by *PDF_create_annotation( )* (see »Links and other annotation types«, page 304). Otherwise it would be impossible to determine the page on which the attribute should be read. This rule applies to the element itself; it is not sufficient to have child elements with content items.

Alternate and replacement text in Acrobat. You can display the *Alt* and *ActualText* attributes of a structure element with Acrobat X/XI/DC as follows:

- Select *View, Show/Hide, Navigation Panes, Tags,* right-click on a structure element in the hierarchy and select *Properties...* to display the object properties dialog. The *Tag* tab displays the *Actual Text* and *Alternate Text* attributes.

Abbreviation expansion (E). Abbreviations and acronyms can be assigned expansion text with the *E* suboption of the *tag* option. The expansion text should consist of a whole word or phrase which can be read by a screen reader. If an abbreviation doesn't have any expansion text the *E* attribute can be supplied nevertheless in order to assist the text-to-speech conversion process. For example, the term *IBM* may have expansion text *I B M* (with intermediate space characters) assigned to it.

In the following code fragment the expansion text *Mister* is assigned to the abbreviation *Mr.*:

```
p.fit_textline("Mr.", x, y, "tag={tagname=ASpan E={Mister} } ...");
```

11.3.6 Print Stream Order and Logical Reading Order

There are two fundamentally different concepts for content ordering in PDF. Figure 11.3 visualizes a sample page with two main text columns, interrupted by a table and an inserted summary on a gray background as well as header and footer.

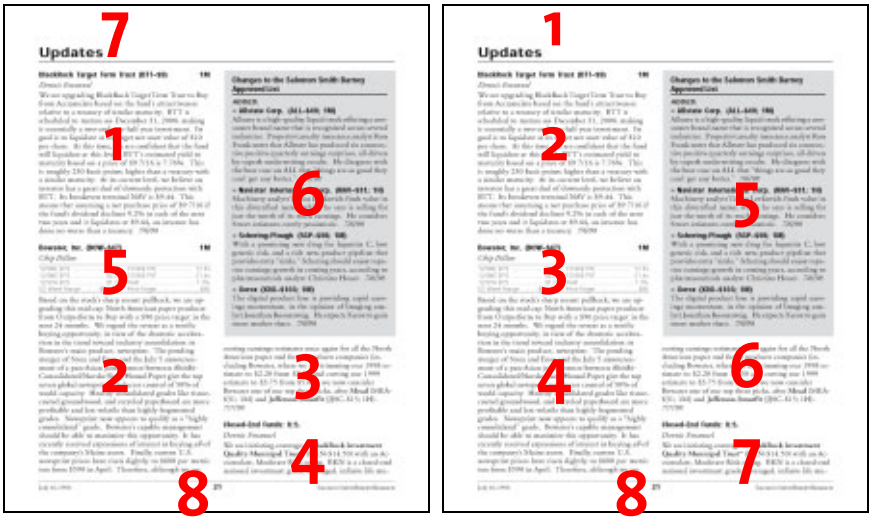


Fig. 11.3 Logical reading order (left) and print stream order (right).

The order of PDFlib API function calls determines the order of PDF text and drawing operators in the page content stream (called »raw print stream order« in Acrobat). Since page contents may be created in any way which suits the controlling application this is only a technical ordering which doesn't necessarily have any semantic significance. The print stream order is displayed in Acrobat's *Order* and *Contents* panes.

Logical reading order is the order in which a human reads the text. It determines the ordering used by a screen reader and Acrobat's *Read Aloud* feature. The logical reading order is determined by the logical structure tree. Tagged PDF requires that all content is tagged in semantically correct order, i.e. the structure hierarchy must include the page contents in the order in which they will be read by a human. Proper tagging ensures that screen readers present the contents in logical order. Since Artifacts are not part of the structure tree they are excluded from the logical reading order.

Reading order and print stream order in Acrobat. You can check the logical reading order in Acrobat X/XI/DC with the following methods:

- Select *View, Show/Hide, Navigation Panes, Tags* and check the order of elements from top to bottom. This ordering should accurately reflect the desired reading order.
- Activate *View, Read Out Loud...* to have Acrobat read the page contents in the reading order specified in the document.

The *Order* and *Content* panes list page contents in print stream order.

Creating content in logical reading order. The natural method which works in many situations is to sequentially generate all constituent parts of a structure element, and then move on to the next element in the logical sequence. In technical terms, the structure tree is created by a single depth-first traversal, and PDFlib functions to create page contents are called in the order in which the contents will be read.

PDFlib supports several methods for creating contents in any order while still creating the structure hierarchy in logical order. These methods are discussed in Section 11.4.4, »Creating Contents out of Order«, page 309.

11.3.7 Tagged PDF Problems in Adobe Acrobat

This section mentions observations that we made while testing Tagged PDF output in Adobe Acrobat. Table 11.5 lists bugs and inconsistent behavior in Acrobat DC, grouped according to the following features:

- 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 screen reader.
- Acrobat's Read Aloud Feature: Tagged PDF enhances Acrobat's capability to read text aloud.
- The »Find...« function in Acrobat's Tags pane can be used to search for Artifacts and unmarked content.

Table 11.5 Acrobat DC problems related to Tagged PDF

Description, recommendations and workarounds
Acrobat's accessibility checker
The `Alt` *attribute is ignored for* `Figure` *tags.*
The accessibility checker complains about form fields of type check box with a check mark symbol »Character encoding – failed« although Read aloud perfectly reads the field contents.
Acrobat's Read Aloud feature
When a tagged page has been placed with PDI and contains only Artifacts, Read Aloud nevertheless reads the contents of the placed page.
Other Acrobat functions
The `Find...` *function in the Tags pane incorrectly reports Artifacts as unmarked content, e.g. table decoration. In contrast, these items are correctly displayed as »Container <Artifact>« in the Content pane.*

11.4 Advanced Tagged PDF Topics

11.4.1 Automatic Table Tagging

PDF_fit_table( ) can automatically create suitable tags for the generated table based on the information supplied to the *PDF_add_table_cell( )* calls for the table contents. The *tag* option of *PDF_fit_table( )* with *tagname=Table* triggers automatic table tagging as detailed in Table 11.6.

Table 11.6 The tag *option in PDF_fit_table( ) and automatic table tagging*

tag option in PDF_fit_table( )	***result***
tagname=Table	*Activates automatic table tagging. Instead of* tagname=Table *a custom tag which is role-mapped to* Table *can also be used.*
tagname=Artifact	*The whole table contents including the caption are marked as Artifact; the* BBox *attribute is added automatically.*
any other tagname	*No* Table *structure will be created, but the cell contents are added as children of the element specified in the* tag *option. Pseudo and inline items are not allowed for PDF_fit_table( ).*
tag option not supplied	*No automatic table tagging. If the* tag *option is supplied to individual calls to PDF_add_table_cell( ) (as suboption to one of the* fit* *options) the corresponding structure elements for the cell contents will be created, but without any table structure. This may be useful if the table is used for layout purposes instead of as a data table.*

Note Automatic table tagging can only be leveraged if PDFlib's table engine is used. While it is possible to correctly tag tables which are created manually (i.e. without PDFlib's table engine), this process requires detailed knowledge of the table structure in the client application. In addition to the row/column structure relevant information about header cells and row/column spans is required. Empty cells must also be tagged.

Visualizing table tags in Acrobat. You can visualize the structure of table elements with Acrobat X/XI/DC as follows:

- Select *Tools, Accessibility, Reading Order* (Acrobat DC) or *TouchUp Reading Order (Acrobat X/XI)*. Table elements are highlighted and marked with a small number near the top left corner of the table. If a table summary is present, it is displayed next to the small number.
- Select the number or structure type name in the top left corner of the table and click *Table Editor* in the *Reading Order* dialog. The table structure is visualized with horizontal and vertical lines. In the *Table Editor Options* dialog you can instruct Acrobat to display *TH/TD* icons according to the type of table cell (see Figure 11.4).
- Right-click on a table cell and select *Table Cell Properties...* to check the cell type (Header cell *TH* vs. data cell *TD)*, the *scope* attribute, *rowspan* and *colspan* attributes and *header/ID* values.

Note the following restrictions when working with the Table Editor in Acrobat:

- The lines which visualize table cells may be displayed in wrong positions.
- Acrobat does not activate the Table Editor if the table uses a custom structure element which is role mapped to *Table* (as opposed to the standard element *Table)*.

Travel Expense Report				
	Meals	Hotels	Transport	subtotals
Madrid				
13-Aug-2012	37.74	112.00	45.00	
14-Aug-2012	27.28	112.00	45.00	
subtotals	65.02	224.02	90.00	379.02
Paris				
15-Aug-2012	96.25	109.00	36.00	
16-Aug-2012	35.00	109.00	36.00	
subtotals	131.25	218.00	72.00	421.25
Totals	196.27	442.00	162.00	**827.27**

Fig. 11.4
Acrobat's Table Editor for Tagged PDF tables displays header (TH) and data (TD) cells.

- The Table Editor does not display *Caption* elements which may be present in a table.
- If a table cell contains vertical text (e.g. a Textline with *orientate=east* or *west)* this cell and its immediate neighbor to the right are not displayed in the Table Editor although they are present in the logical structure tree and their contents are visible on the page.

Automatically created table tags and attributes. Automatic table tagging works only in page scope and operates as follows:

- A separate *Table* element is created for each table instance. For example, if a table is split in two or more instances, multiple *Table* elements are created. The *Summary* attribute is added to the *Table* element if the *Summary* suboption has been supplied to the *tag* option of *PDF_fit_table( )*.
- A *Caption* element is created if the *caption* option was specified in *PDF_fit_table( )*. As a grouping element *Caption* does not allow any content items. You must therefore supply the *tag* suboption of the *caption* option to specify a structure element as child of *Caption*. This element can hold the actual contents of the caption.
- A *TR* element is created for each table row. Rows which are specified in the *header* option of *PDF_fit_table( )* are wrapped by *THead*, rows which are specified in the *footer* option are wrapped by *TFoot*. All other rows are wrapped by *TBody* if headers or footers are present.
- Each table cell is wrapped in a *TH* (table header) or *TD* (table data) element according to the *tagname* suboption of the *tag* option of *PDF_add_table_cell( )*. If this option has not been supplied the cell type is selected as follows:
  - The *Scope* attribute for a cell enforces *TH* (even if *tagname=TD* is specified).
  - If another cell contains the *Headers* option with the *id* of the cell, the target cell is forced to *TH* (even if *tagname=TD* is specified).
  - If the cell is included in a table row which is part of the table header as specified by the *header* option of *PDF_fit_table( )* it is wrapped by *TH,* and *Scope=Column is* added.
- An empty *TD* dummy element is created for each table cell for which *PDF_add_table_cell( )* has not been called.
- *TH* and *TD* elements get appropriate *RowSpan* and *ColSpan* attributes according to the *rowspan* and *colspan* options of *PDF_add_table_cell( )*. The *RowSpan* and *ColSpan* suboptions of the *tag* option can not be used.

- *Table*, *TH* and *TD* elements are assigned appropriate *Width* and *Height* attributes; *Table* elements are also assigned the *BBox* attribute.
- Other table cell attributes may be supplied as suboptions for the *tag* option of *PDF_add_table_cell( )*. The following options are not allowed: *RowSpan*, *ColSpan*, *Height*, *index*, *parent*, *Width*.
- Table rows and cells are emitted in zigzag order starting from the top left cell (i.e. column 1, row 1) to the bottom right cell, regardless of the order of calls to *PDF_add_table_cell( )*.
- Decorative table elements are automatically tagged as *Artifact* with *artifacttype=Layout*, i.e. ruling and shading (fill/stroke) of table cells, rows, columns or the full table, matchbox filling and ruling, *showborder* rules, and visualization aids controlled via *debugshow*, *showcells*, and *showgrid*.

Cookbook *Code samples for automatic table tagging can be found in the* `tagged_table` *and* `invoice_pdfua1` *topics in the* `pdfua` *category of the PDFlib Cookbook.*

Adding tags and attributes to table cells. Abbreviated tagging can be applied to the table caption, a table cell, or the contents of a table cell. Supplying the *tag* option to *PDF_add_table_cell( ) is* useful in the following situations:

- Table cells can be forced to be header cells (instead of *TD* data cells) with *tagname=TH* if they are not included in header rows or don't have any *Scope* attribute.
- Create the proper tag structure required for links; see »Tagging links in a table cell«, page 305, for details.

Some restrictions apply to the suboptions of the *tag* option in *PDF_add_table_cell( ):*

- The following options can not be used: *ColSpan, Height, index, parent, RowSpan, Width*.
- The *tagname* option can only have the value *TH* or *TD* to specify the type of table cell. However, descendant tags can be specified by nesting the *tag* option.
- Since the *id* option must be unique within a document, it is not allowed for table cells which are repeated in multiple table instances, e.g. cells in a header or footer row for tables which create more than one table instance.

Adding tags and attributes to table cell contents. You can also supply *tag* as suboption to the following options (or corresponding suboptions of the *caption* option) of *PDF_add_table_cell( ):*

```
fitannotation, fitfield, fitgraphics, fitimage, fitpath,
fitpdipage, fittextflow, fittextline
```

This is useful in the following situations:

- Specify sub-structure for the contents of a table cell. The *tag* option creates a child element of the cell's *TH* or *TD* element. The following values for *tagname* are not allowed for the *tag* option if automatic table tagging is active (in other words, nested tables are not supported):

  ```
  Table, TR, TH, TD, THead, TBody, TFoot
  ```

- The table attributes listed below cannot be created automatically, but may be required for accessibility purposes. They must be supplied by the user:
  - The *Headers* option must be supplied for *TD* cells which reference one or more *TH* cells inside the table (i.e. they reference header cells which are not part of the *header* option of *PDF_fit_table( )*).

- The *Id* and *Scope=Row* options must be supplied for *TH* cells which are referenced in any *Headers* option *(Scope=Column* is created automatically for *TH* column header cells).

- The caption may contain arbitrary contents which can itself be tagged. For example, the following option list creates a *Caption* element containing a single Textline within a nested *P* element:

```
caption={ fittextline={tag={tagname=P title={Travel Expense Report}} ... } ... }
```

11.4.2 Tagging Interactive Elements

Links, annotations and form fields must also be made accessible. The corresponding interactive elements must be represented in the structure tree at the proper location in the structure tree. It is not possible to create interactive elements as Artifacts.

Cookbook *Code samples for creating tagged links can be found in the* `starter_pdfua1` *sample. The* `image_with_link_pdfua1` *topic creates a link with a background image. The* `table_of_contents_pdfua1` *topic creates a table of contents with TOC and TOCI structure elements and actionable links.*

Links and other annotation types. Annotations require the following items for accessibility (see Figure 11.5):

- A *Link* element (for link annotations) or *Annot* element (for all other annotation types) serves as container for the next two items. *Alt* or *ActualText* options may be provided to supply an alternate description or replacement text. The *Alt* attribute of a *Link* element should describe the purpose of the link. If the link target is located in the current document *(GoTo* action) the *Link* element should additionally be contained in a *Reference* element (with option *direct=false)*.
- Text which represents the interactive element should be created inside the container element. If no text is required this element can be skipped, e.g. for a Text annotation. If the annotation is represented by a raster image or vector graphics this should be tagged as Artifact. In this case the alternative text of the link describes both the graphic and the link. It is recommended to use the *matchbox* option of various content creation functions to prepare geometry information for the next element. The currently active element when the *matchbox* option is supplied must be a structure item, i.e. the option *structureitem=false* is not allowed.
- The annotations must be created with *PDF_create_annotation( )*. In addition to the annotation this function creates a corresponding structure element of type *OBJR* (object reference) for the annotation. The *contents* option of *PDF_create_annotation( )* should be provided for link annotations (this is required in PDF/UA-1). Other annotation types should be created with the *contents* option of *PDF_create_annotation( )* or the *ActualText* tagging option. The *usematchbox* option can be used to conveniently supply the geometry of the visual content created in the second step.

The second and third items may be created in either order. The annotation requirements above also apply to annotations created in table cells with the option *fitannotation* of *PDF_add_table_cell( )*.

The following code fragment creates an interactive link with the required three items (the resulting tag structure is shown in Figure 11.5):

```
/* Create the parent Link element */
id_link = p.begin_item("Link", "Title={Kraxi on the Web} Alt={Kraxi on the Web}");
```

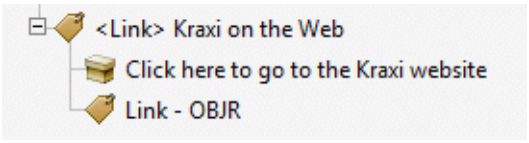


Fig. 11.5
Representation of an accessible link in the structure tree

```
/* Create visible content which represents the link */
p.fit_textline("Click here to go to the Kraxi website", x, y,
        "matchbox={name={kraxi}} fontsize=14 font=" + font);

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

/* Create Link annotation on named matchbox "kraxi". */
p.create_annotation(0, 0, 0, 0, "Link",
        "action={activate=" + action + "} "
        "usematchbox={kraxi} contents={Link to Kraxi Inc. Web site}");

p.end_item(id_link);
```

Note *The required tagging sequence for interactive elements cannot be achieved with Textflow and matchboxes because tags cannot be created inside a Textflow, and creating the tags for interactive elements after placing the Textflow would spoil reading order.*

Tagging links in a table cell. Links in a table cell require the tag structure described above. However, this can be a bit tricky because the structure elements *TH/TD, Link, OBJR* and possibly content must be properly nested. In order to achieve this you must supply the *tag* option to *PDF_add_table_cell()* and make use of its nesting feature. The option list below fills a table cell with a line of text and a link annotation. The enclosing *TD* element is supplied in the outer *tag* option (since *TD* is created automatically by the table engine the outer *tagname* suboption can be omitted), and the *Link* element is supplied in the inner *tag* option. Finally, the required *OBJR* element is created automatically by the *fitannotation* option which serves as equivalent to *PDF_create_annotation()*:

```
fittextline={font=... fontsize=25 fillcolor=blue}
annotationtype=Link fitannotation={contents={Kraxi home page} action={activate ...}}
tag={tagname=TD tag={tagname=Link}}
```

Tagging form fields. Form fields require the following structure elements for accessibility:

- A group of related form fields may be enclosed with a *Part* structure element.
- A *Div* structure element is recommended which contains the following constituents of an individual form field.
- A *Caption* structure element with a *P* structure element which encloses text on the page which describes the purpose of the field.
- For checkboxes and radio buttons it is recommended to create another *Div* element which encloses the check boxes/radio buttons and their corresponding captions. Since the *Caption* structure element logically usually precedes the field it comes first in th structure hierarchy, even if they are placed in reverse order on the page.
- A *Form* structure element serves as container for the next element. *Alt* or *ActualText* options may be provided to supply an alternate description or replacement text.

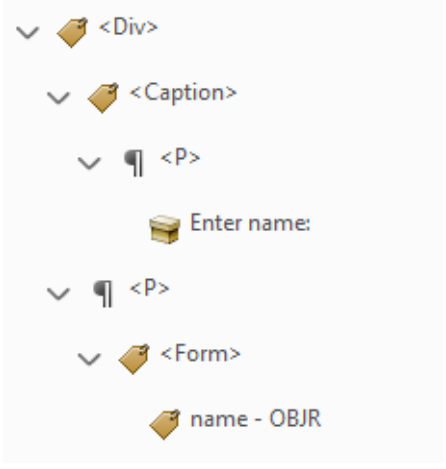


Fig. 11.6
Representation of a form field in the structure tree

Keep in mind that pseudo elements, table elements, ILSEs, Ruby and Warichu elements are not allowed as parents for *Form*. When creating radio buttons, no *Form* element is required in *PDF_create_fieldgroup( )* for the radio button group, but only in *PDF_create_field( )* for creating the individual radio buttons.

- A structure element of type *OBJR* (object reference) for the form field, nested inside the *Form* structure element, is created automatically by *PDF_create_field( )*. The *tooltip* option of *PDF_create_field( )* should be provided to enhance accessibility of the field (this is required in PDF/UA-1).

The following code fragment creates the recommended structure for a text field as shown in Figure 11.6. The nested *Caption/P* structure elements are created with nested abbreviated tagging using the *tag* option in *PDF_fit_textline( )*. Abbreviated tagging in *PDF_create_field( )* is used to create the *Form* structure element:

```
id_Div = p.begin_item("Div", "");

labeltext = "Enter name:";
p.fit_textline(labeltext, x1, y1, "tag={tagname=Caption tag={tagname=P}}");

optlist = "tag={tagname=P tag={tagname=Form}} tooltip={" + labeltext + "} " +
"bordercolor={gray 0} font=" + font;

p.create_field(x2, y2, x3, y3, "name", "textfield", optlist);

p.end_item(id_Div);
```

The form field requirements above also apply to form fields created in table cells with the option *fitfield* of *PDF_add_table_cell( )*.

Cookbook *A code sample for creating all types of Tagged form fields can be found in the topic* `form_fields` *in the* `pdfua` *category of the PDFlib Cookbook.*

Structured bookmarks. Bookmarks can be assigned a structure element in addition to the usual destination. Such bookmarks are called *structured bookmarks* and Acrobat offers additional features for them. Upon right-clicking a structured bookmark in Acrobat

the functions *Delete Page(s)* and *Extract Page(s)* are available which operate on the page or pages which contain the structure element. Structured bookmarks create a connection between a bookmark and a structure element. This connection can be created in two different ways:

- Create a bookmark with *PDF_create_bookmark()* and supply its handle to the *bookmark* option in *PDF_begin_item()* or the *tag* option of various functions:

  ```
  bm = p.create_bookmark("Section 1", "");
  id = p.begin_item("H1", "Title={Section 1} bookmark=" + bm);
  p.fit_textline(text, x, y, "");
  p.end_item(id);
  ```

  This method can also be used with abbreviated tagging:

  ```
  bm = p.create_bookmark("Section 1", "");
  p.fit_textline(text, x, y,
       "tag={tagname=H1 Title={Section 1} bookmark=" + bm + "}");
  ```

  The disadvantage of this method is that the bookmark text must be available before the structure element and its contents are created. This may be inconvenient if the referenced structure element is located higher up in the structure tree.
- Create a structure item with *PDF_begin_item()* and supply its handle to the *item* option of *PDF_create_bookmark()*. Instead of an item handle the keyword *current* can be supplied as a shortcut which refers to the current structure element at the time *PDF_create_bookmark()* is called:

  ```
  id = p.begin_item("H1", "Title={Section 1} ");
  bm = p.create_bookmark("Section 1", "item=current");
  p.fit_textline(text, x, y, "");
  p.end_item(id);
  ```

  This method has the advantage that the bookmark text must only be available when the structure element and its contents are created. However, it cannot be used with abbreviated tagging.

Structured bookmarks can only refer to open structure elements, and can not refer to pseudo or inline items. The client code must ensure that the bookmark's destination matches the structure element (otherwise activating the bookmark in Acrobat would not jump to the element, but to a different location in the document). If the associated structure element spans more than one page the bookmark should point to the first page in this range.

11.4.3 Lists

Lists are used to group related items. They are represented by the following structure elements (see Figure 11.7):

- An *L* element which contains all of the following structure elements. The *ListNumbering* option can be used to specify the numbering system used in the *Lbl* elements. The *ListNumbering* option may be useful for screen readers even without *Lbl* elements.
- An optional *Caption* element. Since *Caption* is a grouping element it cannot contain content items, but only other structure elements (e.g. *P*).
- One or more list items *(LI)* containing the following:

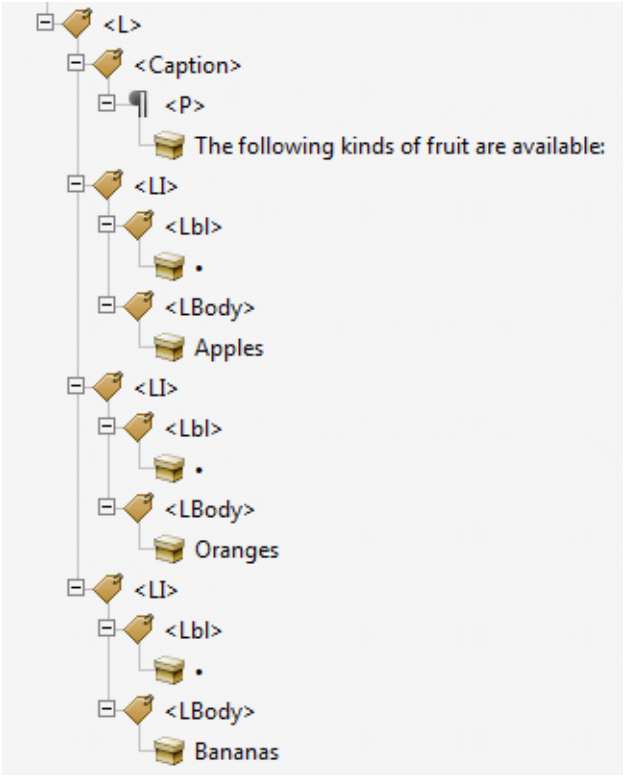


Fig. 11.7
Representation of an accessible list in the structure tree

- An optional label *(Lbl)* with a bullet, number, etc.
- An *LBody* element with the actual contents of the list item. *LBody* may contain either content items or other structure elements including a nested list.

The following code fragment creates a list with a caption and three items. Each list item is preceded by a bullet character U+2022 which is tagged as label (the resulting tag structure is shown in Figure 11.7):

```
id_list = p.begin_item("L", "ListNumbering=Disc");

/* Create both Caption and P elements at once */
p.fit_textline("The following kinds of fruit are available:",
        x1, y, "tag={tagname=Caption tag={tagname=P}}");
    y -= leading;

id_listitem = p.begin_item("LI", "");
    p.fit_textline("&#x2022;", x1, y, "tag={tagname=Lbl}");
    p.fit_textline("Apples", x2, y, "tag={tagname=LBody}"); y -= leading;
p.end_item(id_listitem);

id_listitem = p.begin_item("LI", "");
    p.fit_textline("&#x2022;", x1, y, "tag={tagname=Lbl}");
    p.fit_textline("Oranges", x2, y, "tag={tagname=LBody}"); y -= leading;
p.end_item(id_listitem);

id_listitem = p.begin_item("LI", "");
```

```
        p.fit_textline("&#x2022;", x1, y, "tag={tagname=Lbl}");
        p.fit_textline("Bananas", x2, y, "tag={tagname=LBody}"); y -= leading;
    p.end_item(id_listitem);

    p.end_item(id_list);
```

Cookbook *A code sample for creating a Tagged PDF list can be found in the topics* `list_pdfua1` *in the* `pdfua` *category of the PDFlib Cookbook.*

11.4.4 Creating Contents out of Order

As mentioned in Section 11.3.6, »Print Stream Order and Logical Reading Order«, page 298, it is crucial to create the elements in the structure hierarchy in logical reading order. If the application processes page contents in an order which is different from logical reading order (e.g. always top to bottom regardless of column relationships), several PDFlib features can be used to maintain proper ordering in the structure hierarchy:

- Create child elements out of order with the *index* option. It modifies the position where a new structure element is inserted within its parent element.
- Create structure elements out of order with the *parent* option. It modifies the parent element where a new structure element is inserted.
- Jump back and forth in the structure hierarchy with the *PDF_activate_item()* function. It can be used to add more structure elements or content items to some element in the structure hierarchy.

These methods are discussed in more detail below.

Cookbook *Code samples for tagging elements out of order can be found in the topics* `out_of_order` and `parallel_columns_pdfua1` *in the* `pdfua` *category of the PDFlib Cookbook.*

Creating child elements out of order. In order to create child elements within a structure element out of order you can specify a location in the structure tree with the *index* option of *PDF_begin_item()* or suboptions of the *tag* option of various functions. The following code fragment emits text fragments in reverse order, and corrects the order in the structure tree by inserting each new text fragment as the new first child *(index=0)* of the parent element. Since each new element is inserting as the new first child element of the parent, as a result the logical ordering will be the reverse of the creation order:

```
p.fit_textline("three", x, y, "tag={tagname=P index=0}");
y += leading;
p.fit_textline("two",   x, y, "tag={tagname=P index=0}");
y += leading;
p.fit_textline("one",   x, y, "tag={tagname=P index=0}");
```

You can query the index of the currently active tag within its parent element with *PDF_get_option()* and the *activeitemindex* or the *activeitemkidcount* keyword, and later return to this position in the structure tree. The following code fragment inserts a new element after the element at the stored index:

```
nextindex = p.get_option("activeitemindex", "") + 1;

...create more elements on the same level...

p.fit_textline(text, x, y, "tag={tagname=P index=" + nextindex + "}");
```

Creating structure elements out of order. In order to create child elements at some other location within the structure tree instead of at the current position, use the *parent* option. It must refer to a structure element which has not yet been closed. Since elements created with abbreviated tagging are created and closed in the same function call they cannot be used as target of the *parent* option. You can query the id of the currently active tag with *PDF_get_option( )* and the *activeitemid* keyword, and later return to this position in the structure tree:

```
parent_id = p.get_option("activeitemid", "");
...
p.fit_textline(text, x, y, "tag={tagname=P parent=" + parent_id + "}");
```

For even more flexibility the *parent* and *index* options can be used in combination. Use *PDF_suspend/resume_page( )* to interrupt a page, continue on another page, and then go back to add more content to the suspended page.

Activating items for complex layouts. In order to facilitate the creation of structure information for complex non-linear page layouts PDFlib offers a feature called item activation. It can be called 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 benefit from this technique are the following:

- multiple columns on a page
- inserts which interrupt the main text, such as summaries or other non-linear text items
- tables and illustrations which are placed between columns.

The function *PDF_activate_item( )* allows you to switch back and forth between different branches of the structure tree. The »logical order« approach requires the client application to construct the page contents in logical order even if it would be easier to create it in visual order. In contrast, with item activation the contents can be created in visual order (or any other ordering which is convenient for the application). This technique can also be applied if the content spans multiple pages.

In order to work around problems in Acrobat, no content items should be added immediately after calling *PDF_activate_item( ),* but only other structure elements.

Querying the currently active structure element. In order to use the *parent* and *index* options or *PDF_activate_item( )* some knowledge about the currently active structure element and its children is required. This status information can be maintained by the application, but can also be queried from PDFlib. The function *PDF_get_option( )* with the keywords *activeitemid, activeitemindex, activeitemkidcount, activeitemname, activeitemstandardname* delivers the id, index, number of child elements, name, and standard name (if it is rolemapped) of the current element.

11.4.5 Importing Tagged PDF Pages with PDI

Cookbook *Code samples for importing pages from Tagged PDF documents can be found in the topics* `clone_pdfua` *and* `merge_and_stamp_pdfua1` *in the* `pdfua` *category of the PDFlib Cookbook.*

In Tagged PDF mode pages from a Tagged PDF document are imported along with their structure element tags. We abbreviate a page from a Tagged PDF document imported in

Tagged PDF mode with *usetags=true* as »tagged page«. This status can be queried with the *tagged* keyword of *PDF_info_pdi_page()*. Importing tagged pages works as described below.

Opening a Tagged PDF document. *PDF_open_pdi_document()* checks whether the imported document is compatible to the current PDF/A-1a/2a/3a or PDF/UA mode and reads the structure tree of the imported document.

If the *usetags* option is *false* the document's structure information is ignored, and no tags can be imported from the document. Note that the default of *usetags* is *false* if *PDF_open_pdi_document()* is called in object scope, i.e. before *PDF_begin_document()*.

Note Attribute classes and class maps are not imported.

Cloning input document language. If most or all pages of a PDF document are imported with PDI it is recommended to clone the document language entry (see »Language specification«, page 296) if it is present in the input document. The document language can be cloned with pCOS and the following code fragment:

```
if (p.pcos_get_string(indoc, "type:/Root/Lang").equals("string"))
{
        inputlang = p.pcos_get_string(indoc, "/Root/Lang");
        optlist += " lang=" + inputlang;
}

p.begin_document(filename, optlist);
```

Opening a tagged page. *PDF_open_pdi_page()* selects the structure elements comprising the imported page's contents and filters the tags present on the page. For example, tags for annotations are removed since PDI does not import interactive elements. Finally, one or more structure elements on the imported page are selected which form the top of the imported structure sub-tree. If the *usetags* option is *false* the page's structure information is completely ignored.

Entries in the imported document's role map are copied to the output document's role map if the corresponding element is used on the page. Conflicting role map entries (i.e. a custom tag is already mapped to a different standard tag in the generated document's role map or a previously imported document) are ignored. However, pages with conflicting role map entries are rejected in PDF/UA mode, i.e. the call to *PDF_open_pdi_page()* fails.

Importing documents with invalid tag structure. PDFlib implements strict checks for the tag nesting rules imposed by ISO 32000-1 as detailed in »Nesting rules for structure elements«, page 291. These checks can also be applied to imported documents and the tag structure created from imported pages. The nesting rules in imported pages are not checked by default. However, these checks can be enabled with the *checktags* option of *PDF_open_pdi_document()*. If *checktags=strict* all tag nesting rules are enforced in *PDF_open_pdi_page()*. If the structure hierarchy of the imported page violates the nesting rules for structure elements the call to *PDF_open_pdi_page()* fails, and *PDF_get_errmsg()* reports an error similar to the following:

```
Grouping element type 'Document' cannot contain content items
(but only other structure elements)
```

Since many existing real-world Tagged PDF documents violate the tag nesting rules you can address these problems in imported documents with one of the following methods:

- Inserting an additional tag on top of the imported structure hierarchy (e.g. with the *tags* option of *PDF_fit_pdi_page( )*) is useful to fix common problems where the imported page contains content items immediately under the document root.
- Other problems cannot be solved by inserting additional tags, e.g. when the structure elements for tables or lists are incomplete. You should consider fixing the input documents if possible. If this is not a viable solution you can set *checktags=none* in *PDF_open_pdi_document( )* to import tagged PDF pages with a nonconforming tag structure.
- If the imported structure itself is correct, but conflicts with the generated new tags of the output document you should try to adjust the new tags appropriately. If this is not feasible you can set *checktags=none* in *PDF_begin_document( )* to ignore conflicts in the generated tagging structure. This is not allowed in PDF/UA-1 mode.

Since nonconforming input processed with either variant of *checktags=none* may result in nonconforming PDF output this setting is not recommended.

Querying and checking the tags in an imported page. In some situations it can be difficult to properly integrate imported structure elements in the generated new structure hierarchy. In order to assist in this process, several properties of an imported tagged PDF page can be queried with *PDF_info_pdi_page( )* after a page has been successfully opened:

- The keyword *fittingpossible* reports whether the page can be placed in the current context. If the *tag* option is supplied you can check whether the page can be placed with an additional top-level tag. Only the *tagname* suboption of the *tag* option is evaluated; other suboptions should not be supplied. Using this keyword is recommended in situations where you are not sure about the structure information in imported pages and want to avoid an exception in *PDF_fit_pdi_page( )* because of unsuitable imported structure elements. If a page is rejected by the *fittingpossible* test you can try to insert an additional tag via the *tag* option.
- The keyword *topleveltagcount* reports the number of top-level structure elements since there may be more than one top-level tags. Note that *topleveltagcount* may be 0 in rare cases where the page contents are not covered by any structure element. Such a page behaves like an untagged page. Like other content items it cannot be placed as child of a grouping element, but requires an additional tag on top of the page.
- The keyword *topleveltag* reports the top-level structure element(s) of the imported page. This may be useful to determine whether or not additional structure elements can or must be inserted above the imported page structure.
- The keyword *lang* reports the *lang* attribute of all top-level imported structure element(s). This helps to decide whether a *lang* attribute is required in higher structure elements.

Placing a tagged page. *PDF_fit_pdi_page( )* places the imported page on a new page and integrates its structure hierarchy in the generated document's structure tree, using the currently active element as parent for the imported structure tree. An additional structure element can be created with the tag option of *PDF_fit_pdi_page( )*. It serves as new parent for the imported structure hierarchy.

If *Alt* or *ActualText* attributes are present in the imported hierarchy they are removed if a conflict with existing attributes higher up in the generated document structure is found (see »Nesting rules for alternate and replacement text«, page 298).

If the page has been opened with *usetags=true* in *PDF_open_pdi_document( )* and *PDF_open_pdi_page( )* it can be placed at most once in the output document since the imported structure must be integrated at a unique location of the output document's structure hierarchy. If a page contributes content and structure at more than one location it can be opened multiply, however (i.e. different page handles are placed).

The following strategy is recommended for importing pages with unknown document structure:

- If the *topleveltagcount* keyword of *PDF_info_pdi_page( )* reports a tag count of 0, the page is either empty or contains only Artifacts. Some applications may decide to skip such pages since they don't contribute any relevant content. If the page is nevertheless imported, it is recommended to a supply an additional *Artifact* tag to work around an Acrobat problem.
- If the *fittingpossible* keyword of *PDF_info_pdi_page( )* returns 1 for this page it can safely be placed with *PDF_fit_pdi_page( )*. This test is not required if the page is placed as Artifact.
- Otherwise, an additional tag can be inserted on top of the imported page's structure. The choice of this tag depends on application, especially the type of structure element where the imported page is inserted into the structure tree.
- If the *fittingpossible* keyword of *PDF_info_pdi_page( )* still rejects the page with the additional tag, the application could try another tag or give up the page.

The following code fragment implements the strategy outlined above. It inserts an additional *P* element if the page cannot be placed directly:

```
fittingpossible = true;
additionaltag = "";

topleveltagcount = (int) p.info_pdi_page(page, "topleveltagcount", "");

if (topleveltagcount == 0)
{
        /* The page doesn't contain any structure elements,
         * i.e. it is empty or contains only Artifacts.
         * We add an "Artifact" tag to work around an Acrobat bug.
         */
        additionaltag = "tag={tagname=Artifact} ";
}
else
/*
* Try to place the page without any additional tag;
* if this doesn't work we insert another tag.
*/
if (p.info_pdi_page(page, "fittingpossible", "") == 0)
{
        additionaltag = "tag={tagname=P} ";

        if (p.info_pdi_page(page, "fittingpossible", additionaltag) == 0)
        {
            fittingpossible = false;
        }
}
```

```
if (fittingpossible)
{
        p.fit_pdi_page(page, 0, 0, "adjustpage " + additionaltag);
}
else
{
        System.err.println("Skipping page: " + p.get_errmsg());
}
```

Use cases. The use cases below for importing pages in Tagged PDF mode can be distinguished; Table 11.7 summarizes options and other requirements for these use cases:

- Honor tags: import a page from a Tagged PDF document and copy its tags to the output. The structure elements comprising the page become part of the new structure hierarchy. Optionally an additional tag can be placed on top of the imported hierarchy.
- Drop tags: import a page from a Tagged or untagged PDF document and tag the whole page with a new tag. Existing structure elements are dropped and the contents of the imported page comprise a single tag without any internal structure.
- Place as Artifact: import a page from a Tagged or untagged PDF document and mark the whole page as Artifact, e.g. for a background graphic. Existing structure elements are dropped.

Table 11.7 Importing tagged and untagged PDF pages in Tagged PDF mode

use case	*imported page*	*options*	*requirements and comments*
honor tags	*tagged*	`usetags=true` *in* PDF_open_pdi_document( ) *and* PDF_open_pdi_page( )	▸ *parent*[1] *must not be inline or pseudo element* ▸ *imported page can be placed only once* ▸ *top-level element(s) of imported page must meet nesting rules*
drop tags	*tagged or untagged*[2]	`usetags=false` *in* PDF_open_pdi_document( ) *or* PDF_open_pdi_page( )	▸ *contents of imported page become part of currently active item* ▸ *only non-grouping parent elements allowed*
place as Artifact	*tagged or untagged*	`tag={tagname=Artifact}` *in* PDF_fit_pdi_page( )	▸ *contents of imported page become Artifact*

1. Currently active tag specified with PDF_begin_item( ) *or the* `tag` *option of* PDF_fit_pdi_page( )
2. This situation may also arise for Tagged PDF pages where no structure element actually refers to any page content.

12 PDF Versions and Standards

12.1 Acrobat and PDF Versions

At the user's option PDFlib generates output according to the following PDF versions:

- PDF 1.4 (Acrobat 5, released 2001)
- PDF 1.5 (Acrobat 6, released 2003)
- PDF 1.6 (Acrobat 7, released 2005)
- PDF 1.7 (Acrobat 8, released 2006), technically identical to ISO 32000-1:2008
- PDF 1.7 Adobe extension level 3 (Acrobat 9, released 2008)
- PDF 1.7 Adobe extension level 8 (Acrobat X/XI/DC, released 2010/2012/2015-2019)
- PDF 2.0 according to ISO 32000-2:2017 including the dated revision ISO 32000-2:2020

The PDF output version can be controlled with the *compatibility* option in *PDF_begin_document( )*. In each PDF compatibility mode the PDFlib features for higher levels are not available (see Table 12.1). Trying to use such features will result in an exception.

PDF version of documents imported with PDI. In all compatibility modes only PDF documents with a lower or the same PDF version can be imported with PDI. If you must import a PDF with a newer PDF version you must set the *compatibility* option accordingly (see Section 8.3.3, »Document and Page-related Checks«, page 210). As an exception to the *cannot-import-higher-PDF-version* rule, documents according to PDF 1.7 extension level 3 (Acrobat 9) and PDF 1.7 extension level 8 (Acrobat X/XI/DC) can also be imported into PDF 1.7 documents.

Changing the PDF version of a document. If you must create output according to a particular PDF version, but need to import PDFs which use a higher PDF version you must convert the documents to the desired lower PDF version before you can import them with PDI. You can use the menu item *File, Save As Other..., Optimized PDF...* (Acrobat XI/DC) or *File, Save As..., Optimized PDF...* (Acrobat X) to change the PDF version.

Table 12.1 PDFlib features which require a specific PDF compatibility mode

Feature	*PDFlib API functions and options*
Features which require PDF 2.0 = ISO 32000-2 or a specific PDF/A or PDF/VT standard	
Document part hierarchy	*PDF_begin/end_dpart( ): A document part hierarchy requires PDF 2.0 or PDF/VT.*
Relationship of file attachments	*Option* `relationship` *for PDF_load_asset( ) with* `type=Attachment`*, for PDF_add_portfolio_file( ), and for use as suboption for the* `attachments` *option of PDF_begin/end_document( ) and the* `attachment` *option of PDF_create_annotation( ): relationship specifications require PDF 2.0 or PDF/A-3.*
Associated files	*Option* `associatedfiles` *for PDF_end_document( ), PDF_begin/end_page_ext( ), PDF_begin/end_dpart( ), PDF_begin_template_ext( ), PDF_load_image( ), PDF_open_pdi_page( ), PDF_load_graphics( ): Associated files require PDF 2.0 or PDF/A-3.*

Table 12.1 PDFlib features which require a specific PDF compatibility mode

Feature	PDFlib API functions and options
Features which require PDF 1.7 extension level 8 (Acrobat X/XI/DC) or above	
AES encryption with 256-bit keys	PDF_begin_document( ): AES encryption with 256-bit and a stronger encryption algorithm than in extension level 3 is automatically used with `compatibility=1.7ext8` if the `masterpassword`, `userpassword`, `attachmentpassword`, or `permissions` option is supplied
Features which require PDF 1.7 extension level 3 (Acrobat 9) or above	
Multimedia	PDF_load_asset( ) PDF_create_annotation( ): option `type=RichMedia` PDF_create_action( ): option `type=RichMediaExecute`
Geospatial PDF	PDF_begin_document( ): option `viewports` PDF_load_image( ): option `georeference`
PDF portfolios with folders	PDF_add_portfolio_folder( )
AES encryption with 256-bit keys	PDF_begin_document( ): AES encryption with 128-bit according to PDF 1.7 is automatically used with `compatibility=1.7ext3` if the `masterpassword`, `userpassword`, `attachmentpassword`, or `permissions` option is supplied to avoid a weakness in the AES-256 encryption algorithm according to PDF 1.7ext3
embed 3D models in PRC format	PDF_load_3ddata( ): option `type=PRC`
barcode fields	PDF_create_field( ) and PDF_create_fieldgroup( ): option `barcode`
Features which require PDF 1.7 = ISO 32000-1 (Acrobat 8) or above	
PDF portfolios	PDF_begin_document( ): option `portfolio` PDF_add_portfolio_file( )
Unicode file names for attachments	PDF_begin/end_document( ): option `attachments`, suboption `filename`
Features which require PDF 1.6 (Acrobat 7) or above	
NChannel color	PDF_create_devicen( ): option `subtype=nchannel`
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 with 128-bit keys	PDF_begin_document( ): AES encryption is automatically used with `compatibility=1.6` or `1.7` when the `masterpassword`, `userpassword`, `attachmentpassword`, or `permissions` option is supplied
encrypt file attachments only	PDF_begin/end_document( ): option `attachmentpassword`
attachment description	PDF_begin/end_document( ): suboption `description` for option `attachments`
embed 3D models in U3D format	PDF_load_3ddata( ), PDF_create_3dview( ) PDF_create_annotation( ): option `type=3D` PDF_create_action( ): option `type=GoTo3DView`
Features which require PDF 1.5 (Acrobat 6) or above	
DeviceN color space with more than 8 colorants	PDF_create_devicen( ): option `names`
various field options	PDF_create_field( ) and PDF_create_fieldgroup( )

Table 12.1 PDFlib features which require a specific PDF compatibility mode

Feature	***PDFlib API functions and options***
page layout	*PDF_begin/end_document( ): option* `pagelayout=twopageleft/right`
various annotation options	*PDF_create_annotation( )*
extended permission settings	`permissions=plainmetadata` *in PDF_begin_document( ), see Table 3.4*
Tagged PDF	*various 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( )*
compressed object streams	*compressed object streams will automatically be generated with* `compatibility=1.5` *or above unless* `objectstreams=none` *has been set in PDF_begin_document( )*

12.2 The PDF Standard ISO 32 000

ISO 32000-1. PDF 1.7 has been standardized as ISO 32000-1. The technical contents of this international standard are identical to Adobe's PDF 1.7 reference, the file format of Acrobat 8. PDF documents created with PDFlib conform to ISO 32000-1. A copy of this standard is freely available at the following location:

`http://www.adobe.com/devnet/pdf/pdf_reference.html`

ISO 32000-2. ISO 32000-2:2017 specifies PDF 2.0 and incorporates features from the following groups:

- Acrobat 9 features which are supported in PDFlib with the *compatibility=pdf1.7ext3* document option, e.g. georeferenced PDF, hierarchical Portfolios, and AES-256 encryption; see Table 12.1 for a detailed list.
- Acrobat X features which are supported in PDFlib with *compatibility=pdf1.7ext8*, particularly AES-256 encryption with a stronger encryption algorithm.
- Features introduced with the PDF/A-3 and PDF/VT standards such as associated files and Document Part Hierarchy.
- Other features which are not supported in Acrobat DC and below, e.g. new structure element types.

More details about PDF 2.0 can be found on the PDFlib Web site.

Deprecated PDF 2.0 features. PDF 2.0 deprecates the features listed in Table 12.2 which were available in earlier PDF versions. If any of these features is used, PDFlib emits a warning. We recommend to avoid these features even in PDF 1.x mode.

Table 12.2 PDFlib features which are deprecated in PDF 2.0

Feature	***PDFlib API methods and options***
accessibility restrictions	*PDF_begin_document( ): option* `permissions`, *keyword* `noaccessible`
viewer preferences for viewing and printing	*PDF_begin/end_document( ): option* `viewerpreferences`, *suboptions* `printarea`, `printclip`, `viewarea`, `viewclip`
separation dictionaries	*PDF_begin_page_ext( ) and PDF_end_page_ext( ): option* `separationinfo`
platform-specific action parameters	*PDF_create_action( ): options* `defaultdir`, `parameters` *and* `operation` *for* `type=Launch`
Movie actions	*PDF_create_action( ) with* `type=Movie` *(use* `type=RichMediaExecute` *instead)*
Movie annotations	*PDF_create_annotation( ) with* `type=Movie` *(use* `type=RichMedia` *instead)*
Flash-based multimedia	*PDF_load_asset( ) with* `type=Flash`
Flash navigators for portfolios	*PDF_end_document( ), option* `portfolio`, *suboptions* `initialview=custom` *and* `navigator`
blendmode arrays	*PDF_create_gstate( ): option* `blendmode` *accepts only a single keyword, but not a list with multiple values*
barcode fields	*PDF_create_field( ) and PDF_create_fieldgroup( ):* `barcode`

12.3 PDF/A for Archiving

12.3.1 The PDF/A Standards

Note *General information about the PDF/A standard can be found on the PDFlib Web site.*

The PDF/A formats specified in the ISO 19005 standard series 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 in the PDF Association. PDFlib GmbH is a founding member of the PDF Association which hosts the PDF/A Technical Working Group (TWG) as one of its activities. The aim of this industry organization is »to promote Open Standards-based electronic document implementations using PDF technology through education, expertise and shared experience for stakeholders worldwide«. For more information visit the PDF Association's Web site at *www.pdfa.org*.

PDF/A-1a:2005 and PDF/A-1b:2005 according to ISO 19005-1. PDF/A-1 is based on PDF 1.4 and imposes restrictions on the use of color, fonts, annotations, and other elements. There are two flavors of PDF/A-1:

- 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 Tagged PDF: it requires structure information and reliable text semantics in order to preserve the document's logical structure and natural reading order. PDF/A-1a not only ensures that the document will look the same when it is processed in the future, but also that its contents can be reliably interpreted and will be accessible to physically impaired users.

PDF/A-1 support in PDFlib is based on the following documents:

- The PDF/A-1 standard (ISO 19005-1:2005)
- Technical Corrigendum 1 (ISO 19005-1:2005/Cor.1:2007)
- Technical Corrigendum 2 (ISO 19005-1:2005/Cor.2:2011)
- TechNote 0010 »Clarifications of ISO 19005« published by the PDF Association.

When PDF/A-1 (without any conformance level) is mentioned, both conformance levels PDF/A-1a and PDF/A-1b are meant.

PDF/A-2a, PDF/A-2b, and PDF/A-2u according to ISO 19005-2. The PDF/A-2 standard flavors are based on ISO 32000-1 (i.e. PDF 1.7), which means that they support more features than PDF/A-1. Unlike PDF/A-1 the newer PDF/A-2 standard allows transparency, layers, JPEG 2000 compression, PDF/A file attachments, PDF packages and other PDF features. PDF/A-2 defines the following flavors:

- ISO 19005-2 Level B conformance (PDF/A-2b) ensures the visual appearance of a document.

- ISO 19005-2 Level A conformance (PDF/A-2a) adds reliable Unicode text semantics and Tagged PDF with structure information. The tags make sure that PDF/A-2a documents are fully accessible.
- ISO 19005-2 Level U conformance (PDF/A-2u) sits in between PDF/A-2a and PDF/A-2b in that it requires reliable Unicode text semantics, but not structure information. PDF/A-2u guarantees that the pages can faithfully be reproduced and that the text can be extracted and searched.

PDF/A-2 support in PDFlib is based on the following documents:
- The PDF/A-2 standard (ISO 19005-2:2011)
- TechNote 0010 »Clarifications of ISO 19005« published by the PDF Association.

When PDF/A-2 (without any conformance level) is mentioned, all three conformance levels PDF/A-2a, PDF/A-2b, and PDF/A-2u are meant.

PDF/A-3a, PDF/A-3b, and PDF/A-3u as defined in ISO 19005-3. PDF/A-3 is similar to PDF/A-2 with the following differences:
- While PDF/A-2 allows only file attachments which conform to PDF/A-1 or PDF/A-2, PDF/A-3 allows arbitrary file types as attachments.
- Attached files are associated with the whole document, a page, or some other element of the document. The relationship between the file attachment and the corresponding part of the document must be specified explicitly, e.g. source, alternative, or supplemental data.

PDF/A-3 support in PDFlib is based on the following document:
- The PDF/A-3 standard (ISO 19005-3:2012)
- TechNote 0010 »Clarifications of ISO 19005« published by the PDF Association.

When PDF/A-3 (without any conformance level) is mentioned, all three conformance levels PDF/A-3a, PDF/A-3b, and PDF/A-3u are meant.

The ZUGFeRD standard for electronic invoices is an important application based on PDF/A-3. It embeds a machine-readable XML version of the invoice in a human-readable document which conforms to PDF/A-3. More information on ZUGFeRD can be found on the PDFlib Website.

12.3.2 General Requirements

Cookbook *Code samples for generating PDF/A can be found in the* `pdfa` *category of the PDFlib Cookbook.*

If the PDFlib client program obeys to the rules documented in this chapter, valid PDF/A output is guaranteed. If PDFlib detects a violation of PDF/A rules it will throw an exception which must be handled by the application. No PDF output is created in this case. Table 12.3 lists general requirements for creating conforming PDF/A output.

Creating combined PDF/A and PDF/UA-1 documents. A PDF/A document can at the same time conform to PDF/UA-1. In fact, if you want to create PDF/A-1a/2a/3a we recommend to adhere to the PDF/UA requirements in order to improve the accessibility of the generated documents. See »Creating combined PDF/UA-1 and PDF/A documents«, page 348, for details and restrictions.

Table 12.3 General requirements for PDF/A conformance levels A, B, and U

item	*PDFlib requirements for PDF/A conformance (all conformance levels)*
PDF/A conformance level and PDF compatibility	*PDF_begin_document( ): the* `pdfa` *option must be set to the required PDF/A conformance level, e.g.* `pdfa=PDF/A-2b` *PDF/A-1: Operations that require PDF 1.5 or above (e.g. layers) must be avoided.* *PDF/A-2/3: Operations that require PDF 1.7ext3 or above (e.g. PDF Portfolios) must be avoided.*
fonts	*The font option* `embedding` *must be* `true`*. The options* `unicodemap=false` *and* `dropcorewidths=true` *are not allowed.* *Embedding is also required for the PDF core fonts. The only exception to the embedding requirement applies to fonts which are exclusively used for invisible text (mainly useful for OCR results). This can be controlled with the* `optimizeinvisible` *option.*
page sizes	*PDF_begin/end_page_ext( ): there are no strict page size limits in PDF/A. However, it is recommended to keep the page size (width and height, and all box entries) in the range 3…14400 points (508 cm) in PDF/A-1, or 3…14400 user units in PDF/A-2/3.*
layers	*PDF/A-1: PDF_define_layer( ) and PDF_set_layer_dependency( ) must be avoided.* *PDF/A-2/3: layers can be used, but some options of PDF_define_layer( ) must be avoided.*
security	*PDF_begin_document( ): the* `userpassword`*,* `masterpassword`*,* `attachmentpassword` *and* `permissions` *options must be avoided.*
external content	*PDF_begin_template_ext( ), PDF_load_graphics( ) and PDF_open_pdi_page( ): the* `reference` *option must be avoided.* *PDF_load_asset( ): the* `external` *option must be avoided.*
file size	*The file size of the generated PDF document must not exceed 2 GB, and the number of PDF objects must be smaller than 8.388.607. See Section 3.1.6, »Maximum Size of PDF Documents and other Limits«, page 61, for more details about these limits.*
PDF import	*PDF_open_pdi_document( ) is restricted unless* `infomode=true`*; see Section 12.3.7, »Importing PDF/A Documents with PDI«, page 326.*

Creating combined PDF/A and PDF/X documents. A PDF/A document can at the same time conform to PDF/X-1a, PDF/X-3, or PDF/X-4, but not to PDF/X-4p or PDF/X-5. In order to create such a combo file supply appropriate values for the *pdfa* and *pdfx* options of *PDF_begin_document( )*, e.g.:

```
ret = p.begin_document("combo.pdf", "pdfa=PDF/A-2b pdfx=PDF/X-4");
```

12.3.3 Color and Image Requirements

PDF/A guarantees faithful color reproduction by requiring device-independent color specifications. Color spaces may come from the following sources:

- images loaded directly with *PDF_load_image( )* and *PDF_fill_imageblock ( )*, and indirectly via *PDF_load_graphics( )*
- explicit color specifications with *PDF_set_graphics_option( )* or *PDF_setcolor( )*
- color specifications through option lists, e.g. in Textflows
- blending color spaces for transparency groups: *PDF_begin/end_page_ext( )*, *PDF_begin_template_ext( )*, and *PDF_load_graphics( )*: option *transparencygroup* with suboption *colorspace*
- alternate color space of spot or DeviceN colors
- annotations and form fields may specify colors for borders, backgrounds and contents.

Table 12.4 lists PDF/A requirements for color processing which must be obeyed in all of the operations listed above.

Table 12.4 Color and image requirements for PDF/A conformance levels A, B, and U

item	PDFlib requirements for PDF/A (all conformance levels)
output condition (output intent)	PDF_load_iccprofile( ) with `usage=outputintent` or PDF_process_pdi( ) with `action=copyoutputintent` 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 and no suitable default color space is present for the page.
grayscale color	Grayscale color can only be used if a grayscale, RGB, or CMYK output intent is present or the `defaultgray` option has been set (the `defaultgray` option is not available for form fields).
RGB color	RGB color can only be used if an RGB output intent is present or the `defaultrgb` option has been set. As an exception, RGB color can always be used for form fields.
CMYK color	CMYK color can only be used if a CMYK output intent is present or the `defaultcmyk` option has been set (the `defaultcmyk` option is not available for form fields).
Separation (spot) and DeviceN color	▸ The alternate color space must conform to the rules above. ▸ PDF/A-2/3: PDF_makespotcolor( ) must be called before PDF_create_devicen( ) for all custom spot colors in the DeviceN color space.
transparency and overprinting	PDF/A-1: Transparency must be avoided; this affects the following API features: ▸ PDF_load_image( ): the `masked` option must be avoided unless the mask refers to a 1-bit image. ▸ PDF_load_image( ): images with implicit transparency (alpha channel) are not allowed; they must be loaded with the `ignoremask` option. ▸ PDF_load_graphics( ): SVG graphics containing transparent elements must be avoided. ▸ PDF_create_gstate( ): the `opacityfill` and `opacitystroke` options must be avoided unless they have a value of 1; if `blendmode` is used it must be `Normal`; if `softmask` is used it must be `none`. ▸ PDF_create_annotation( ): the `opacity` option must be avoided. PDF/A-2/3: Transparency is allowed, but the following rule must be obeyed in PDF_create_gstate( ): `overprintmode=1` is not allowed if the current color space is an ICCBased CMYK color space and `overprintfill` or `overprintstroke` is `true`.
transparency groups	PDF_begin/end_page_ext( ), PDF_begin_template_ext( ), PDF_open_pdi_page( ) and PDF_load_graphics( ): the option `transparencygroup` is restricted as follows: ▸ PDF/A-1: Option `transparencygroup` is not allowed. ▸ PDF/A-2/3: The suboption `colorspace` of the `transparencygroup` option must meet the requirements stated above for grayscale, RGB, and CYMK color. For PDF_open_pdi_page( ) and PDF_load_graphics( ) `transparencygroup=auto` is enforced. ▸ PDF/A-2/3: The suboption `colorspace` of the `transparencygroup` option must meet the requirements stated above for Grayscale, RGB and CMYK color. The option `transparencygroup=none` and the suboption `colorspace=none` are not allowed for PDF_begin/end_page_ext( ).
images and templates	PDF_load_image( ): the `interpolate=true` option must be avoided. PDF/A-2/3: JPEG 2000 images must meet certain conditions, see »JPEG 2000 images«, page 187, for details.

Output intents. The output condition defines the intended target device, which is important for consistent color rendering. Unlike PDF/X, which always requires an output intent, the use of an output intent ICC profile is optional in PDF/A. An output intent is only required if device-dependent colors, e.g. RGB, are used in the document. If only device-independent colors, e.g. ICC-based colors, are used in the document no output intent is required. While PDF/X supports only printer ICC profiles as output intents, in PDF/A also monitor profiles are allowed. This makes it possible to use the widely used sRGB profile as output intent. The output intent can be specified with an ICC profile as follows:

```
icc = p.load_iccprofile("sRGB", "usage=outputintent");
```

As an alternative to loading an ICC profile, the output intent can be copied from an imported PDF/A document (see »Copying the PDF/A output intent from an imported document«, page 327). The output intent of the generated output document must be set exactly once. It should be set immediately after *PDF_begin_document()*.

Color strategies for creating PDF/A. The summary of color strategies in Table 12.5 may be helpful for planning PDF/A applications. The easiest approach which works in many situations is to use the *sRGB* output intent ICC profile since it supports grayscale and RGB color. In addition, *sRGB* is known to PDFlib internally and thus doesn't require any external profile data or configuration.

In order to create black text 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 conforms to PDF/A without any output intent profile (unlike DeviceGray, which requires an output intent profile). PDFlib initializes the current color to black at the beginning of each page. Depending on whether or not an ICC output intent was specified, it will use the DeviceGray or *Lab* color space for black. Use the following call to manually set *Lab* black color:

```
p.set_graphics_option("fillcolor={lab 0 0 0}");
```

Table 12.5 PDF/A color strategies for conformance levels A, B, and U

	color spaces which can be used in the document					
output intent ICC profile	**CIELab**	**ICCBased**	**separation and DeviceN**	**Grayscale[1]**	**RGB[1, 2]**	**CMYK[1]**
none	yes	yes	yes	–	–	–
grayscale	yes	yes	yes	yes	–	–
RGB, e.g. sRGB	yes	yes	yes	yes	yes	–
CMYK	yes	yes	yes	yes	–	yes

1. Device color space without any ICC profile or default color space for the page, pattern or template
2. RGB color is always allowed for form fields.

In addition to the color spaces listed in Table 12.5, custom 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 the PDF/A standard. For custom spot colors the alternate color space must be chosen so that it is compatible with the output intent. For all custom spot colors used in a DeviceN color space *PDF_makespotcolor( )* must be called.

12.3.4 Requirements for Interactive Features

Table 12.6 lists all operations which are restricted when generating PDF/A-conforming output. Calling one of the prohibited functions in PDF/A mode triggers an exception.

Table 12.6 Requirements for interactive features for all PDF/A conformance levels

item	*PDFlib requirements for PDF/A (all conformance levels)*
annotations	*PDF/A-1: PDF_create_annotation( ) is subject to the following restrictions:* ► *Annotations with* `type=FileAttachment` *and* `Movie` *must be avoided.* ► *The* `zoom` *and* `rotate` *options for Text annotations must not be set to* `true`. ► *The* `annotcolor` *and* `interiorcolor` *options can only be used if an RGB output intent has been specified. The* `fillcolor` *option can only be used if an RGB or CMYK output intent has been specified, and a corresponding rgb or cmyk color space must be used.* ► *The* `opacity` *option must not be used.* *PDF/A-2/3: PDF_create_annotation( ): only* `type=Link` *is allowed.*
attachments	*PDF/A-1: PDF_begin/end_document( ): the* `attachments` *option must be avoided.* *PDF/A-2: PDF_begin/end_document( ): the* `attachments` *option must refer to PDF/A-1 or PDF/A-2 documents.* *PDF/A-3: Arbitrary file types can be attached with the* `associatedfiles` *option, but the* `attachments` *option must be avoided. The following conditions must be obeyed:* ► *Attachments can be associated with various parts of the document with the* `associatedfiles` *option of PDF_end_document( ), PDF_begin/end_page_ext( ), PDF_begin/end_dpart( ), PDF_begin_template_ext( ), PDF_load_image( ), PDF_open_pdi_page( ), PDF_load_graphics( ). Each attachment must be associated with exactly one part of the document, i.e. each asset handle created with PDF_load_asset( ) must be supplied to exactly one* `associatedfiles` *option.* ► *The* `mimetype` *and* `relationship` *suboptions are required.* ► *The* `description` *suboption is recommended.* ► *The* `external=true` *suboption must be avoided.*
actions and JavaScript	*PDF_create_action( ): actions with* `type=Hide`, `Launch`, `Movie`, `ResetForm`, `ImportData`, `JavaScript` *must be avoided; for* `type=name` *only* `NextPage`, `PrevPage`, `FirstPage`, *and* `LastPage` *are allowed.* *PDF_begin/end_document( ): the option* `action` *can only be used with the trigger event* `open`. *PDF_begin/end_page_ext( ): option* `action` *must be avoided.*
form fields	*PDF_create_field/fieldgroup( ) are subject to the following restrictions:* ► *All used fonts must be embedded.* ► *Options* `backgroundcolor`, `bordercolor`, `fillcolor`, *and* `strokecolor`: *RGB colors are always allowed, Grayscale colors are only allowed with an output intent (any type), and CMYK colors are only allowed with a CMYK output intent (see also Table 12.4).* ► *The* `action` *option is not allowed.*

12.3.5 Additional PDF/A Requirements for Level U Conformance

Most standard requirements for PDF/A-2u and PDF/A-3u are met automatically by PDFlib. When generating documents with Level U conformance only the operation listed in Table 12.7 is restricted. In other words, if your application already creates PDF/A-2b or PDF/A-3b and you observe the restriction in Table 12.7 the generated documents can also be declared as PDF/A-2u or PDF/A-3u, respectively.

Table 12.7 Additional restriction for PDF/A conformance level U

item	*PDFlib function and option requirements for PDF/A-2u/3u conformance*
fonts	*The font option* `unicodemap=false` *must be avoided.*

12.3.6 Additional PDF/A Requirements for Level A Conformance

When creating PDF/A-1a, PDF/A-2a and PDF/A-3a all Tagged PDF requirements according to Section 11.3, »Tagged PDF Basics«, page 285, must be met. Table 12.8 lists required and recommended operations for generating output according to Level A. In addition to the general Tagged PDF rules we strongly recommend to obey the PDF/UA requirements to improve the accessibility of the generated documents; see Section 12.6, »PDF/UA for Universal Accessibility«, page 348, for details.

The user is responsible for creating correct structure information. 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.

Table 12.8 Additional requirements for PDF/A conformance level A

item	*PDFlib requirements for PDF/A-1a/2a/3a conformance*
fonts	*The font option* `unicodemap=false` *must be avoided.*
Tagged PDF	*All requirements for Tagged PDF must be met (see Section 11.3, »Tagged PDF Basics«, page 285). The structure hierarchy of the document should reflect the logical structure of the document as accurately as possible.*
word boundaries	*Words must be separated by space characters (U+0020). The* `autospace` *option can be used to simplify this task.*
text output and PUA Unicode characters	*PDF/A-2a/3a: PUA Unicode characters (e.g. logos and symbols) must have appropriate replacement text specified in the* `ActualText` *option of PDF_begin_item( ) for the enclosing content item or the equivalent* `tag` *option of the corresponding output function (see below for details).*
annotations	*PDF_create_annotation( ): the* `contents` *option is recommended for annotations which do not display any text.*

PUA characters. PDF/A-2a and PDF/A-3a include additional requirements for characters with a Unicode value in the Private Use Area or PUA, i.e. mainly the range U+E000 - U+F8FF (see »BMP and PUA«, page 105, for details). PUA characters are usually decorative and symbolic glyphs, or custom glyphs such as a company logo. PDF/A-2a/3a require that PUA characters are accompanied by an *ActualText* attribute which contains a textual representation of the character. The *ActualText* may be assigned to an individual PUA character or to a longer sequence of characters which includes a PUA character. It is recommended to supply the *ActualText* option with a *Span* inline-level element.

You can use *PDF_info_font()* to check the Unicode value of a particular code for a specific font (see Section 6.6.2, »Font-specific Encoding, Unicode, and Glyph Name Queries«, page 153):

```
uv = (int) p.info_font(font, "unicode", "code=" + c);
```

If the resulting Unicode value *uv* falls into the PUA it requires an *ActualText* attribute. The following code fragment assumes a font called *PDFlibLogo* which includes a graphical representation of the PDFlib corporate logo. When placing the logo on a page a *Span* element with a suitable *ActualText* suboption containing the text *PDFlib Logo* is supplied in the *tag* option:

```
p.fit_textline(text, 50, 700,
        "fontname=PDFlibLogo encoding=unicode embedding fontsize=24 " +
        "tag={tagname=Span ActualText={PDFlib Logo}}");
```

If you don't have any information about the glyph and therefore no suitable *ActualText* is readily available you may use the name of the glyph in the font. It can be determined as follows:

```
gn_idx = (int) p.info_font(font, "glyphname", "code=" + c);
glyphname = p.get_option(gn_idx, "");
```

The glyph name may be used in *ActualText*, probably in combination with a fixed phrase. For example, code *0x1A* in the *Wingdings* font contains an image of a computer keyboard with the glyph name *keyboard*. This glyph maps to U+F037, i.e. a PUA value. Using the ActualText *symbol for keyboard* may make sense for this symbol. It should be noted that programmatically constructing *ActualText* must be considered a makeshift solution. Human-selected text is always preferable to machine-generated *ActualText*.

12.3.7 Importing PDF/A Documents with PDI

Additional rules apply when pages from an existing PDF document are to be imported into a PDF/A-conforming output document (see Section 8.3, »Importing PDF Pages with PDI«, page 208, for details on PDF import). All imported documents must conform to a PDF/A conformance level which is compatible to the current PDF/A mode according to Table 12.9.

Note PDFlib does not validate PDF input documents for PDF/A conformance, nor can it convert arbitrary input PDF documents to PDF/A.

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 conform to

the selected PDF/A conformance level. Documents which are incompatible to the current PDF/A level will be rejected in *PDF_open_pdi_document()*.

Table 12.9 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-2a, PDF/A-3a**	**PDF/A-2b, PDF/A-3b**	**PDF/A-2u, PDF/A-3u**
PDF/A-1a:2005	allowed	–	–	–	–
PDF/A-1b:2005	allowed	allowed	–	–	–
PDF/A-2a, PDF/A-3a	allowed	–	allowed	–	–
PDF/A-2b, PDF/A-3b	allowed	allowed	allowed	allowed	allowed
PDF/A-2u, PDF/A-3u	allowed	–	allowed	–	allowed

Cookbook *A full code sample can be found in the Cookbook topic* `pdfa/clone_pdfa`.

If one or more PDF/A documents are imported, they must all have been prepared for a compatible output condition according to Table 12.10. The output intents in all imported documents must be identical or compatible; it is the user's responsibility to make sure that this condition is met.

Table 12.10 Output intent compatibility when importing PDF/A documents (all conformance levels)

	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. The output intent of the imported document and the output intent of the generated document must be identical.

While PDFlib can correct certain items, it is not intended to work as a PDF/A validator or to enforce PDF/A conformance for imported documents. For example, PDFlib will not embed fonts which are missing from imported PDF pages.

If you want to combine imported pages such that the resulting PDF output document conforms to the same PDF/A conformance level and output condition as the input document(s), you can query the PDF/A status of the imported PDF as follows:

```
pdfalevel = p.pcos_get_string(doc, "pdfa");
```

This statement retrieves a string designating the PDF/A conformance level of the imported document if it conforms to a PDF/A level, or *none* otherwise. The returned string can be used to set the PDF/A conformance level of the output document appropriately, using the *pdfa* option in *PDF_begin_document()*.

Copying the PDF/A output intent from an imported document. In addition to querying the PDF/A conformance level you can also copy the PDF/A output intent from an

imported document. Since PDF/A documents do not necessarily contain any output intent you must first use pCOS to check for the existence of an output intent before attempting to copy it.

Cookbook *A full code sample can be found in the Cookbook topic* `pdfa/clone_pdfa`.

This can be used as an alternative to setting the output intent via *PDF_load_iccprofile( )*, and will copy the imported document's output intent to the generated output document. Copying the output intent works for imported PDF/A and PDF/X documents.

12.3.8 XMP Metadata for PDF/A

PDF/A relies on the XMP format for embedding metadata in PDF documents. PDF/A supports two kinds of XMP metadata: a set of well-known metadata schemas called predefined schemas which are taken from the underlying version of the XMP specification, and custom extension schemas. PDFlib automatically creates the required PDF/A conformance entries in the XMP as well as several common entries (e.g. *CreationDate)*.

Document-level XMP. XMP document metadata can be supplied with the *metadata* option of *PDF_begin_document( )*, *PDF_end_document( )*, or both. In PDF/A mode PDFlib verifies whether user-supplied XMP document metadata conforms to the PDF/A requirements. XMP metadata from imported PDF documents can be fetched from the input PDF via the pCOS path */Root/Metadata*.

Cookbook *A full code sample can be found in the Cookbook topic* `interchange/import_xmp_from_pdf`.

Component-level XMP. In addition to the document as a whole, XMP metadata can also be attached to other components in a PDF document such as pages or images. While there are no PDF/A-1 requirements for component-level metadata, PDF/A-2 and PDF/A-3 mandate that custom properties in component-level XMP are also described by an extension schema description similar to document-level XMP.

Component-level XMP metadata can be supplied with the *metadata* option of *PDF_begin/end_page_ext( )*, *PDF_load_image( )* and other functions.

Predefined XMP schemas. The use of XMP for document metadata in PDF/A is based on the following specifications:

- PDF/A-1: XMP 2004 specification
- PDF/A-2 and PDF/A-3: XMP 2005

The schemas described in the respective XMP specification are called predefined schemas, and are listed in Table 12.11 along with their namespace URI and the preferred namespace prefix. Only properties of predefined schemas can be used with PDF/A unless an extension schema description is present (see below). The full list of properties in the predefined XMP 2004 schemas for PDF/A-1 is available in TechNote 0008 from the PDF/A Competence Center of the PDF Association. PDF/A-2/3 add predefined schemas from XMP 2005, but the additional schemas are related to images and dynamic media, and are therefore unlikely to be useful for document metadata.

XMP extension schema descriptions. If your metadata requirements are not covered by the predefined schemas you can define an XMP extension schema. PDF/A describes an extension mechanism which must be used when custom schemas are to be embed-

Table 12.11 Predefined XMP schemas for PDF/A (see XMP 2004 and XMP 2005 for details)

Schema name and description	*namespace URI*	*preferred namespace prefix*
XMP 2004 schemas for use in PDF/A-1, PDF/A-2, and PDF/A-3		
Adobe PDF schema	`http://ns.adobe.com/pdf/1.3/`	`pdf`
Dublin Core schema	`http://purl.org/dc/elements/1.1/`	`dc`
EXIF schema for EXIF-specific properties	`http://ns.adobe.com/exif/1.0/`	`exif`
EXIF schema for TIFF properties	`http://ns.adobe.com/tiff/1.0/`	`tiff`
Photoshop schema	`http://ns.adobe.com/photoshop/1.0/`	`photoshop`
XMP Basic Job Ticket schema	`http://ns.adobe.com/xap/1.0/bj`	`xmpBJ`
XMP Basic schema	`http://ns.adobe.com/xap/1.0/`	`xmp`
XMP Media Management schema	`http://ns.adobe.com/xap/1.0/mm/`	`xmpMM`
XMP Paged-Text schema	`http://ns.adobe.com/xap/1.0/t/pg/`	`xmpTPg`
XMP Rights Management schema	`http://ns.adobe.com/xap/1.0/rights/`	`xmpRights`
Additional XMP 2005 schemas for use in PDF/A-2 and PDF/A-3		
Camera Raw Schema	`http://ns.adobe.com/camera-rawsettings/1.0/`	`crs`
EXIF Schema for Additional EXIF Properties	`http://ns.adobe.com/exif/1.0/aux/`	`aux`
XMP Dynamic Media Schema	`http://ns.adobe.com/xmp/1.0/DynamicMedia/`	`xmpDM`

ded in a document. Table 12.12 summarizes the schemas which must be used for describing one or more extension schemas and their properties, along with their namespace URI and the required namespace prefix. Note that the namespace prefixes are required (unlike the preferred namespace prefixes for predefined schemas).

If custom XMP properties for component-level XMP (e.g. on page level) are used, the corresponding extension schema description can be supplied together with the custom XMP properties in the same function (e.g. *PDF_begin_page_ext( )*). As an alternative, the extension schema description for component-level XMP can be supplied with the document-level XMP in *PDF_begin_document( )*.

More details of constructing XMP extension schema descriptions and an example are available in TechNote 0009 from the PDF/A Competence Center.

Cookbook *Full code and XMP samples can be found in the Cookbook topics* `pdfa/pdfa_extension_schema` *and* `pdfa/pdfa_extension_schema_with_type`.

Table 12.12 PDF/A extension schema container schema and auxiliary schemas

Schema name and description	*namespace URI*[1]	*required namespace prefix*
PDF/A extension schema container schema: container for all embedded extension schema descriptions	`http://www.aiim.org/pdfa/ns/extension/`	`pdfaExtension`
PDF/A schema value type: describes a single extension schema with an arbitrary number of properties	`http://www.aiim.org/pdfa/ns/schema#`	`pdfaSchema`

Table 12.12 PDF/A extension schema container schema and auxiliary schemas

Schema name and description	***namespace URI[1]***	***required namespace prefix***
PDF/A property value type: describes a single property	`http://www.aiim.org/pdfa/ns/property#`	`pdfaProperty`
PDF/A `ValueType` *value type: describes a custom value type used in extension schema properties; only required if types beyond the XMP 2004 list of types are used.*	`http://www.aiim.org/pdfa/ns/type#`	`pdfaType`
PDF/A field type schema: describes a field in a structured type	`http://www.aiim.org/pdfa/ns/field#`	`pdfaField`

1. The namespace URIs are incorrectly listed in ISO 19005-1, and have been corrected in Technical Corrigendum 1.

12.4 PDF/X for Print Production

12.4.1 The PDF/X Family of Standards

Note *If you are new to PDF/X the document »PDF/X in a Nutshell« published by the PDF Association is recommended as an introduction (see* `www.pdfa.org/pdfx-in-a-nutshell/`).

The PDF/X formats specified in ISO 15930 strive to provide a consistent and robust subset of PDF which can be used to exchange data suitable for commercial printing. PDFlib can generate output and process input conforming to the PDF/X flavors described below. Table 12.13 summarizes the main differences between the PDF/X flavors.

Table 12.13 Comparison of PDF/X flavors

	PDF/X-3	*PDF/X-4*	*PDF/X-4p*	*PDF/X-5n*
PDF version	*PDF 1.4*	*PDF 1.6*	*PDF 1.6*	*PDF 1.6*
CMYK and spot color	*yes*	*yes*	*yes*	*yes*
device-independent color (ICCBased and Lab)	*yes*	*yes*	*yes*	*yes*
transparency and layers	*–*	*yes*	*yes*	*yes*
externally referenced output intent ICC profile	*–*	*–*	*yes*	*yes*
n-colorant output intent ICC profile	*–*	*–*	*–*	*yes*
externally referenced content	*–*	*–*	*–*	*–*

PDF/X-1a:2003 as defined in ISO 15930-4. This standard is based on PDF 1.4, with some features (e.g. transparency) prohibited. PDF/X-1a supports only CMYK and spot colors, but not modern color management with ICC profiles. For this reason PDF/X-1a is considered outdated.

PDF/X-3:2003 as defined in ISO 15930-6. This standard is based on PDF 1.4 and supports workflows based on device-independent color in addition to grayscale, CMYK, and spot colors. Output devices can be monochrome, RGB, or CMYK. Some PDF 1.4 features, especially transparency, are prohibited. Because of the old PDF version and the exclusion of transparency PDF/X-3 is considered outdated.

PDF/X-4 as defined in ISO 15930-7. This standard is based on PDF 1.6. In PDF/X-4 transparency and layers are allowed, but some other PDF 1.6 features are still prohibited. The variant PDF/X-4p allows output intent ICC profiles to be kept external from the PDF document to save space.

PDFlib implements 15930-7:2010. Compared to the 2008 version the 2010 version introduced changes related to layer handling.

PDF/X-5 as defined in ISO 15930-8. This standard is targeted at »partial exchange« which requires prior discussion between producer and receiver of a file. It can be regarded as an extension of PDF/X-4 and PDF/X-4p (i.e. based on PDF 1.6) and consists of the following flavors:

- PDF/X-5g (deprecated) allows graphical content external from the PDF document;

- PDF/X-5pg (deprecated) allows external graphical content and external output intent ICC profiles;
- PDF/X-5n supports external output intent ICC profiles for n-colorant print characterizations, also called *xCLR* profiles.

If no specific features of PDF/X-5 are required the document should be prepared according to PDF/X-4 or PDF/X-4p since these are the more general standards.

PDFlib implements ISO 15930-8:2010 including Corrigendum 1 published in 2011.

12.4.2 General Requirements

Cookbook *Code samples for generating PDF/X can be found in the* `pdfx` *category of the PDFlib Cookbook.*

If the PDFlib client program obeys to the rules documented in this chapter, valid PDF/X output is guaranteed. If PDFlib detects a violation of PDF/X rules it throws an exception and no PDF output is created. Table 12.14 lists general requirements for creating conforming PDF/X output.

Table 12.14 General requirements for PDF/X conformance

item	***PDFlib requirements for PDF/X compatibility***
PDF/X conformance level and PDF compatibility	*PDF_begin_document( ): the* `pdfx` *option must be set to the required PDF/X conformance level, e.g.* `pdfx=PDF/X-4`. *PDF/X-1a and PDF/X-3: Operations that require PDF 1.5 or above must be avoided.* *PDF/X-4 and PDF/X-5: Operations that require PDF 1.7 or above must be avoided.*
fonts	*The font option* `embedding` *must be* `true`. *Embedding is also required for the PDF core fonts.*
page sizes	*PDF_begin/end_page_ext( ): The page boxes, which are settable via the* `cropbox`, `bleedbox`, `trimbox`, *and* `artbox` *options, must satisfy 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`.
layers	*PDF/X-1a and PDF/X-3: Layers must be avoided.* *PDF/X-4 and PDF/X-5: Layers can be used, but some options of PDF_define_layer( ) must be avoided.*
document info fields and XMP metadata	*The* `Creator` *and* `Title` *info fields must be set to a non-empty value with PDF_set_info( ) or (in PDF/X-4 and PDF/X-5) with the* `xmp:CreatorTool` *and* `dc:title` *XMP properties in the* `metadata` *option of PDF_begin/end_document( )* *Values other than* `True` *or* `False` *for the* `Trapped` *info field in PDF_set_info( ) or the corresponding XMP property* `pdf:Trapped` *in the* `metadata` *option of PDF_begin/end_document( ) must be avoided.*
security	*PDF_begin_document( ): the* `userpassword`, `masterpassword`, *and* `permissions` *options must be avoided.*

Table 12.14 General requirements for PDF/X conformance

item	PDFlib requirements for PDF/X compatibility
external graphical content (references)	*PDF/X-1a/3/4: The* `reference` *option in PDF_begin_template_ext( ), PDF_load_graphics( ) and PDF_open_pdi_page( ) must be avoided.* *PDF/X-5g and PDF/X-5pg (deprecated): The target provided in the* `reference` *option in PDF_begin_template_ext( ), PDF_load_graphics( ) or PDF_open_pdi_page( ) must conform to one of the following standards: PDF/X-1a, PDF/X-3, PDF/X-4, PDF/X-4p, PDF/X-5g, or PDF/X-5pg, and must have been prepared for the same output intent. Since certain XMP metadata entries are required in the target, not all PDF/X documents are acceptable as target. PDF/X documents generated with PDFlib 8 or above can be used as target.*
file size	*PDF/X-4 and PDF/X-5: The file size of the generated PDF document must not exceed 2 GB, and the number of PDF objects must be smaller than 8.388.607. See Section 3.1.6, »Maximum Size of PDF Documents and other Limits«, page 61, for more details about these limits.*
PDF import	*PDF_open_pdi_document( ) is restricted unless* `infomode=true`*; see Section 12.4.6, »Importing PDF/X Documents with PDI«, page 338.*

12.4.3 Output Intent and Color Requirements

PDF/X guarantees faithful color reproduction by requiring device-independent color specifications. Color spaces may come from the following sources:

- images loaded directly with *PDF_load_image( )* and *PDF_fill_imageblock( ),* and indirectly via *PDF_load_graphics( )*
- explicit color specifications with *PDF_set_graphics_option( )* or *PDF_setcolor( )*
- color specifications through option lists, e.g. in Textflows
- alternate color space of spot or DeviceN colors
- blending color spaces for transparency groups: *PDF_begin/end_page_ext( ), PDF_begin_template_ext( ),* and *PDF_load_graphics( ):* option *transparencygroup* with suboption *colorspace*

The operations above are subject to various restrictions when creating PDF/X output as detailed below.

Output intents and standard output conditions. The output intent (also called output condition) defines the intended target device or printing condition, which is mainly useful for reliable proofing. The PDF/X output intent is usually described by a Grayscale, RGB or CMYK ICC printer profile with the exception of PDF/X-5n which supports n-colorant ICC profiles. The details vary among PDF/X flavors:

- PDF/X-1a/3/4/5g: the output intent can be specified by embedding an ICC profile for the target device or printing condition.
- PDF/X-4p, PDF/X-5pg,and PDF/X-5n: by referencing an external ICC profile for the output intent (the *p* in the name of the standard means that an external profile is referenced). The output intent ICC profile is not only referenced by name, but a strong reference is created which is protected by a checksum. Although a referenced ICC profile is not embedded in the PDF output, it must nevertheless be available at PDF creation time to create a strong reference. The *urls* option must be provided with one or more valid URLs where the ICC profile can be found:

```
if (p.load_iccprofile("CGATS TR 001",
        "usage=outputintent urls={http://www.color.org}") == -1)
{
```

```
    /* Error */
}
```

PDF/X-5n: in order to make the output intent ICC profile available to the recipient of the PDF document, the ICC profile can be included in the PDF document via the *embedprofile* option. This attaches the ICC profile to the PDF document similar to a file attachment. If the profile is embedded this way, the *urls* option is not required.

- PDF/X-1a and PDF/X-3: by supplying the name of a standard output intent without embedding the corresponding ICC profile. However, this feature is deprecated.

Table 12.15 compares output intent handling in all PDF/X flavors. The ICC profile for the output intent should be specified by the print provider based on the intended printing process and paper. If the printing process is unknown or the print provider doesn't specify any output intent you can find some recommendations for selecting a suitable ICC profile at www.pdflib.com.

Table 12.15 Output intent handling for various PDF/X flavors

	PDF/X-1a	PDF/X-3	PDF/X-4	PDF/X-4p	PDF/X-5n
output intent color space	*Grayscale or CMYK*	*Grayscale, RGB or CMYK*	*Grayscale, RGB or CMYK ICC profile*		*n-colorant ICC profile* (xCLR)
output intent ICC profile embedding	*otherwise required*		*always embedded*	*external reference, but ICC profile can be attached*	
other restrictions	–	–	*CMYK output intent profile cannot be used for other purposes*		*output intent profile cannot be used for other purposes*

Color requirements for PDF/X-1a. PDF/X-1a is targeted towards CMYK-based workflows. It allows only the color spaces Grayscale, CMYK as well as separation and DeviceN. Table 12.16 lists requirements for the CMYK-based PDF/X-1a standard.

Table 12.16 Color requirements for PDF/X-1a conformance

item	*PDFlib requirements for PDF/X-1a compatibility*
output condition (output intent)	*PDF_load_iccprofile( ) with* usage=outputintent *or PDF_process_pdi( ) with* action=copyoutputintent *must be called immediately after PDF_begin_document( ). A Grayscale or CMYK output intent must be provided.*
grayscale color	*The* defaultgray *option must be avoided.*
RGB color	*RGB color and the* defaultrgb *option must be avoided.*
CMYK color	*The* defaultcmyk *option must be avoided.*
ICC-based color	*The* iccbasedgray/rgb/cmyk *color space in PDF_set_graphics_option( ) or PDF_setcolor( ) and the* iccprofilegray/rgb/cmyk *options must be avoided.*
Lab color	*The* Lab *color space in PDF_set_graphics_option( ) or PDF_setcolor( ) must be avoided.*
Separation (spot) and DeviceN color	*The alternate color space must be Grayscale or CMYK. Since the built-in Pantone database is based on Lab alternate colors these colors cannot be used.*

Color requirements for PDF/X-3, PDF/X-4/4p and PDF/X-5g/pg. PDF/X-3/4 and PDF/X-5g/5pg support device-independent workflows with ICC profiles. Table 12.17 lists requirements for these standards. Since PDF/X-5n targets different workflows its requirements are listed separately in the next section.

A special rule applies to PDF/X-4/5: a CMYK output intent profile (i.e. loaded with *usage=outputintent)* cannot be used for an ICCBased color space (i.e. loaded with *usage=iccbased)* in the same document. This requirement is mandated by the PDF/X standard and applies only to CMYK profiles, but not to Grayscale or RGB profiles. If you run into this conflict you can simply omit the ICC profile and use plain CMYK color instead since the output intent CMYK profile is applied anyway. In order to avoid unnecessary error messages PDFlib ignores a CMYK ICC profile embedded in an image or SVG graphics if it is identical to the PDF/X output intent.

The restriction still applies to imported PDF documents: if an imported page uses the same CMYK ICC profile as the generated document's output intent it is rejected by *PDF_open_pdi_page( )*.

Table 12.17 Color requirements for PDF/X-3, PDF/X-4/4p and PDF/X-5g/5pg conformance

item	***PDFlib requirements for PDF/X-3, PDF/X-4/4p, PDF/X-5g/5pg compatibility***
output condition (output intent)	*PDF_load_iccprofile( ) with* `usage=outputintent` *or PDF_process_pdi( ) with* `action=copyoutputintent` *must be called immediately after PDF_begin_document( ). A Grayscale, RGB or CMYK output intent must be provided.* *PDF/X-3: 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.*
grayscale color	*Grayscale color can only be used if a Grayscale or CMYK output intent is present or the* `defaultgray` *option has been set.*
RGB color	*RGB color can only be used if an RGB output intent is present or the* `defaultrgb` *option has been set.*
CMYK color	*CMYK color can only be used if a CMYK output intent is present or the* `defaultcmyk` *option has been set.*
Separation (spot) and DeviceN color	▸ *PDF/X-3/4 and PDF/X-5g/pg: The alternate color space must conform to the rules above.* ▸ *PDF/X-4 and PDF/X-5g/pg: PDF_makespotcolor( ) must be called before PDF_create_devicen( ) for all custom spot colors in the DeviceN color space.* ▸ *PDF/X-4 and PDF/X-5g/pg: the* `colorspace` *suboption of the* `process` *option of PDF_create_devicen( ) must match the PDF/X output intent.*

Output intent and color requirements for PDF/X-5n. The output intent ICC profile for PDF/X-5n must describe an n-colorant printing device. Such profiles contain the color space designation *xCLR*, where *x* is a hexadecimal digit in the range 2 to F, i.e. *2CLR*...*FCLR* for 2...15 colorants. *xCLR* profiles are not supported in plain PDF, but can only be used as PDF/X-5n output intent. As PDF doesn't support such profiles directly, they cannot be embedded as output intent, but must be referenced as external file. As an external object, however, it can be embedded as file attachment to enhance availability.

xCLR profiles contain a colorant table with the names and color values of named colorants. With respect to the allowed color spaces in a PDF/X-5n document it is relevant whether the *xCLR* output intent profile contains the colorant *Black*, or possibly all of the CMYK colorants *Cyan, Magenta, Yellow, Black*. For example, a *7CLR* profile for a printing process with CMYK, Orange, Green and Violet colorants (CMYKOGV) satisfies both conditions.

Table 12.18 lists requirements for PDF/X-5n.

Table 12.18 Color requirements for PDF/X-5n conformance

item	PDFlib requirements for PDF/X-5n compatibility
output condition (output intent)	*PDF_load_iccprofile( ) with* `usage=outputintent` *or PDF_process_pdi( ) with* `action=copyoutputintent` *must be called immediately after PDF_begin_document( ). An n-colorant* (`xCLR`) *profile must be provided.*
grayscale color	*Grayscale color can only be used if the output intent contains the colorant* `Black` *or the* `defaultgray` *option has been set.*
RGB color	*RGB color can only be used if the* `defaultrgb` *option has been set.*
CMYK color	*CMYK color can only be used if the output intent contains all of the colorants* `Cyan`, `Magenta`, `Yellow` *and* `Black` *or the* `defaultcmyk` *option has been set.*
Separation (spot) and DeviceN color	▸ *The alternate color space must conform to the rules above.* ▸ *PDF_makespotcolor( ) must be called before PDF_create_devicen( ) for all custom spot colors in the DeviceN color space. Spot colors in the colorant list of the output intent are exempted from this requirement.* ▸ *The options* `subtype=nchannel` *and* `process` *of PDF_create_devicen( ) are not allowed.*

Choosing a suitable PDF/X output intent. The PDF/X output intent is usually selected as a result of discussions between you and the print service provider who takes care of print production. If your printer cannot provide any information regarding the choice of output intent, you can find suitable ICC output intent profiles on the Internet.

Color strategies for PDF/X-3/4/5. Regarding color handling the following two strategies can be implemented, possibly in combination:

- Device-independent color: regardless of the type of output intent ICC profile, device-independent color spaces can be used, i.e. ICC-based or CIELab. The Lab color value *(0, 0, 0)* specifies pure black in a device-independent manner. Use the following call to manually set Lab black color:

```
p.set_graphics_option("fillcolor={lab 0 0 0}");
```

- Device-dependent color: device-specific grayscale, RGB, or CMYK color can be used. While grayscale color can be used with any kind of output intent, RGB or CMYK colors can only be used with a matching output intent.

Spot and DeviceN 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/X-3/4/5. For custom spot colors the alternate color space must be chosen so that it is compatible with the output intent. For all custom spot colors used in a DeviceN color space *PDF_makespotcolor( )* must be called.

The summary of color strategies in Table 12.19 may be helpful for planning PDF/X applications.

12.4.4 Image and Transparency Requirements

Raster images must obey the color-related requirements discussed in Section 12.4.3, »Output Intent and Color Requirements«, page 333. Table 12.20 lists additional image-related requirements for PDF/X conformance.

Table 12.19 PDF/X-3/4/5 color strategies

output intent ICC profile	color spaces which can be used in the document					
	CIELab	ICCBased	separation and DeviceN	Grayscale[1]	RGB[1]	CMYK[1]
grayscale	yes	yes	yes	yes	–	–
RGB[2]	yes	yes	yes	yes	yes	–
CMYK	yes	yes	yes	yes	–	yes
xCLR for PDF/X-5n	yes	yes	yes	yes	–	–

1. Device color space without any ICC profile or default color space for the page
2. The PDF/X standard requires a printer profile; monitor profiles, e.g. sRGB, can not be used. RGB printer profiles are very rare.

Table 12.20 image requirements for PDF/X conformance

item	PDFlib requirements for PDF/X compatibility
images	▸ PDF/X-1a: Images and graphics with RGB, ICC-based, YCbCr, or Lab color must be avoided. ▸ PDF/X-1a/3: JBIG2 images must be avoided. ▸ PDF/X-4/5: JPEG 2000 images must satisfy certain conditions, see »JPEG 2000 images«, page 187, for details.
transparency	PDF/X-1a and PDF/X-3: Transparency must be avoided; this affects the following API features: ▸ PDF_load_image( ): the `masked` option must be avoided unless the mask refers to an image with 1-bit depth. ▸ PDF_load_image( ): images with implicit transparency (alpha channel) are not allowed; they must be loaded with the `ignoremask` option. ▸ PDF_load_graphics( ): SVG graphics containing transparent elements must be avoided. ▸ PDF_create_gstate( ): the `opacityfill` and `opacitystroke` options must be avoided unless they have a value of 1; if `blendmode` is used it must be `Normal`; if `softmask` is used it must be `none`. PDF/X-4/5: Transparent images and graphics can be used.
transparency groups	PDF_begin/end_page_ext( ), PDF_begin_template_ext( ), PDF_open_pdi_page( ) and PDF_load_graphics( ): the option `transparencygroup` is restricted as follows: ▸ PDF/X-1a and PDF/X-3: The option `transparencygroup` is not allowed. ▸ PDF/X-4/5p/5pg: The suboption `colorspace` of the `transparencygroup` option must meet the requirements in Table 12.17 for Grayscale, RGB and CMYK color. For PDF_open_pdi_page( ) and PDF_load_graphics( ) `transparencygroup=auto` is enforced. ▸ PDF/X-5n: The suboption `colorspace` of the `transparencygroup` option must meet the requirements stated in Table 12.18 for Grayscale, RGB and CMYK color. The option `transparencygroup=none` and suboption `colorspace=none` are not allowed for PDF_begin/end_page_ext( ).

12.4.5 Requirements for interactive Features

Table 12.21 lists all operations which are restricted when generating PDF/X-conforming output. Calling one of the prohibited functions in PDF/X mode triggers an exception.

Table 12.21 PDF/X requirements for interactive features

item	PDFlib function and option requirements for PDF/X compatibility
annotations and form fields	PDF_create_annotation( ), PDF_create_field( ): annotations inside the `BleedBox` (or `TrimBox`/`ArtBox` if no `BleedBox` is present) must be avoided.
file attachments	PDF/X-1a/3: PDF_begin/end_document( ): option `attachments` must be avoided; PDF_create_annotation( ) with `type=FileAttachment` must be avoided.
actions and JavaScript	PDF_create_action( ): all actions including JavaScript must be avoided.
viewer preferences / view and print areas	PDF_begin/end_document( ): if the `viewarea`, `viewclip`, `printarea`, and `printclip` suboptions for the `viewerpreferences` option are used values other than `media` or `bleed` are not allowed.

12.4.6 Importing PDF/X Documents with PDI

Additional rules apply when pages from an existing PDF document are imported into a PDF/X output document (see Section 8.3, »Importing PDF Pages with PDI«, page 208, for details). All imported documents must conform to a PDF/X level which is compatible to the current PDF/X mode according to Table 12.22. For all allowed combinations with PDF/X-4/5 output the following additional rule must be observed: if an imported page uses the same CMYK ICC profile as the generated document's output intent, it is rejected by PDF_open_pdi_page( ) since this would violate the PDF/X-4/5 standard.

If a certain PDF/X conformance level is configured in PDFlib and the imported documents adhere to a compatible level, the generated output is guaranteed to conform to the selected PDF/X conformance level. Documents which are incompatible to the current PDF/X level will be rejected in PDF_open_pdi_document( ).

Table 12.22 Compatible PDF/X input levels for various PDF/X output levels

	PDF/X level of the imported document						
PDF/X output level	**PDF/X-1a**	**PDF/X-3**	**PDF/X-4**	**PDF/X-4p**	**PDF/X-5g**	**PDF/X-5pg**	**PDF/X-5n**
PDF/X-1a	allowed						
PDF/X-3	allowed	allowed					
PDF/X-4	allowed	allowed	allowed	allowed			
PDF/X-4p	allowed	allowed	allowed	allowed[1]			
PDF/X-5g	allowed	allowed	allowed	allowed	allowed[2]	allowed[2]	
PDF/X-5pg	allowed	allowed	allowed	allowed[1]	allowed[2]	allowed[1,2]	
PDF/X-5n							allowed[3]

1. PDF_process_pdi( ) with `action=copyoutputintent` copies the reference to the external output intent ICC profile.
2. If the imported page contains referenced XObjects, PDF_open_pdi_page( ) copies both proxy and reference.
3. The imported document must use the same output intent (identical checksum).

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 PDF/X validator or to enforce PDF/X compatibility for imported documents. For example, PDFlib will not embed fonts which are missing from imported PDF pages, and does not apply any color correction to imported pages.

If you want to combine imported pages such that the resulting PDF output document conforms to the same PDF/X conformance level and output condition as the input document(s), you can query the PDF/X status of the imported PDF as follows:

```
pdfxlevel = p.pcos_get_string(doc, "pdfx");
```

This statement retrieves 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.

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

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.

12.5 PDF/VT for Variable and Transactional Printing

12.5.1 The PDF/VT Standard

Note *General information about the PDF/VT standard can be found on the PDFlib Web site. If you are new to PDF/VT the document »PDF/VT Application notes« published by the PDF Association is also recommended (see* `www.pdfa.org/publication/pdfvt-application-notes/`*).*

The PDF/VT standard has been published in 2010 as ISO 16612-2. It is »designed to enable variable document printing (VDP) in a variety of environments«. PDF/VT documents contain the final content elements and associated metadata, but not any variables or templates. PDF/VT is based on PDF/X-4/4p and PDF/X-5g/pg, and supports PDF 1.6 features including transparency and layers. In addition to the requirements of PDF/X the PDF/VT standard adds supplementary features to PDF to meet the requirements of high-volume personalized printing. PDF/VT enables high-performance rendering *(RIPping)* of digital print files by adding efficient resource management to PDF.

PDF/VT-1 conformance levels. ISO 16612-2 specifies several PDF/VT conformance levels, all of which are based on PDF 1.6:

- PDF/VT-1 is designed for single-file exchange and is based on PDF/X-4 (PDF/X-4p is not allowed). All resources required for rendering a PDF document are contained in a single PDF/VT-1 file.
- PDF/VT-2 (deprecated) is designed for multi-file exchange and based on one of PDF/X-4p, PDF/X-5g, or PDF/X-5pg. PDF/VT-2 documents may reference external ICC profiles, external page contents, or both. A PDF/VT document and all its referenced PDF files and external ICC profiles are collectively called a PDF/VT-2 file set.

12.5.2 PDF/VT Concepts

This section provides an overview of the technical concepts on which PDF/VT is based.

Document part hierarchy. The document part *(DPart)* hierarchy specifies the sequence and relationship of documents or parts of documents in a PDF/VT file. In a common scenario the PDF/VT file contains sub-documents for many recipients and each document part corresponds to the pages comprising the sub-document for a single recipient. In addition to assigning pages to recipients, the document part hierarchy can also reflect more complex structures. For example, the recipients may be grouped according to the ZIP code in their address, the ZIP codes may be organized according to the state, the states according to the countries, etc. This kind of document organization creates a tree-like structure which includes all pages in the document. The elements of this tree are called DPart nodes, where each inner node contains other DPart nodes and each leaf node specifies one or more pages for a recipient.

The document part hierarchy in a PDF/VT file can be used for accessing pages, alternatively to other methods such as access by page number or by page label. The DPart hierarchy is required in PDF/VT files. The optional record level value selects the level in the DPart hierarchy which corresponds to a record for an individual recipient. This is relevant for the scope hints (see below).

Document Part Metadata (DPM). Each node in the document part hierarchy from the root node down to the leaves in the document tree may contain Document Part Meta-

data (DPM). It can be used to communicate information about a recipient's document and its parts. In particular, properties which are relevant for production (e.g. number of copies of a document part) or information about the recipient (e.g. the corresponding ZIP code) can be encoded in DPM.
JDF (or other) production metadata is not required in PDF/VT, but adds substantial value in JDF-enabled workflows. The PDF/VT standard also specifies a method for representing document part metadata as an external XML document.

Optimizations for recurring graphical content. Print elements are often re-used on multiple pages, e.g. a company logo or product image which appears on all pages of a mailing. Optimized processing of recurring graphical content is an important strategy for improving file size and processing speed of print files. PDF has always supported XObjects as a means for optimizing the file size by including the required data for a print element only once in the file, and allowing references to this data from multiple pages (or multiple instances on the same page). XObjects may contain a raster image or arbitrary text and vector graphics contents. While XObjects in PDF aim at optimizing the overall size of a document, PDF up to now did not include any means for optimized rendering of repeated page contents. There is nothing in PDF which could tell the consuming software that, say, an image on a particular page will appear again on another page later in the same document or maybe in the next print job. PDF/VT extends the existing concept of XObjects in PDF and adds the following means for optimizing print performance:

- *Unique identification:* XObjects can be assigned an identifier (called *GTS_XID)* which is unique across all documents. This identifier can be used for caching implementations which need to identify equivalent XObjects. In simple terms, the graphic which has already been processed for job 1 and is found to be re-used in job 2 does not have to be ripped again, but the rendered results can be taken from the cache.
- *Scope hints and environment context:* XObjects may contain information (called *GTS_Scope)* regarding the range of pages or documents where the graphical content is re-used. This way XObjects can carry information about the useful lifetime of their rendered results in the cache: will the content be re-used only for the current recipient, re-used anywhere else in the same file or file stream, or not at all. If an environment context (called *GTS_Env)* is provided, the XObject may specify global use, i.e. it will be re-used in more than one PDF/VT instance. There are no restrictions regarding the environment context string. For example, a customer or job name can be used for identifying the environment.
- *Encapsulation hints:* XObject caching algorithms in RIPs must take into account the interaction of an XObject with the calling context and existing print elements on the same page (or other pages, e.g. when imposing multiple pages on the same sheet). For example, if an XObject does not specify the color or line width, but varies its appearance based on the color and line width in effect when it is referenced, caching of the rendered result is ineffective due to the varying appearance. A similar situation arises if the XObject contains transparent elements so that the existing background must be blended with the XObject. In order to facilitate caching of XOBjects in the RIP, PDF/VT introduces the concept of Encapsulated XObjects which can be marked as such using the *GTS_Encapsulated* key. Encapsulated XObjects must satisfy certain rules which facilitate caching in the RIP.

All of these entries are optional: PDF/VT does not require any of the optimizations for recurring graphical content, but using them offers significant print performance advantages with PDF/VT capable RIPs.

12.5.3 Summary of Rules for generating PDF/VT-1 and PDF/VT-2

Cookbook *A code sample for generating PDF/VT-1 can be found in the* `pdfvt` *category of the PDFlib Cookbook.*

Creating PDF/VT-1 and PDF/VT-2 with PDFlib is achieved by the following means:

- Since PDF/VT is based on PDF/X, all requirements for the underlying PDF/X conformance level must be met. The *pdfvt* and *pdfx* document options must be used with suitable values.
- Document parts must be specified with the *DPart* API. Each node in the document part hierarchy may optionally carry DPM metadata. The record level in the DPart tree can be specified.
- Scope hints for recurring graphical content should be provided.
- If the document contains transparency additional effort should be spent in order to satisfy the conditions for encapsulated XObjects.
- Additional rules apply when importing pages from existing PDF documents (see Section 12.5.7, »Importing PDF/X and PDF/VT Documents with PDI«, page 347).

Table 12.23 summarizes required and optional operations for generating PDF/VT-conforming output in addition to the corresponding PDF/X requirements listed in Section 12.4.2, »General Requirements«, page 332. The items apply to both PDF/VT-1 and PDF/VT-2 unless otherwise noted. Specific aspects are discussed in more detail in subsequent sections below.

Table 12.23 Required and optional operations for PDF/VT-1 and PDF/VT-2 conformance

item	*PDFlib requirements for PDF/VT conformance*
conformance level	*The* `pdfvt` *option in* PDF_begin_document( ) *must be set to the required PDF/VT conformance level, e.g.* `pdfvt=PDF/VT-1` *PDF/VT-1: The option* `pdfx=PDF/X-4` *is automatically set; supplying any other value for the* `pdfx` *option is an error.* *PDF/VT-2: The option* `pdfx` *must be specified as well with one of the values* `PDF/X-4p`, `PDF/X-5g`, *or* `PDF/X-5pg`.
document part hierarchy	*The document part hierarchy must be specified. This involves the following operations:* ► PDF_begin_document( ): *The* `nodenamelist` *option must specify names for all levels of the DPart tree. The* `recordlevel` *option can be used to specify the level of the DPart tree which corresponds to recipient records.* ► PDF_begin_dpart( ) *and* PDF_end_dpart( ): *The document part hierarchy must be constructed.* ► *Document parts may optionally carry document part metadata (DPM). DPM can be generated by constructing a DPM dictionary with the POCA interface.*
scope hints for recurring graphical content	PDF_load_image( ), PDF_open_pdi_page( ), PDF_begin_template_ext( ) *and* `templateoptions` *option of* PDF_load_graphics( ): *The* `scope` *suboption should be supplied to the* `pdfvt` *option with one of the values* `unknown`, `singleuse`, `record`, `file`, `stream`, *or* `global` *to provide usage hints for recurring images, templates, and imported PDF pages.* *For* `scope=stream` *and* `scope=global` *the suboption* `environment` *must specify a PDF/VT environment context, i.e. an identifier that can be used by a conforming PDF/VT reader to provide a management interface for managing related XObjects. For example, the customer name or job name could be used to identify the environment.*

Table 12.23 Required and optional operations for PDF/VT-1 and PDF/VT-2 conformance

item	*PDFlib requirements for PDF/VT conformance*
external graphical content (references)	*PDF/VT-1 and PDF/VT-2 based on PDF/X-4p: PDF_begin_template_ext( ), PDF_open_pdi_page( ) and PDF_load_graphics( ): The* `reference` *option must be avoided since it is not allowed in the underlying PDF/X standards.*
encapsulated XObjects	*PDF_begin_document( ): If the document doesn't contain any transparent elements the option* `usestransparency=false` *should be supplied so that PDFlib can encapsulate all Form XObjects without a transparency group.* *PDF_load_image( ): Images should use the* `renderingintent` *or* `mask` *option. Otherwise the rules in Section 12.5.6, »Encapsulated XObjects«, page 346, should be obeyed in order to allow as many XObjects as possible to be marked as encapsulated.*

Creating combined PDF/VT and PDF/A documents. It may be useful to create PDF/VT print documents which at the same time conform to PDF/A for archiving (see Section 12.3, »PDF/A for Archiving«, page 319, for more information on PDF/A). Keep the following in mind when creating such dual-use documents:

- Since PDF/A does not allow any external references, only PDF/VT-1 can be combined with PDF/A, but not PDF/VT-2. The following document options can be used to create PDF/VT and PDF/A at the same time:

```
ret = p.begin_document("combo.pdf", "pdfx=PDF/X-4 pdfvt=PDF/VT-1 pdfa=PDF/A-2b");
```

- The restrictions stated in this manual for PDF/VT and PDF/A must be obeyed, and only features which are allowed in both standards can be used.
- Imported PDF documents must adhere to both the PDF/X and PDF/A standards in order to be acceptable.

12.5.4 Document Part Hierarchy and Document Part Metadata (DPM)

Cookbook *Code samples for generating PDF/VT with a document part hierarchy can be found in the* `pdfvt` *category of the PDFlib Cookbook.*

Creating the document part hierarchy. All pages in the document must be organized in the document part hierarchy. In simple cases, e.g. invoices, this tree structure consists of two levels, the *root* level and the *recipient* level. The pages in the document comprise a linear arrangement of recipient records, where each record contains one or more pages. The structure of the document part hierarchy must be specified in the document option list, e.g.

```
if (p.begin_document(outfile,
        "pdfvt=PDF/VT-1 nodenamelist={root recipient} recordlevel=1") == -1)
```

More complex documents may require a deeper document hierarchy, e.g. customized brochures consisting of front, body, and back parts which are maintained in the *docpart* level:

```
if (p.begin_document(outfile,
        "pdfvt=PDF/VT-1 nodenamelist={root recipient docpart} recordlevel=1") == -1)
```

The *recordlevel* option specifies a zero-based index into the *nodenamelist* where the recipient or record level can be found, which is relevant for the *scope=record* option.

After defining the structure of the hierarchy the pages must be grouped in document hierarchy nodes. This is achieved with *PDF_begin/end_dpart( )*:

```
/* Create root node of the DPart hierarchy */
p.begin_dpart("");
        p.begin_dpart("");          /* Create a new node at the recipient level */
            p.begin_page_ext(...);  /* Create one or more pages */
            ...
            p.end_page_ext(...);
        p.end_dpart("");            /* Close recipient node */

        p.begin_dpart("");          /* Create next node at the recipient level */
            p.begin_page_ext(...);  /* Create one or more pages */
            ...
            p.end_page_ext(...);
        p.end_dpart("");            /* Close recipient node */
/* Close root node */
p.end_dpart("");
```

The calls to *PDF_begin/end_dpart( )* must create a tree structure according to the *nodenamelist* option, i.e. the maximum nesting level must correspond to the number of entries in the *nodenamelist* array.

The *groups* option of *PDF_begin_document( )* and the *group* and *pagenumber* options of *PDF_begin/end_page_ext( )* are not allowed in PDF/VT mode since they would interfere with the DPart hierarchy.

Creating Document Part Metadata (DPM). For each node in the document part hierarchy metadata information can be supplied which applies to the corresponding pages for a leaf node, or the whole subtree under this node. While PDF/VT does not prescribe any particular kind of metadata, it is intended to be used with the JDF standard published by the CIP4 organization. In particular, the document called »ICS — Common Metadata for Document Production Workflows« (available from the CIP4 Web site) describes metadata for use in print production workflows.

The CIP4 metadata format makes use of common PDF data types, and the metadata is written to the PDF output in a manner specified in the PDF/VT standard.

PDFlib users can use the POCA (PDF Object Creation API) interface to create arbitrary data structures consisting of PDF dictionaries, arrays, and other data types. The top-level dictionary for DPM metadata can be passed to the *dpm* option of *PDF_begin/end_dpart( )*:

```
dpm           = p.poca_new("type=dict usage=dpm");
cip4_root     = p.poca_new("type=dict usage=dpm");
cip4_metadata = p.poca_new("type=dict usage=dpm");

p.poca_insert(dpm,           "type=dict   key=CIP4_Root value=" + cip4_root);
p.poca_insert(cip4_root,     "type=dict   key=CIP4_Metadata value=" + cip4_metadata);
p.poca_insert(cip4_metadata, "type=string key=CIP4_Conformance value=base");
p.poca_insert(cip4_metadata, "type=string key=CIP4_Creator value=starter_pdfvt1");
p.poca_insert(cip4_metadata, "type=string key=CIP4_JobID value={Kraxi Systems invoice}");

/* Create node in the DPart hierarchy and add DPM metadata  */
p.begin_dpart("dpm=" + dpm);

p.poca_delete(dpm, "recursive=true");
```

Cookbook *The PDFlib pCOS Cookbook includes a code sample for retrieving DPM from a PDF/VT document and creating the corresponding XML representation.*

12.5.5 Scope Hints for recurring Graphical Content

Scope hints. The *scope* suboption of the *pdfvt* option should be supplied with suitable values to *PDF_load_image( )*, *PDF_load_graphics( )*, *PDF_open_pdi_page( )*, and *PDF_begin_template_ext( )*. In order to achieve this, the client application must know where and how often images, pages, and templates will be re-used in the current PDF document or (even better) across multiple documents. Although the *scope* option is not required, it is strongly recommended since it provides important optimization information for the RIP. If no such information is available, the *scope* option should be omitted or specified with the value *unknown* instead of supplying possibly wrong values.

For the scopes *singleuse*, *record* and *file* PDFlib checks whether the scope value is consistent with the actual use of the XObject, and emits a warning in the log file if the scope is too large, e.g. as follows:

```
XObject with handle 9 was assigned PDF/VT scope 'record', but was used only once
(use 'scope=singleuse' instead)
```

If you get such a warning you should check whether you can assign more suitable scope values to the corresponding image, graphic, imported PDF page, or template to avoid unnecessary caching in the RIP, thereby improving print performance.

While too few references to an XObject (relative to the *scope* option) trigger only a warning which indicates a possible optimization in client code, referencing an XObject too often is an error since this would violate the PDF/VT standard. More specifically, an exception will be thrown in the following situations:

- Option *scope=singleuse* was supplied and the XObject is used more than once in the document.
- Option *scope=record* was supplied and the XObject is used in more than one recipient record.

The *environment* suboption must be provided for *scope=stream* and *scope=global*. It should contain a suitable string which will be used to identify cached objects across documents. Depending on the workflow a customer or job reference may be used for this option. For example, a corporate logo which is heavily used across single print jobs may be identified this way.

Unique IDs. PDFlib automatically assigns unique IDs for all imported PDF pages, images and graphics (for graphics the *templateoptions* option must be provided). Equivalent images, pages, and graphics will be assigned identical ID values to enable efficient caching in the RIP. Unique IDs for templates should be provided by the user via the *xid* suboption of the *pdfvt* option. Identifiers for templates should be identical for template definitions which create equivalent PDF Form XObjects according to PDF/VT (i.e. templates with the same visual output). Templates which are not equivalent must have different identifiers or no identifier at all.

If the application needs the unique ID strings created by PDFlib, e.g. for use in DPM metadata, they can be retrieved with the *xid* option of *PDF_info_image( )* for images and templates, *PDF_info_graphics( )* for graphics, and *PDF_info_pdi_page( )* for imported PDF pages.

12.5.6 Encapsulated XObjects

Encapsulating XObjects can greatly improve ripping performance. However, XObjects can only be encapsulated in fully opaque documents, or (in documents where transparency is used) if certain requirements are met.

Transparency in the document. Since PDFlib does not know in advance whether or not transparency will be used in the document the *usestransparency* option can be supplied to *PDF_begin_document( )* as a hint. If the client uses this option to assert that no transparency will be used, all XObjects can be encapsulated. If transparency is used in the document the client must obey certain rules in order to have XObjects encapsulated. If *usestransparency=false* is supplied but the document contains transparent elements nevertheless PDFlib throws an exception.

Transparency is considered as being used in the generated document if one or more of the following conditions are true:

- *PDF_load_image( )* or *PDF_fill_imageblock( )* is called with any of the following options:
  - the image contains an alpha channel and the *ignoremask* option is *false;*
  - the *masked* option with a handle to an image with more than 1 bit per pixel.
- Graphics imported with *PDF_load_graphics( )* contain any transparent element.
- *PDF_create_gstate( )* is called with any of the following options:
  - the *softmask* option with an option list (i.e. not with the keyword *none);*
  - one of the options *opacityfill* or *opacitystroke* with a value different from 1;
  - the *blendmode* option with a value other than *None* and *Normal.*
- PDF pages imported with *PDF_open_pdi_page( )* or *PDF_fill_pdfblock( )* contain any transparent element. Transparency in imported PDF pages can be identified with the pCOS interface and the *usespagetransparency* pseudo object (see pCOS Path Reference for details).

Encapsulating XObjects. PDFlib encapsulates most XObjects according to the following rules:

- Image XObjects created with *PDF_load_image( )* or *PDF_fill_imageblock( )* are encapsulated if any of the following conditions is true:
  - the *renderingintent* option of *PDF_load_image( )* has been supplied with a value different from *Auto;*
  - the *mask* option of *PDF_load_image( )* has been supplied with the value *true.*
- Form XObjects created with *PDF_begin_template_ext( )* are encapsulated if the document option *usestransparency=false* has been supplied, or the *transparencygroup* option has been supplied with the suboptions *colorspace* and *isolated=true.*
- Form XObjects created with *PDF_load_graphics( )* or *PDF_load_image( )* with the *templateoptions* option are always encapsulated. If the document option *usestransparency=true* has been supplied a transparency group with a suitable color space is attached to the Form XObject; otherwise no group is created.
- Form XObjects created with *PDF_open_pdi_page( )* are encapsulated if the imported PDF page doesn't contain any layers or *PDF_open_pdi_document( )* has been called with *uselayers=false.* If the document option *usestransparency=true* has been supplied a transparency group with a suitable color space is attached to the Form XObject; otherwise no group is created. If an imported page contains XObjects which are already encapsulated this property is retained unmodified when importing the page.

If an XObject cannot be encapsulated a warning is written to the log file.

Scope for encapsulated XObjects. It is recommended to supply the *scope* suboption of the *pdfvt* option for all encapsulated XObjects created with *PDF_load_image( )*, *PDF_fill_imageblock( )*, *PDF_load_graphics( )* and *PDF_begin_template_ext( )*. Since it is allowed to supply the *scope* suboption for unencapsulated XObjects as well, it can be supplied to all relevant API calls if the lifetime of an image, graphic, imported page or template is known, regardless of the encapsulation status of the generated XObject.

12.5.7 Importing PDF/X and PDF/VT Documents with PDI

Special rules apply when pages from an existing PDF document will be imported into a PDF/VT-conforming output document (see Section 8.3, »Importing PDF Pages with PDI«, page 208, for details on PDI). All imported documents must conform to compatible PDF/X and PDF/VT conformance levels according to Table 12.24. If a particular PDF/VT conformance level is configured in PDFlib and the imported documents adhere to one of the compatible PDF/X levels, the generated output is guaranteed to conform to the selected PDF/VT conformance level. Imported documents which do not adhere to one of the acceptable PDF/X levels will be rejected.

Table 12.24 Compatible PDF/X and PDF/VT input levels for various PDF/VT output levels

	PDF/X and PDF/VT level of the imported document			
PDF/VT output level	***PDF/X-1a PDF/X-3 PDF/X-4 PDF/VT-1***	***PDF/X-4p PDF/VT-2 based on PDF/X-4p***	***PDF/X-5g PDF/VT-2 based on PDF/X-5g***	***PDF/X-5pg PDF/VT-2 based on PDF/X-5pg***
PDF/VT-1 (always based on PDF/X-4)	*allowed*	*allowed*		
PDF/VT-2 based on PDF/X-4p	*allowed*	*allowed*[1]		
PDF/VT-2 based on PDF/X-5g	*allowed*	*allowed*	*allowed*[2]	*allowed*[2]
PDF/VT-2 based on PDF/X-5pg	*allowed*	*allowed*[1]	*allowed*[2]	*allowed*[1,2]

1. PDF_process_pdi( ) with `action=copyoutputintent` *will copy the reference to the external output intent ICC profile.*
2. If the imported page contains referenced XObjects, PDF_open_pdi_page( ) will copy both proxy and reference to the target.

DPart and DPM limitations with PDF/VT import. Note the following restriction when importing PDF/VT documents: Document part hierarchy (DPart) and document part metadata (DPM) are not imported. They can be queried with pCOS and reconstructed with *PDF_begin/end_dpart( )* and POCA functions.

12.6 PDF/UA for Universal Accessibility

12.6.1 The PDF/UA-1 Standard

Note *General information about the PDF/UA standard can be found on the PDFlib Web site.*

Note *If you are new to PDF/UA the document »PDF/UA in a Nutshell« published by the PDF Association is recommended (see* `www.pdfa.org/publication/pdfua-in-a-nutshell/`*).*

PDF/UA-1 improves the universal accessibility of PDF documents by specifying details of Tagged PDF elements and other document aspects. PDF/UA-1 can be regarded as an application of WCAG 2.0 (Web Content Accessibility Guidelines[1]) to PDF documents.

PDF/UA-1 is based on Tagged PDF as defined in PDF 1.7 and ISO 32000-1. It doesn't add any new features to the PDF file format, but mainly makes some accessibility and tagging aspects required which are optional in PDF 1.7. It also clarifies the relationship of different types of structure elements.

PDF/UA-1 as defined in ISO 14289-1. PDF/UA-1 improves the accessibility of PDF documents with the following means:

- enforcing certain requirements regarding the document structure, i.e. tagging rules;
- requiring certain pieces of auxiliary information, e.g. metadata and alternate text for graphics;
- preventing certain PDF elements which defeat the purpose of accessibility.

PDFlib's implementation of PDF/UA-1 is based on the following document:

- The PDF/UA-1 standard (ISO 14289-1:2014)

Creating combined PDF/UA-1 and PDF/A documents. It may be useful to create PDF/UA documents which at the same time conform to PDF/A for archiving (see Section 12.3, »PDF/A for Archiving«, page 319). In fact, if you want to create PDF/A-1a/2a/3a we recommend to adhere to the PDF/UA requirements in order to improve the accessibility of the generated documents. In order to create combined PDF/A and PDF/UA-1 documents supply appropriate values for the *pdfa* and *pdfua* options of *PDF_begin_document()*, e.g.:

```
ret = p.begin_document("combo.pdf", "pdfa=PDF/A-2a pdfua=PDF/UA-1 lang=en");
```

When creating such dual-use documents keep in mind that combo files which adhere to both standards must obey the requirements imposed by both standards. The PDF compatibility level is the minimum of the PDF compatibility of the involved standards; imported PDF documents must adhere to both PDF/UA and PDF/A standards.

We recommend to avoid PDF/A-1a for Tagged PDF output and work with the newer PDF/A-2a or PDF/A-3a standards instead because there is a minor conflict between PDF/UA-1 and PDF/A-1a: PDF/UA-1 requires the page option *taborder=structure* in the presence of annotations. However, the *taborder* option requires PDF 1.5 and thus cannot be used in PDF/A-1a. As a result, annotations cannot be used in combined PDF/A-1a and PDF/UA-1 documents.

Cookbook *The* `invoice_pdfua1` *sample in the PDFlib Cookbook demonstrates how to create a combined PDF/UA-1 and PDF/A-2a document.*

1. See www.w3.org/TR/WCAG20/

If PDFlib detects a violation of technical PDF/UA requirements it throws an exception. No PDF output is created in this case. Table 12.25 lists general requirements for creating conforming PDF/UA output.

Table 12.25 General requirements for PDF/UA conformance

item	*PDFlib requirements for PDF/UA-1 conformance*
general document requirements	*PDF_begin/end_document( ): the* pdfua *option must be set to* PDF/UA-1 *which requires Tagged PDF (the* tagged *option is automatically set). Operations which require PDF 1.7ext3 or above (e.g. rich media annotations, portfolios) must be avoided. The* lang *option is required.* *Option* viewerpreferences: *only* true *allowed for suboption* displaydoctitle *Option* permissions*: keyword* noaccessible *not allowed*
tagging	*All tagging rules (see »Nesting rules for structure elements«, page 291) must be obeyed; the* checktags *document option must not be set to* none.
fonts	*The font option* embedding *must be* true. *The options* unicodemap=false *and* dropcorewidths=true *are not allowed. Embedding is also required for PDF core fonts.*
text output and PUA Unicode characters	*PUA Unicode characters (e.g. logos and symbols) must have appropriate replacement text specified in the* ActualText *option of PDF_begin_item( ) for the enclosing content item or the equivalent* tag *option of the corresponding output function (see »PUA characters«, page 325).*
invisible text	*The only exception to the embedding requirement applies to fonts which are exclusively used for invisible text (mainly useful for OCR results), i.e.* textrendering=3. *This can be controlled with the* optimizeinvisible *option.* *If invisible text does not have any rendered equivalent (e.g. a scanned image) it must be marked as* Artifact.
layers	*Layers can be used, but some options of PDF_define_layer( ) must be avoided (see PDFlib API Reference).*
external content	*PDF_begin_template_ext( ), PDF_load_graphics( ) and PDF_open_pdi_page( ): the* reference *option must be avoided.*
PDF import	*PDF_open_pdi_document( ): imported documents must conform to PDF/UA; see Section 12.6.4, »Importing PDF/UA Documents with PDI«, page 352.*
metadata	*PDF_set_info( ) with* key=Title *or* metadata *in PDF_begin/end_document( ) with* dc:title *in the supplied XMP must be provided with a non-empty value.*

12.6.2 Tagging Requirements

Since PDF/UA is based on Tagged PDF the requirements discussed in Section 11.3, »Tagged PDF Basics«, page 285, must be obeyed. However, PDF/UA-1 imposes a number of additional tagging requirements to improve accessibility.

Semantic Requirements. The user must create a document hierarchy and obey the semantic rules listed below. Selecting appropriate structure elements is a crucial component of PDF/UA standard conformance. It is important to understand that the application is responsible for these aspects since PDFlib cannot check them:

- Tagging must use structure elements which are appropriate for the document structure: if it's a heading, it must be tagged as heading. If it's a table, it must be tagged as a table.
- Contents which are not meaningful for the document must not be included in the document hierarchy, but instead must be tagged as *Artifact*.

- Structure elements must be arranged in logical reading order. This can be accomplished most easily by creating the tags in reading order. However, for complex layouts this can also be accomplished with *PDF_activate_item( )* (see Section 11.4.4, »Creating Contents out of Order«, page 309).
- Content must be tagged appropriately if the intended information is not otherwise accessible because of the content's color, format or layout.
- Text represented in a graphic requires the *Alt* option with an explanation if it doesn't contain text in a natural language (e.g. font or script samples).
- Image captions must be marked with a *Caption* tag.
- List elements *(L)* must be created if the content is intended to be read as a list.
- Headers and footers must be tagged as *Artifact* with *artifacttype=Pagination* and *artifactsubtype=Header* or *Footer*.
- Only a single *Figure* tag must be created for groups of graphical elements which logically belong together.
- Footnotes, endnotes, note labels and references to locations within the document must be tagged as *Note* or *Reference* as appropriate.

Note *The document »Tagged PDF Best Practice Guide: Syntax« published by the PDF Association provides guidance regarding the correct use of Tagged PDF structure elements for PDF/UA-1.*

Tag-specific requirements. Table 12.26 lists requirements for specific tags to achieve PDF/UA-1 conformance. These rules must also be obeyed for custom element types which are mapped to the listed standard types. For example, if the custom tag *Illustration* is mapped to *Figure* in the *rolemap* option, it is also subject to the conditions for *Figure*.

Table 12.26 Tag-specific requirements for PDF/UA-1 conformance

element type	PDFlib requirements for PDF/UA-1 conformance
standard element types	*PDF_begin_document( ): the* `rolemap` *option must not map standard element types.*
Figure	*One of the options* `Alt` *or* `ActualText` *must be supplied.*
Formula	*The option* `Alt` *must be supplied.*
Table	*Table elements must be created for logical tables, but must not be created for tables which are created for layout purposes. Tables formatted by PDFlib can be tagged automatically, see Section 11.4.1, »Automatic Table Tagging«, page 301.*
TH	*Tables should include headers. The option* `Scope` *is recommended for* `TH` *elements (it is always created by automatic table tagging if the* `header` *option of PDF_fit_table( ) is used).*
L	*The element type* `L` *(list) requires the* `ListNumbering` *option. If none of its* `LI` *(list item) children contains a* `Lbl` *(label) element* `ListNumbering` *must be* `None`*. On the other hand, if* `ListNumbering=None`*, but there are visible list labels these should be marked as Artifacts.*
Note	*The* `id` *option is required for footnotes and endnotes tagged as* `Note` *(although this doesn't provide any advantage since the* `id` *isn't referenced anywhere).*

Headings. Appropriate heading tags must be used for all headings. There are two approaches regarding the hierarchic nesting of headings in a PDF document:

- Strongly structured documents: grouping elements nest as deeply as required to reflect the organization of the content with articles, sections, subsections etc. At each

level, the children of the grouping element should consist of a heading *H*, one or more paragraphs *P* for content at that level, and additional grouping elements for nested subsections. Strong structure is typically used in XML documents.

- Weakly structured documents: the document's structure hierarchy is relatively flat, having only one or two levels of grouping elements, with all the headings, paragraphs, and other BLSEs as their immediate children. The organization of the content is not reflected in the logical structure, but may be expressed by specific heading levels *H1, H2, H3 etc.* Heading tags may not have any descendants. Weak document structure is typically used in HTML.

In PDF/UA-1 mode this distinction must be made explicit with the *structuretype* option in *PDF_begin_document( )*. PDFlib enforces the following rules regarding the use of heading elements depending on the type of document structure.

- All documents:
  - The *Title* option should be used in all heading tags to denote document sections (e.g. »Chapter 1«)
  - Heading elements *H, H1, H2,* etc. must not have any descendants.
- Weakly structured documents, i.e. *structuretype=weak* (the default):
  - Heading sequences must start at *H1* and must not skip any numeric level. For example, the sequence *H1 H3* is not allowed.
  - Additional heading levels *H7, H8* etc. may be used if more than six heading levels are required. Since these are not standard element types they require an entry in the *rolemap* option. The recommended mapping is *P*.
  - Unnumbered heading elements *H* must not be used.
- Strongly structured documents *(structuretype=strong)*:
  - *H* must be used for headings, but there cannot be more than one *H* tag in each node of the structure hierarchy.
  - Numbered heading elements *H1, H2,* etc. must not be used.

12.6.3 Additional Requirements for specific Content Types

Table 12.27 lists PDF/UA-1 requirements and recommendations related to various content types and interactive elements.

Table 12.27 PDF/UA-1 requirements and recommendations for specific content types and interactive elements

Content type	*PDFlib requirements for PDF/UA-1 conformance*
text	*The natural language of all text on a page must be declared; changes of natural language within a sequence of text must also be declared. The natural language can be declared with the* `lang` *option of PDF_begin_document( ) and other means; see »Language specification«, page 296, for details. The language attribute of an element is inherited to all its descendants.*
vector graphics and raster images	*Raster images and vector graphics must be tagged as* `Artifact`, `Figure`, *or* `Formula`. *This affects low-level path construction functions like PDF_rect( ) etc., path objects with PDF_draw_path( ), PDF_fit_image( ) and similar functions. SVG graphics containing vector graphics or raster images are also affected by this requirement.*
imported PDF pages	*PDF pages placed with PDF_fit_pdi_page( ) which contain a graphic should be tagged as* `Artifact` *or* `Figure`.

Table 12.27 PDF/UA-1 requirements and recommendations for specific content types and interactive elements

Content type	PDFlib requirements for PDF/UA-1 conformance
annotations	*If annotations are present on the page: PDF_begin/end_page_ext( ): only the value* `structure` *is allowed for the option* `taborder`. *PDF_create_annotation( ) with* `type=Link`: ▸ *The annotation must be contained in a* `Link` *structure element.* ▸ *The* `ismap` *option of PDF_create_action( ) is not allowed for actions in Link annotations.* *PDF_create_annotation( ) with a type different from* `Link` *and* `Popup`: ▸ *The annotation must be contained in an* `Annot` *structure element.* ▸ *The* `contents` *option or the* `tag` *option with the suboption* `ActualText` *is required for visible annotations*[1].
form fields	*If any form field is present on the page: PDF_begin/end_page_ext( ): only the value* `structure` *is allowed for option* `taborder`. *PDF_create_field( ) and options* `fieldname`, `fieldtype` *of PDF_add_table_cell( ): a* `Form` *tag must be created with PDF_create_field( ) or the* `tag` *option. The* `tooltip` *option of PDF_create_field( ) and PDF_create_fieldgroup( ) is required.*
page labels	*Page labels created with the* `labels` *option in PDF_begin/end_document( ) and* `label` *option in PDF_begin/end_page_ext( ) should be semantically appropriate.*
bookmarks	*Generating bookmarks with PDF_create_bookmark( ) is recommended. The bookmarks should reflect proper reading order and nesting of the content.*
attachments	*PDF_load_asset( ): option* `description` *is recommended. Attachments should be accessible in their own right.*

1. An annotation is regarded as visible if its rectangle lies at least partially inside the page's CropBox and the `display` *option of PDF_create_annotation( ) is different from* `hidden` *and* `noview`.

12.6.4 Importing PDF/UA Documents with PDI

Additional rules apply when pages from an existing PDF document are imported into a PDF/UA-conforming output document (see Section 8.3, »Importing PDF Pages with PDI«, page 208, for details on PDF import). In order to import pages from existing PDF documents the imported documents and pages must be compatible to the current document according to the following criteria (otherwise they will be rejected):

- *PDF_open_pdi_document( ):* only PDF/UA-1 documents can be imported and the *usetags* option must be *true*. Documents which do not conform to PDF/UA-1 according to the standard identification in the XMP document metadata will be rejected in *PDF_open_pdi_document( )*.
- *PDF_open_pdi_page( ):* the rolemap of the imported document must be compatible with the mapping provided by the *rolemap* option of *PDF_begin_document( )*. This means that custom element types must not be mapped to different standard types by the *rolemap* option and the rolemap of the imported document (or the rolemap of a previously imported document)
- *PDF_open_pdi_page( ):* the heading structure of the imported page must be compatible with the structure type of the generated document, i.e. if *structuretype=weak* only *H1*, *H2*, etc. (but not *H)* must be used on the page; if *structuretype=strong* only *H* (but not *H1*, *H2*, etc.) must be used on the imported page. Pages with both numbered and unnumbered headings will be rejected.

If PDF/UA-1 conformance is configured in PDFlib and the imported document adheres to the requirements above, the generated output is guaranteed to conform to PDF/UA-1 as well.

Note *PDFlib does not validate PDF input documents for PDF/UA conformance, nor can it convert arbitrary input PDF documents to PDF/UA.*

13 PPS and the PDFlib Block Plugin

The PDFlib Personalization Server (PPS) supports a template-driven PDF workflow for variable data processing. Using the Block concept, imported pages can be populated with variable amounts of single- or multi-line text, images, PDF pages or vector graphics. This can be used to easily implement applications which require customized PDF documents, for example:

- mail merge
- flexible direct mailings
- transactional and statement processing
- business card personalization

You can create and edit Blocks interactively with the PDFlib Block Plugin, convert existing PDF form fields to PDFlib Blocks with the form field conversion Plugin. Blocks can be filled with PPS. The results of Block filling with PPS can be previewed in Acrobat since the Block Plugin contains an integrated version of PPS.

Note *Block processing requires the PDFlib Personalization Server (PPS). Although PPS is contained in all PDFlib packages, you must purchase a license key for PPS; a PDFlib or PDFlib+PDI license key is not sufficient. The PDFlib Block Plugin for Adobe Acrobat is required for creating Blocks in PDF templates interactively.*

Cookbook *Code samples regarding variable data and Blocks can be found in the* `blocks` *category of the PDFlib Cookbook.*

13.1 Installing the PDFlib Block Plugin

The Block Plugin works with the following Acrobat versions (it doesn't work with Adobe Reader):

- Windows: Acrobat 8/9/X/XI/DC 32-bit
- Windows: Acrobat DC 64-bit
- macOS: Acrobat DC

Since Acrobat DC is available in 32-bit and 64-bit versions two different installers are available. It is important to use the appropriate installer which matches the installed Acrobat version.

Installing the PDFlib Block Plugin on Windows. To install the PDFlib Block Plugin and the PDF form field conversion Plugin in Acrobat, the plugin files must be placed in a subdirectory of the Acrobat plugin folder. This is done automatically by the plugin installer, but can also be done manually. The plugin files are called *Block.api* and *AcroFormConversion.api*.

The plugin folder for Acrobat 32-bit on 64-bit Windows typically looks as follows:

```
C:\Program Files (x86)\Adobe\Acrobat DC\Acrobat\plug_ins\PDFlib Block Plugin
```

The plugin folder for Acrobat 64-bit typically looks as follows:

```
C:\Program Files\Adobe\Acrobat DC\Acrobat\plug_ins\PDFlib Block Plugin
```

Installing the PDFlib Block Plugin on macOS. Proceed as follows to install the Plugin for all users:

- Double-click the disk image to mount it. A folder with the Plugin files will be visible.
- Copy the Plugin folder to the following path in the system's *Library* folder (create the *Plug-Ins* folder if it doesn't yet exist):

```
/Library/Application Support/Adobe/Acrobat/XXX/Plug-ins
```

Alternatively you can install the Plugin only for a single user as follows:

- Click the desktop to make sure you're in the Finder, hold down the *Option* key, and choose *Go, Library* to open the user's *Library* folder.
- Copy the Plugin folder to the following path in the user's *Library* folder (create the *Plug-Ins* folder if it doesn't yet exist):

```
/Users/<username>/Library/Application Support/Adobe/Acrobat/XXX/Plug-ins
```

Multi-lingual Interface. The PDFlib Block Plugin supports multiple languages in the user interface. Depending on the application language of Acrobat, the Block Plugin chooses its interface language automatically. Currently English, German and Japanese interfaces are available. If Acrobat runs in any other language mode, the Block Plugin uses the English interface.

Sandbox Protection for Acrobat DC on Windows. Acrobat DC 2020 introduced a new security model called *Sandbox Protections* which can be activated via *Preferences, Security (Enhanced), Protected Mode* and *Protected View*. If it is enabled various operations are restricted and a yellow bar with a security message appears at the top of the document window. More information about Sandbox Protections can be found at:

```
helpx.adobe.com/acrobat/using/whats-new/2020-august.html
www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/sandboxprotections.html
```

If Sandboxing is enabled it affects the Preview feature of the PDFlib Block Plugin. Protected View by default grants access to Acrobat's *AppData* directory, the temp directory and several other directories, but not to arbitrary user directories. The Block Plugin can only read from and write to directories which are included in the default directory list of Protected View or which have been configured (whitelisted) in a policy file at the following location (for 32-bit and 64-bit versions of Acrobat):

```
C:\Program Files (x86)\Adobe\Acrobat DC\Acrobat\PDFlibBlockCustomPolicies.txt
C:\Program Files\Adobe\Acrobat DC\Acrobat\PDFlibBlockCustomPolicies.txt
```

By default the policy file grants access to the following directories, but more directory names can be added by the Administrator:

```
; Protected Path Section
FILES_ALLOW_ANY = C:\Users\<username>\*.*
FILES_ALLOW_ANY = C:\Users\Public\*.*
```

If Protected Mode or Protected View is enabled and directories are used which are not whitelisted, some features of the Block Plugin including Preview and Block import/export may fail.

Troubleshooting. If the PDFlib Block Plugin doesn't seem to work check the following:

- Make sure that in *Edit*, *Preferences*, *[General...]*, *General* the box *Use only certified plug-ins* is unchecked. The plugins are not loaded if Acrobat runs in Certified Mode.
- Some PDF forms created with Adobe Designer or Adobe Experience Manager may prevent the Block Plugin as well as other Acrobat plugins from working properly since they interfere with Acrobat's internal security model. For this reason we suggest to avoid Designer's static PDF forms, and only use dynamic PDF forms as input for the Block Plugin.

13.2 Overview of the Block Concept

13.2.1 Separation of Document Design and Program Code

PDFlib Blocks make it easy to place variable text, images, PDF pages or vector graphics on imported pages. In contrast to simple PDF pages, pages with Blocks intrinsically carry information about the required processing which will be performed later on the server side. The Block concept separates the following tasks:

- The designer creates the page layout and specifies the location of variable page elements along with relevant properties such as font size, color, or image scaling. After creating the layout as a PDF document, the designer uses the PDFlib Block Plugin for Acrobat to specify variable data Blocks and their associated properties.
- The programmer writes code to connect the information contained in PDFlib Blocks on imported PDF pages with dynamic information, e.g., database fields. The programmer doesn't need to know any details about a Block (whether it contains a name or a ZIP code, the exact location on the page, its formatting, etc.) and is therefore independent from any layout changes. PPS will take care of all Block-related details based on the Block properties found in the file.

In other words, the code written by the programmer is »data-blind« – it is generic and does not depend on the particulars of any Block. For example, the designer can move the Block with name of the addressee in a mailing to a different location on the page, or change the font size. The generic Block handling code doesn't need to be changed, and will generate correct output once the designer changed the Block properties with the Acrobat plugin to use the first name instead of the last name.

As an intermediate step Block filling can be previewed in Acrobat to accelerate the development and test cycles. Block previews are based on default data (e.g. a string or an image file name) which is specified in the Block definitions.

13.2.2 Block Properties

The behavior of Blocks can be controlled with Block properties. Properties are assigned to a Block with the Block Plugin.

Predefined Block properties. Blocks are defined as rectangles on the page which are assigned a name, a type, and an open set of properties which will later be processed by PPS. The name is an arbitrary string which identifies the Block, such as *firstname*, *lastname*, or *zipcode*. PPS supports different kinds of Blocks:

- *Textline Blocks* hold a single line of textual data which will be processed with the Textline output method in PPS.
- *Textflow Blocks* hold one or more lines of textual data. Multi-line text will be formatted with the Textflow formatter in PPS. Textflow Blocks can be linked so that one Block holds the overflow text of the previous Block (see »Linking Textflow Blocks«, page 379).
- *Image Blocks* hold a raster image. This is similar to placing a TIFF or JPEG file in a DTP application.
- *PDF Blocks* hold arbitrary PDF contents imported from a page in another PDF document. This is similar to placing a PDF page in a DTP application.
- *Graphics Blocks* hold vector graphics. This is similar to placing an SVG file in a layout application.

Blocks can carry a number of predefined properties depending on their type. Properties can be created and modified with the Block Plugin (see Section 13.3.2, »Editing Block Properties«, page 364). A full list of predefined Block properties can be found in Section 13.7, »Block Properties«, page 382. For example, a text Block can specify the font and size of the text, an image or PDF Block can specify the scaling factor or rotation PPS offers dedicated functions for processing the Block types, e.g. *PDF_fill_textblock()*. These functions search a placed PDF page for a Block by its name, analyze its properties, and place client-supplied data (single- or multi-line text, raster image, PDF page, or vector graphics) on the new page according to the specified Block properties. The programmer can override Block properties by specifying the corresponding options in the Block filling functions.

Properties for default contents. Special Block properties can be defined which hold the default contents of a Block, i.e. the text, image, PDF, or graphics contents that will be placed in the Block if no variable data is supplied to the Block filling functions, or in situations where the Block contents are currently constant, but may change in the next print run.

Default properties are also used by the Preview feature of the Block Plugin (see Section 13.5, »Previewing Blocks in Acrobat«, page 372).

Custom Block properties. Predefined Block properties make it possible to quickly implement variable data processing applications, but they are restricted to the set of properties which are internally known to PPS and can automatically be processed. In order to provide more flexibility, the designer can also assign custom properties to a Block. These can be used to extend the Block concept in order to match the requirements of more advanced variable data processing applications.

There are no rules for custom properties since PPS will not process custom properties in any way, except making them available to the client. The client code can retrieve the value of custom properties and process it as appropriate. Based on a custom property of a Block the application may make layout-related or data-gathering decisions. For example, a custom property for a scientific application could specify the number of digits for numerical output, or a database field name may be defined as a custom Block property for retrieving the data corresponding to this Block.

13.2.3 Why not use PDF Form Fields?

Experienced Acrobat users may ask why we implemented a new Block concept instead of relying on the existing form field mechanism available in PDF. The primary distinction is that PDF form fields are optimized for interactive filling, while PDFlib Blocks are targeted at automated filling. Applications which need both interactive and automated filling can combine PDF forms and PDFlib Blocks with the form field conversion plugin (see Section 13.4, »Converting PDF Form Fields to PDFlib Blocks«, page 369).

Although there are many parallels between both concepts, PDFlib Blocks offer several advantages over PDF form fields as detailed in Table 13.1.

Table 13.1 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*
OpenType layout features	*–*	*dozens of OpenType layout features, e.g. ligatures, swash characters, oldstyle figures*
complex script support	*limited*	*shaping and bidirectional formatting, e.g. for Arabic and Devanagari*
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, JBIG2, JPEG 2000, TIFF*
use vector graphics for filling	*–*	*SVG*
color support	*RGB*	*grayscale, RGB, CMYK, Lab, spot color (HKS and Pantone spot colors integrated in the Block Plugin), DeviceN*
PDF standards	*–*	*PDF/A, PDF/X, PDF/VT, PDF/UA*
graphics and text properties can be overridden upon filling	*–*	*yes*
transparent contents	*–*	*yes*
Text Blocks can be linked	*–*	*yes*

13.3 Editing Blocks with the Block Plugin

13.3.1 Creating Blocks

Activating the Block tool. The Block Plugin for creating PDFlib Blocks is similar to the form tool in Acrobat. All Blocks on the page will be visible when the Block tool is active. When another Acrobat tool is selected the Blocks will be hidden, but they are still present. You can activate the Block tool in the following ways:

- By clicking the Block icon  which you can locate as follows in Acrobat DC: click *Tools, Advanced Editing*.
- Via the menu item *PDFlib Blocks, PDFlib Block Tool*.

Creating and modifying Blocks. When the Block tool is active you can drag the crosshair pointer to create a Block at the desired position on the page and with the desired size. Blocks are always rectangular with edges parallel to the page edges (use the *rotate* property for Block contents which are not parallel to the page edges). After dragging a Block rectangle the Block properties dialog appears where you can edit the properties of the Block (see Section 13.3.2, »Editing Block Properties«, page 364). The Block tool automatically creates a Block name which can be changed in the properties dialog. Block names must be unique on a page, but can be repeated on another page.

You can change the Block type in the top area to one of *Textline*, *Textflow*, *Image*, *PDF*, or *Graphics*. Different colors are used for representing the Block types (see Figure 13.1). The *Block Properties* dialog hierarchically organizes the properties in groups and subgroups depending on the Block type.

Note *After you added Blocks or made changes to existing Blocks in a PDF, use Acrobat's »Save as...« command (as opposed to »Save«) to achieve smaller file sizes.*

Selecting Blocks. Several Block operations, such as copying, moving, deleting, or editing Properties, work with one or more selected Blocks. You can select Blocks with the Block tool as follows:

- To select a single Block simply click on it.
- To select multiple Blocks hold down the Shift key while clicking on the second and subsequent Block.
- Press Ctrl-A (on Windows) or Cmd-A (on macOS) 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 macOS. 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.

If you right-click (or Ctrl-click on macOS) an area on the page where no Block is located the context menu contains entries for creating a Block Preview and for configuring the Preview feature.

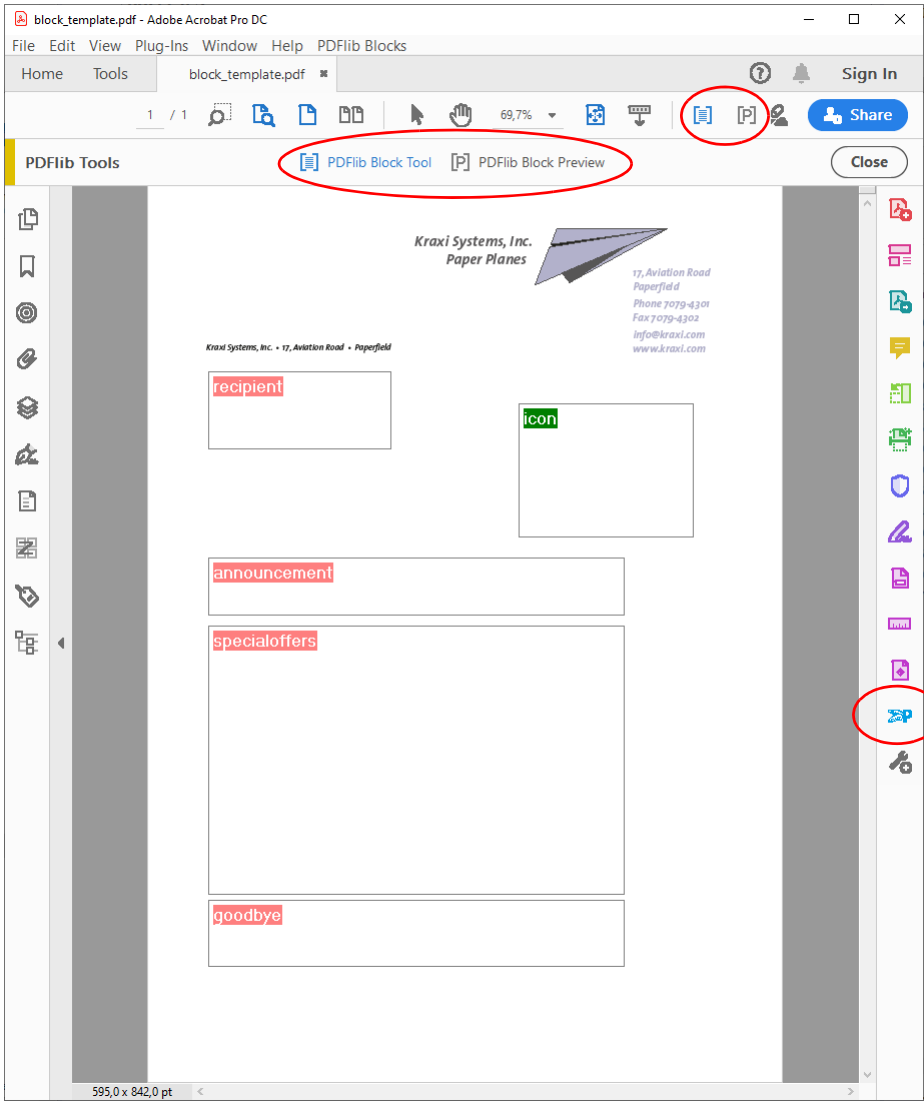


Fig. 13.1
Visualization of Blocks

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 a double arrow and you can resize the Block.

To adjust the position or size of multiple Blocks, select two or more Blocks and use the *Align*, *Center*, *Distribute*, or *Size* commands from the *PDFlib Blocks* menu or the context menu. The position of one or more Blocks can also be changed in small increments by using the arrow keys.

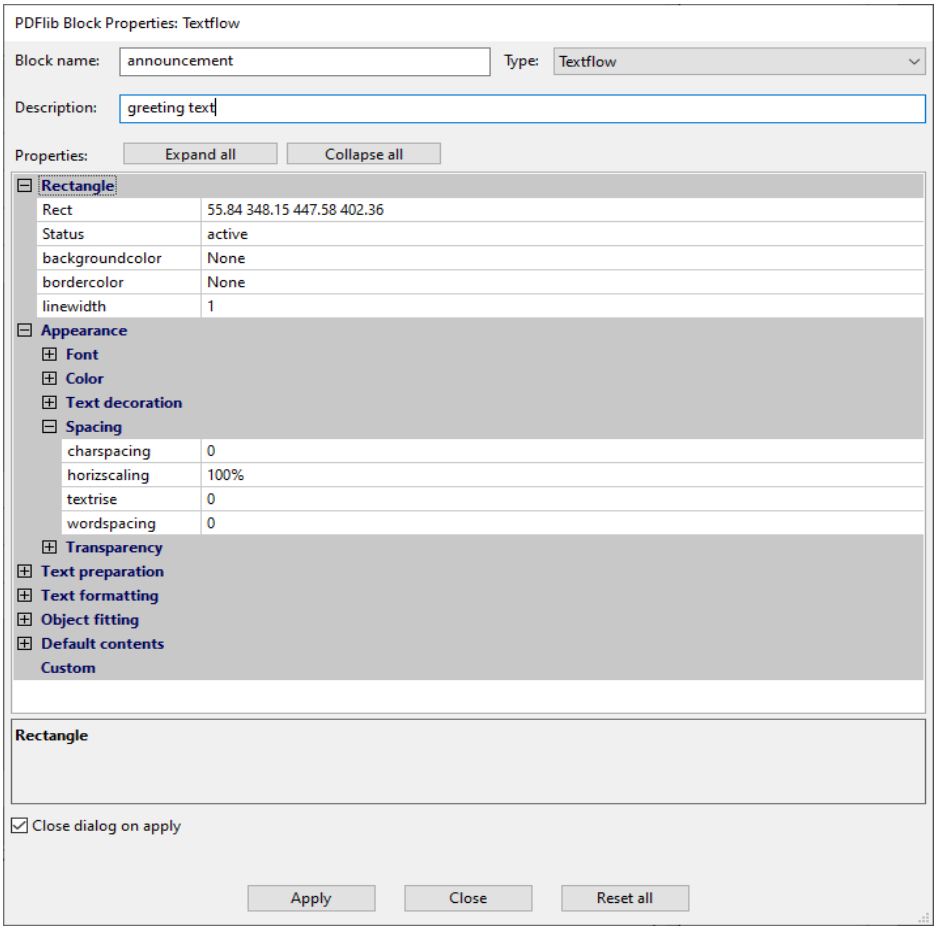


Fig. 13.2
The Block properties dialog

Alternatively, you can enter numerical Block coordinates in the properties dialog. The origin of the coordinate system is in the upper left corner of the page. The coordinates will be displayed in the unit which is currently selected in Acrobat:

- To change the display units in Acrobat DC proceed as follows: go to *Edit, Preferences, [General...], Units & Guides, Page & Ruler Units* and choose one of Points, Inches, Millimeters, Picas, Centimeters.
- To display cursor coordinates use *View, Show/Hide, Cursor Coordinates*.

Note that the selected unit will only affect the *Rect* property, but not any other numerical properties (e.g. *fontsize)*.

Using a grid to position Blocks. You can take advantage of Acrobat's grid feature for precisely positioning and resizing Blocks:

- Display the grid: *View, Show/Hide, Rulers & Grids, Grid*;
- Enable grid snapping: *View, Show/Hide, Rulers & Grids, Snap to Grid*;
- Change the grid: go to *Edit, Preferences, [General...], Units & Guides*. Here you can change the spacing and position of the grid as well as the color of the grid lines.

If *Snap to Grid* is enabled the size and position of Blocks will be aligned with the configured grid. *Snap to Grid* affects newly generated Blocks as well as existing Blocks which are moved or resized with the Block tool.

Creating Blocks by selecting an image or graphic. As an alternative to manually dragging Block rectangles you can use existing page contents to define the Block size. First, make sure that the menu item *PDFlib Blocks, Click Object to define Block* is enabled. Now you can use the Block tool to click on an image on the page in order to create a Block with the same size and location as the image. You can also click on other graphical objects, and the Block tool will try to select the surrounding graphic (e.g., a logo). The *Click Object* feature is intended as an aid for defining Blocks. If you want to reposition or resize the Block you can do so afterwards without any restriction. The Block will not be locked to the image or graphics object which was used as a positioning aid.

The *Click Object* feature tries to recognize which vector graphics and images form a logical element on the page. When some page content is clicked, its bounding box (the surrounding rectangle) will be selected unless the object is white or very large. In the next step other objects which are partially contained in the detected rectangle will be added to the selected area, and so on. The final area will be used as the basis for the generated Block rectangle. The end result is that the *Click Object* feature will try to select complete graphics, not only individual lines.

Automatically detect font properties. The Block Plugin can analyze the underlying font which is present at the location where a Textline or Textflow Block is positioned, and can automatically fill in the corresponding properties of the Block:

```
fontname, fontsize, fillcolor, charspacing, horizscaling, wordspacing,
textrendering, textrise
```

Since automatic detection of font properties can result in undesired behavior if the background shall be ignored, it can be activated or deactivated using *PDFlib Blocks, Detect underlying font and color*. By default this feature is turned off.

Locking Blocks. Blocks can be locked to protect them against accidentally moving, resizing, or deleting. With the Block tool active, select the Block and choose *Lock* from its context menu. While a Block is locked you cannot move, resize, or delete it, nor edit its properties.

13.3.2 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 13.2). As detailed in Section 13.7, »Block Properties«, page 382, there are several groups of properties available, subject to the Block type. The *Apply* button will only be enabled if you changed one or more properties in the dialog. The *Apply* button will be inactive for locked Blocks.

Note *Some properties may be inactive depending on the Block type and certain property settings. For example, the property subgroup* `Ruler tabs for hortabmethod=ruler` *where you can edit tabulator settings is enabled only if the* `hortabmethod` *property in the group* `Text formatting`, `Tabs` *is set to* `ruler`.

Note If you enter text for a Block property you may experience character replacements, e.g. straight quotes are replaced by smart quotes. This substitution is done by the operation system and can be disabled via »System Preferences«, »Keyboard«, »Text«, »Use smart quotes and dashes«.

To change a property's value enter the desired number or string in the property's input area (e.g. *linewidth),* choose a value from a drop-down list (e.g. *fitmethod, orientate),* or select a font, color value or file name by clicking the »...« button at the right-hand side of the dialog (e.g. *backgroundcolor, defaultimage)*. For the *fontname* property you can either choose from the list of fonts installed on the system or type a custom font name. Regardless of the method for entering a font name, the font must be available on the system where the Blocks will be filled with PPS.

Modified properties will in be displayed in bold face in the Block Properties dialog. If any of the properties in a Block has been modified, the suffix *(*)* will be appended to the displayed Block name. When you are done editing properties click the *Apply* button to update the Block. The properties just defined will be stored in the PDF file as part of the Block definition.

Stacked Blocks. Overlapping Blocks can be difficult to select since clicking an area will always select the topmost Block. In this situation the *Select Block* entry in the context menu can be used to select one of the Blocks by name. As soon as a Block has been selected this way, the next action within its area will not affect other Blocks, but only the selected one. For example, press *Enter* to edit the selected Block's properties. This way Block properties can easily be edited even for Blocks which are partially or completely covered by other Blocks.

Using and restoring repeated values of Block properties. In order to save some amount of typing and clicking, the Block tool remembers the property values which have been entered into the previous Block's properties dialog. These values will be reused when you create a new Block. Of course you can override these values with different ones at any time.

Pressing the *Reset all* button in the properties dialog resets most Block properties to their respective default values. The following items remain unmodified:

- the *Name, Type, Rect,* and *Description* properties;
- all custom properties.

Note Do not confuse the default values of predefined Block properties with the `defaulttext`, `defaultimage`, `defaultpdf`, *and* `defaultgraphics` *properties which hold placeholder data for generating previews (see »Default Block contents«, page 372).*

Editing multiple Blocks at once. Editing the properties of multiple Blocks at once is a big time saver. You can select multiple Blocks as follows:

- Activate the Block tool via the menu item *PDFlib Blocks, PDFlib Block Tool.*
- Click on the first Block to select it. The first selected Block is the master Block. Shift-click other Blocks to add them to the set of selected Blocks. Alternatively, click *Edit, Select All* to select all Blocks on the current page.
- Double-click on any of the Blocks to open the Block Properties dialog. The Block where you double-click will be the new master Block.
- Alternatively, you can click on a single Block to designate it as master Block, and then press the *Enter* key to open the Block Properties dialog.

The Properties dialog displays only the subset of properties which apply to all selected Blocks. The dialog is populated with property values taken from the master Block. Now you can apply properties to all selected Blocks as follows:

- If the checkbox *Apply all properties of the master Block* is unchecked: upon clicking *Apply* only the properties changed manually in the dialog (highlighted in black) are copied to all selected Blocks.
- If the checkbox *Apply all properties of the master Block* is checked: upon pressing *Apply* all current properties of the master Block as well as all properties changed manually in the dialog are copied to all selected Blocks. This can be used to copy all properties from a particular Block to one or more other Blocks.

The following predefined properties as well as custom properties can not be shared, i.e. they can not be edited for multiple Blocks at once:

```
Name, Description, Subtype, Type, Rect, Status
```

13.3.3 Copying Blocks between Pages and Documents

The Block Plugin offers several methods for moving and copying Blocks within the current page, the current document, or between documents:

- move or copy Blocks by dragging them with the mouse, or pasting Blocks to another page or open document
- duplicate Blocks on one or more pages of the same document using standard copy/paste operations
- export Blocks to a new file (with empty pages) or to an existing document (apply the Blocks to existing pages)
- import Blocks from another document

In order to update the page contents while maintaining Block definitions you can replace the underlying page(s) while keeping the Blocks. Use *Tools, Organize Pages, Replace* for this purpose.

Moving and copying Blocks. You can relocate Blocks or create copies of Blocks by selecting one or more Blocks and dragging them to a new location while pressing the Ctrl key (on Windows) or Alt key (on macOS). The mouse cursor will change while this key is pressed. A copied Block has the same properties as the original Block, with the exception of its name and position which will automatically be adjusted in the new Block.

You can also use copy/paste to copy Blocks to another location on the same page, to another page in the same document, or to another document which is currently open in Acrobat:

- Activate the Block tool and select the Blocks you want to copy.
- Use Ctrl-C (on Windows) or Cmd-C (on macOS) or *Edit, Copy* to copy the selected Blocks to the clipboard.
- Navigate to the target page (if necessary).
- Make sure that the Block Tool is active, and use Ctrl-V (on Windows) or Cmd-V (on macOS) or *Edit, Paste* to paste the Blocks from the clipboard to the current page and document.

Duplicating Blocks on other pages. You can create duplicates of one or more Blocks on an arbitrary number of pages in the current document simultaneously:

- Activate the Block tool and select the Blocks you want to duplicate.

- Choose *Import and Export, Duplicate...* from the *PDFlib Blocks* menu or the context menu.
- Choose which Blocks to duplicate *(Selected Blocks* or *All Blocks on this Page)* and the range of target pages to which you want to duplicate the selected Blocks.

Exporting and importing Blocks. Using the export/import feature for Blocks it is possible to share the Block definitions on a single page or all Blocks in a document among multiple PDF files. This is useful for updating the page contents while maintaining existing Block definitions. To export Block definitions to a separate file proceed as follows:

- Activate the Block tool and select the Blocks you want to export.
- Choose *Import and Export, Export...* from the *PDFlib Blocks* menu or the context menu. Enter the page range and a file name of the new PDF with the Block definitions.

You can import Block definitions via *PDFlib Blocks, Import and Export, Import...* . Upon importing Blocks you can choose whether to apply the imported Blocks to all pages in the document or only to a page range. If more than one page is selected the Block definitions will be copied unmodified to the pages. If there are more pages in the target range than in the imported Block definition file you can use the *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.

13.3.4 Customizing the Block Plugin User Interface with XML

Some aspects of the Block Plugin user interface are stored/reloaded upon each Acrobat session, and can be controlled via an XML configuration file. A sample configuration file *factory settings.xml* is included in the distribution. If the configuration has been modified the new settings are stored in *user settings.xml*. The modified configuration will be loaded every time Acrobat is started and written when Acrobat is closed. The configuration file is stored in a location similar to the following (the names of system directories may be localized; replace *DC* with another Acrobat track name as appropriate):

```
Windows: C:\Users\<user>\AppData\Local\Adobe\Acrobat\DC\PDFlib\Block Plugin 5
macOS:   /Users\<user>/Library/Application Support/Adobe/Acrobat/DC/PDFlib/Block Plugin 5
```

The following XML elements can be used to modify the configuration manually:

- The element */Block_Plugin/MainDialog/CloseOnApply* controls the initial status of the *Close dialog on apply* checkbox in the Block properties dialog. This checkbox determines whether the Block Properties dialog will be kept open after creating a Block or modifying Block properties.
- The element */Block_Plugin/MainDialog/ApplyAllProps* controls the initial status of the *Apply all properties of the mastesr Block* checkbox in the Block properties dialog. This checkbox determines whether all properties of the master Block are copied to multiple selected Blocks or only those properties which have been modified in the dialog.

- The element */Block_Plugin/FontDialog/ShowBaseFonts* controls whether the base 14 fonts will be displayed in the font list of the Block Properties dialog (property group *Appearance*, property *fontname)* in addition to the fonts installed on the system.
- The element */Block_Plugin/Command/ControlByClick* controls the initial status of the menu item *PDFlib Blocks, Click object to define Block*.
- The element */Block_Plugin/Command/DetectFonts* controls the initial status of the menu item *PDFlib Blocks, Detect underlying font and color*.
- The element */Block_Plugin/Command/KeyAccelerator* with the possible values *control* (which designates the Ctrl key on Windows and the Command key on macOS), *shift* for the Shift key, *control+shift* or *none* specifies the accelerator key for the following keyboard shortcuts:

  ```
  A (select all), C (copy), I (Block Properties dialog), V (paste), X (cut)
  ```

  The change will be effective upon the next start of Acrobat since keyboard shortcuts cannot be changed at runtime. If this entry is absent, no accelerators are available. The default is *control*.
- The element *Configuration/Preferences/PreviewStatusMessage* controls whether a status message dialog *(e.g. »10 Block(s) processed: ...«)* is shown after each Preview operation.

13.4 Converting PDF Form Fields to PDFlib Blocks

As an alternative to creating PDFlib Blocks manually, you can automatically convert PDF form fields to Blocks. This is especially convenient if you have complex PDF forms which you want to fill automatically with PPS or need to convert a large number of existing PDF forms for automated filling. In order to convert all form fields on a page to PDFlib Blocks choose *PDFlib Blocks, Convert Form Fields, Current Page*. To convert all form fields in a document choose *All Pages* instead. Finally, you can convert only selected form fields (choose Acrobat's *Select Object* Tool via *Tools, Rich Media)* to select one or more form fields) with *Selected Form Fields*.

Form field conversion details. Automatic form field conversion will convert form fields of the types selected in the *PDFlib Blocks, Convert Form Fields, Conversion Options...* dialog to Blocks of type *Textline* or *Textflow*. By default all form field types will be converted. Attributes of the converted fields will be transformed to the corresponding Block properties according to Table 13.3.

Multiple form fields with the same name. Multiple form fields on the same page are allowed to have the same name, while Block names must be unique on a page. When converting form fields to Blocks a numerical suffix will therefore be added to the name of generated Blocks in order to create unique Block names (see also »Associating form fields with corresponding Blocks«, page 369).

Note that due to a problem in Acrobat the field attributes of form fields with the same names are not reported correctly. If multiple fields have the same name, but different attributes these differences will not be reflected in the generated Blocks. The Conversion process will issue a warning in this case and provide the names of affected form fields. In this case you should carefully check the properties of the generated Blocks.

Associating form fields with corresponding Blocks. Since the form field names will be modified when converting multiple fields with the same name (e.g. radio buttons) it is difficult to reliably identify the Block which corresponds to a particular form field. This is especially important when using an FDF or XFDF file as the source for filling Blocks such that the final result resembles the filled form.

In order to solve this problem the AcroFormConversion plugin records details about the original form field as custom properties when creating the corresponding Block. Table 13.2 lists the custom properties which can be used to reliably identify the Blocks; all properties have type *string*.

Table 13.2 Custom properties for identifying the original form field corresponding to the Block

custom property	*meaning*
PDFlib:field:name	*Fully qualified name of the form field*
PDFlib:field:pagenumber	*Page number (as a string) in the original document where the form field was located*
PDFlib:field:type	*Type of the form field; one of* `pushbutton`, `checkbox`, `radiobutton`, `listbox`, `combobox`, `textfield`, `signature`
PDFlib:field:value	*(Only for* `type=checkbox`*) Export value of the form field*

Table 13.3 Conversion of PDF form fields to PDFlib Blocks

PDF form field attribute...	***...will be converted to the PDFlib Block property***
all fields	
Position	`Rect`
Name	`Name`
Tooltip	`Description`
Appearance, Text, Font	`fontname`
Appearance, Text, Font Size	`fontsize`; `auto` *font size will be converted to a fixed font size of 2/3 of the Block height, and* `fitmethod` *will be set to* `auto`. *For multi-line fields/Blocks this combination will automatically result in a suitable font size which may be smaller than the initial value of 2/3 of the Block height.*
Appearance, Text, Text Color	`strokecolor` *and* `fillcolor`
Appearance, Border, Border Color	`bordercolor`
Appearance, Border, Fill Color	`backgroundcolor`
Appearance, Border, Line Thickness	`linewidth`: *Thin=1, Medium=2, Thick=3*
General, Common Properties, Form Field	`Status`: *Visible*=`active` *Hidden*=`ignore` *Visible but doesn't print*=`ignore` *Hidden but printable*=`active`
General, Common Properties, Orientation	`orientate`: *0*=`north`, *90*=`west`, *180*=`south`, *270*=`east`
text fields	
Options, Default Value	`defaulttext`
Options, Alignment	`position`: *Left*={`left center`} *Center*={`center center`} *Right*={`right center`}
Options, Multi-line	*checked creates Textflow Block* *unchecked creates a Textline Block*
radio buttons and check boxes	
If »Check box/Button is checked by default« is selected: Options, Check Box Style or Options, Button Style	`defaulttext`: *Check=4* *Circle=l* *Cross=8* *Diamond=u* *Square=n* *Star=H* *(these characters represent the respective symbols in the ZapfDingbats font)*
list boxes and combo boxes	
Options, Selected (default) item	`defaulttext`
buttons	
Options, Icon and Label, Label	`defaulttext`

Binding Blocks to the corresponding form fields. In order to keep PDF form fields and the generated PDFlib Blocks synchronized, the generated Blocks can be bound to the corresponding form fields. This means that the plugin will internally maintain the relationship of form fields and Blocks. When the conversion process is activated again, bound Blocks will be updated to reflect the attributes of the corresponding PDF form fields. Bound Blocks are useful to avoid duplicate work: when a form is updated for interactive use, the corresponding Blocks can automatically be updated, too.

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 Acrobat's menu functions or tools until the conversion is finished.

13.5 Previewing Blocks in Acrobat

Note You can try the Preview feature with the `block_template.pdf` *document in the PDFlib distribution. The required resources (e.g. font and image) are also included in the PDFlib distribution.*

PDFlib Blocks will be processed by PPS where the Block filling process can be customized regarding the data sources (e.g. text from a database, image files on disk) as well as visual and interactive aspects of the generated documents. This process is detailed in Section 13.6, »Filling Blocks with PPS«, page 377.

However, the Block Plugin contains an integrated version of PPS which can be used to generate Preview versions of the filled Blocks interactively in Acrobat without any programming. Although this Preview feature cannot offer the same flexibility as custom programming, it provides a quick overview of Block filling results. The Block Preview can be used for improving the position and size of Blocks as well as for checking the Block properties (e.g. font name and size). You can change the Blocks and create a new Preview until you are satisfied with the results shown in the Preview. Previews can be generated for the current page or the whole document.

The Preview will always be shown in a new PDF document. The original document (which contains the Blocks) will not be modified by generating a Preview. You can save or discard the generated Preview documents according to your requirements.

Default Block contents. Since the server-side data sources (e.g. a database) for the text, image, vector graphics or PDF contents of a Block are not available in the Plugin, the Preview feature always uses the Block's default contents, i.e. the values specified in the *defaulttext*, *defaultimage*, *defaultpdf*, or *defaultgraphics* properties. Usually, a sample data set will be used as default data which is representative for the real Block contents used with PPS. Blocks without any default contents are ignored when generating the Preview, as well as Blocks with *Status=ignoredefault*.

The default properties are empty for new Blocks. Before using the Preview feature you must fill the *defaulttext*, *defaultimage*, *defaultpdf*, or *defaultgraphics* properties (depending on the Block type) in the *Default contents* property group, or supply suitable values for the options of the same name in the *Advanced PPS options...* dialog.

Note Entering default text for symbolic fonts can be a bit tricky; see »Using symbolic fonts for default text«, page 375, for details.

Generating Block Previews. You can create Block Previews with one of the following methods:

- By clicking the PDFlib Block Preview icon  which you can locate as follows in Acrobat DC: click *Tools, Advanced Editing*.
- Via the menu item *PDFlib Blocks, Preview, Generate Preview*.
- If the Block tool is active you can right-click outside of any Block to bring up a context menu with the entries *Generate Preview* and *Preview Configuration*.

The Previews will be created based on the PDF file on disk. Any changes that you may have applied in Acrobat will only be reflected in the Preview if the Block PDF has been saved to disk using *File, Save* or *File, Save As...* . You can identify modified Blocks by the asterisk after the Block name. The Preview feature can be configured to save the Block PDF automatically before creating a Preview. This way you can make sure that interactive changes will immediately be reflected in the Preview.

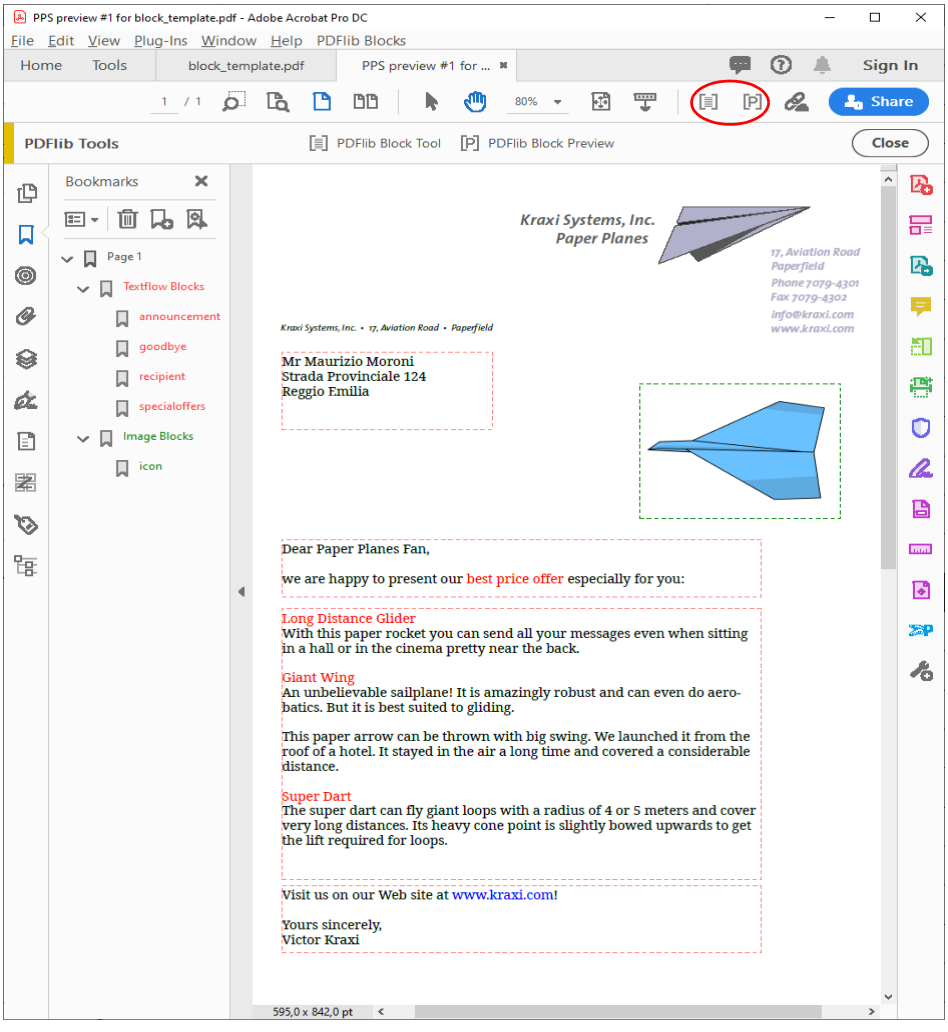


Fig. 13.3 Preview PDF for the document shown in Figure 13.1. It contains Block info layers and annotations

Configuring the Preview. Several aspects of Block Preview creation and the underlying PPS operation can be configured via *PDFlib Blocks, Preview, Preview Configuration...:*

- Preview for the current page or the full document;
- Output directory for the generated Preview documents;
- Automatically save the Block PDF before creating the Preview;
- Add Block info layers and annotations;
- Copy Blocks to the generated output;
- *Clone PDF/A, PDF/UA or PDF/X status of Block PDF:* since these standards restrict the use of layers and annotations the *Block info layers and annotations* option is mutually exclusive with this option.
- *Copy Blocks to Preview File* allows you copy the PDFlib Blocks to the generated Preview upon filling. All Blocks will be copied, regardless of whether or not they could successfully be filled.

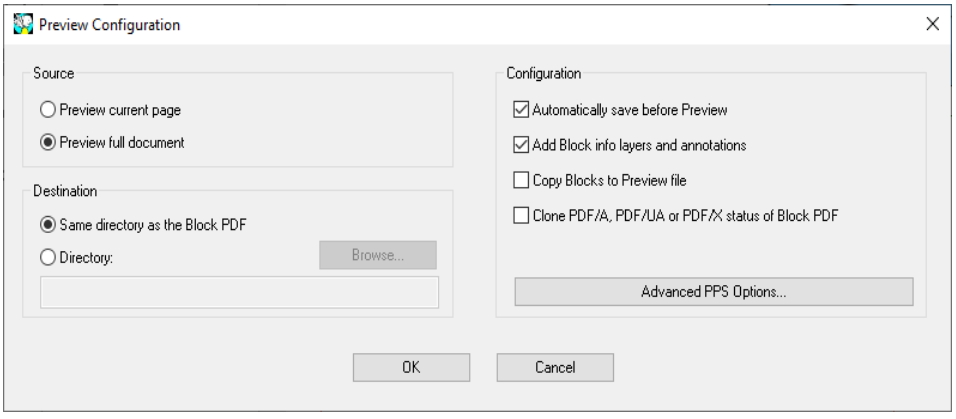


Fig. 13.4 Block Preview configuration

- The *Advanced PPS options* dialog can be used to specify additional option lists for PPS functions according to the PPS API. For example, the *searchpath* option for *PDF_set_option( )* can be used to specify a directory where fonts or images for Block filling are located. It is recommended to specify advanced options in cooperation with the developer of the PPS code.

Block ordering. When the document is stored with »Save as...« in Acrobat the Blocks are sorted in alphabetical order according to the Block name. This is also the ordering in which Blocks are processed by Preview and reported by pCOS. However, applications will typically fill Blocks based on their name (as opposed to the storage order on file), so the ordering in the PDF document is usually not relevant.

Information provided with the Preview. The generated Preview documents contain the original page contents (the background), the filled Blocks, and optionally various other pieces of information. This information can be useful for checking and improving Blocks and PPS configuration. The following items will be created for each active Block with default contents:

- *Error markers:* Blocks which could not be filled successfully are visualized by a crossed-out rectangle so that they can easily be identified. Error markers will always be created if a Block couldn't be processed.
- *Bookmarks:* The processed Blocks will be summarized in bookmarks which are structured according to the page number, the Block type, and possible errors. Bookmarks can be displayed via *View, Show/Hide, Navigation Panes, Bookmarks* (Acrobat X/XI/DC). Bookmarks will always be created.
- *Annotations:* For each processed Block an annotation will be created on the page in addition to the actual Block contents. The annotation rectangle visualizes the original Block boundary (depending on the default contents and filling mode this may differ from the boundary of the Block contents). The annotation contains the name of the Block and an error message if the Block couldn't be filled. Annotations are generated by default, but can be disabled in the Preview configuration. Since the use of annotations is restricted in the PDF/A and PDF/X standards, annotations are not created if the *Clone PDF/A, PDF/UA or PDF/X status of Block PDF* option is enabled.

- *Layers:* The page contents will be placed on layers to facilitate analysis and debugging. A separate layer will be created for the page background (i.e. the contents of the original page), each Block type, error Blocks which couldn't be filled, and the annotations with Block information. If a layer remains empty (e.g. no errors occurred) it will not be created. The layer list can be displayed via *View, Navigation Panels, Layers*. By default, all layers on the page will be displayed. In order to hide the contents of a layer click on the eye symbol to the left of the layer name. Layer creation can be disabled in the Preview configuration. Since the use of layers is restricted in the standards PDF/A-1 and PDF/X-3, layers are not created if the *Clone PDF/A, PDF/UA or PDF/X status of Block PDF* option is enabled.

Cloning PDF/A, PDF/UA or PDF/X status. The *Clone PDF/A , PDF/UA or PDF/X status of Block PDF* configuration is useful when PDF output according to one of these standards must be created. Clone mode can be enabled if the input conforms to one of the following standards:

```
PDF/A-1a:2005, PDF/A-1b:2005
PDF/A-2a, PDF/A-2b, PDF/A-2u
PDF/A-3a, PDF/A-3b, PDF/A-3u

PDF/UA-1

PDF/X-3:2003
PDF/X-4, PDF/X-4p
PDF/X-5n
```

When Previews are created in clone mode, PPS duplicates the following aspects of the Block PDF in the generated Preview:

- the PDF standard identification;
- output intent condition;
- page sizes including all page boxes;
- Tagged PDF: document language (if present);
- XMP document metadata.

When cloning standard-conforming PDF documents all Block filling operations must conform to the respective standard. For example, if no output intent is present RGB images without ICC profile can not be used. Similarly, all used fonts must be embedded. The full list of requirements can be found in Section 12.3, »PDF/A for Archiving«, page 319, and Section 12.4, »PDF/X for Print Production«, page 331. If a Block filling operation in PDF/A or PDF/X cloning mode would violate the selected standard (e.g. because a default image uses RGB color space, but the document does not contain a suitable output intent) an error message pops up and no Preview will be generated. This way users can catch potential standard violations very early in the workflow.

Using symbolic fonts for default text. Two methods are available to supply default text for Blocks with symbolic fonts:

- Working with 8-bit legacy codes, e.g. as shown in the Windows character map application: supply the 8-bit codes for the *defaulttext* either by entering the corresponding 8-bit character literally (e.g. by copy/pasting from the Windows character map) or as a numerical escape sequence. In this case you must keep the default value of the *charref* property in the *Text preparation* property group as *false* and can not work

with character references. For example, the following default text will produce the »smiley« glyph from the symbolic Wingdings font if *charref=false:*

```
J
\x4A
\112
```

- Working with the Unicode values or glyph names used in the font: set the *charref* property in the *Text preparation* property group to *true* and supply character references or glyph name references for the symbols (see Section 5.6.2, »Character References«, page 119). For example, the following default text will produce the »smiley« glyph from the symbolic Wingdings font if *charref=true:*

```
&#xF04A;
&.smileface;
```

Keep in mind that with both methods an alternate representation will be visible instead of the actual symbolic glyphs in the Block properties dialog.

13.6 Filling Blocks with PPS

In order to fill Blocks with PPS you must first place the page containing the Blocks on the output page with the *PDF_fit_pdi_page()* function. After placing the page its Blocks can be filled with the *PDF_fill_*block()* functions.

Simple example: add variable text to a template. Adding dynamic text to a PDF template is a very common task. The following code fragment opens a page in an input PDF document (the template or Block container), places it on the output page, and fills some variable text into a text Block called *firstname:*

```
doc = p.open_pdi_document(filename, "");
if (doc == -1)
        throw new Exception("Error: " + p.get_errmsg());

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

p.begin_page_ext(width, height, "");
/* Place the imported page */
p.fit_pdi_page(page, 0.0, 0.0, "");

/* Fill a single Block on the placed page */
p.fill_textblock(page, "firstname", "Serge", "encoding=winansi");

p.close_pdi_page(page);
p.end_page_ext("");
p.close_pdi_document(doc);
```

Cookbook *A full code sample can be found in the Cookbook topic* `blocks/starter_block`.

Overriding Block properties. In certain situations the programmer wants to use only some of the properties provided in a Block definition, but override other properties with custom values. This can be useful in various situations:

- Business logic may decide to enforce certain overrides.
- The scaling factor for an image or PDF page will be calculated by the application instead of taken from the Block definition.
- Change the Block coordinates programmatically, for example when generating an invoice with a variable number of data items.
- Individual spot color names could be supplied in order to match customer requirements in a print shop application.

Property overrides can be achieved by supplying property names and the corresponding values in the option list of the *PDF_fill_*block()* functions, e.g.

```
p.fill_textblock(page, "firstname", "Serge", "fontsize=12");
```

This will override the Block's internal *fontsize* property with the supplied value 12. Almost all property names can be used as options.

Property overrides apply only to the respective function calls; they will not be stored in the Block definition.

Moving Textflow Blocks while filling. The fixed size of a Textflow Block may not match its varying textual contents. If there is only few text a gap between two Blocks may arise; if there is too much text it may not fit into the Block rectangle. In this situation you can query the results of Textflow fitting to adjust the position of the next Block:

- The default *fitmethod* is *auto*, i.e. the text is forced to fit into the Block rectangle. To allow excess text to overflow the Block you must set the *fitmethod* to *nofit*. This can be specified in the Block properties in the Block template at design time or by supplying the *fitmethod* option to *PDF_fill_textblock( )*.
- Supply the dummy option *textflowhandle=-1* (in PHP: *textflowhandle=0)* to *PDF_fill_textblock( )* so that this method returns a Textflow handle for the contents of the Block.
- The returned Textflow handle is supplied to *PDF_info_textflow( )* to query the end position of the text using the keyword *textendy*.
- Query the Block's lower vertical position with *PDF_pcos_get_number( )* and the pCOS path *pages[..]/blocks/<blockname>/Rect[1]*.
- Calculate the difference between both values. If this offset is positive the Textflow didn't completely fill the Block; if it is negative the Textflow overflowed the Block. In both cases you can move the next Block up or down by this offset. This can be achieved with the *refpoint* option of *PDF_fill_textblock( )* which overrides the *Rect* property. Since this option requires absolute coordinates you must query the vertical position of the Block (see previous step) and supply the sum of the original position and the offset to the *refpoint* option.
- You can apply this method to an arbitrary number of Textflow Blocks by accumulating the per-Block offsets. Depending on the contents each Block will move successive Blocks up or down by an appropriate amount.

Cookbook A full code sample can be found in the `starter_block` *sample.*

Placing the imported page on top of the filled Blocks. The imported page must have been placed on the output page before using any of the Block filling functions. This means that the original page will usually be placed below the Block contents. However, in some situations it may be desirable to place the original page on top of the filled Blocks. This can be achieved by placing the page once with the *blind* option of *PDF_fit_pdi_page( )* in order to make its Blocks and their position known to PPS, and place it again after filling the Blocks in order to actually show the page contents:

```
/* Place the page in blind mode to prepare the Blocks, without the page being visible */
p.fit_pdi_page(page, 0.0, 0.0, "blind");

/* Fill the Blocks */
p.fill_textblock(page, "firstname", "Serge", "encoding=winansi");
/* ... fill more Blocks ... */

/* Place the page again, this time visible */
p.fit_pdi_page(page, 0.0, 0.0, "");
```

Cookbook A full code sample can be found in the Cookbook topic `blocks/block_below_contents`.

Ignoring the container page when filling Blocks. Imported Blocks can also be useful as placeholders without any reference to the underlying contents of the Block's page. You

can place a container page with Blocks in blind mode on one or more pages, i.e. with the *blind* option of *PDF_fit_pdi_page( )*, and subsequently fill its Blocks. This way you can take advantage of the Block and its properties without placing the container page on the output page, and can duplicate Blocks on multiple pages (or even on the same output page).

Cookbook *A full code sample can be found in the Cookbook topic* `blocks/duplicate_block`.

Linking Textflow Blocks. Textflow Blocks can be linked so that one Block holds the overflow text of a previous Block. For example, if you have long variable text which may need to be continued on another page you can link two Blocks and fill the remaining text of the first Block into the second Block.

PPS internally creates a Textflow from the text provided to *PDF_fill_textblock( )* and the Block properties. For unlinked Blocks this Textflow is placed in the Block and the corresponding Textflow handle is deleted at the end of the call; overflow text is lost in this case.

With linked Textflow Blocks the overflow text of the first Block can be filled into the next Block. The remainder of the first Textflow is used as Block contents instead of creating a new Textflow. Linking Textflow Blocks works as follows:

- In the first call to *PDF_fill_textblock( )* within a chain of linked Textflow Blocks the value -1 (in PHP: 0) must be supplied for the *textflowhandle* option. The Textflow handle created internally is returned by *PDF_fill_textblock( )*, and must be stored by the application.
- In the next call to *PDF_fill_textblock( )* the Textflow handle returned in the previous step can be supplied to the *textflowhandle* option (the text supplied in the *text* parameter is ignored in this case, and should be empty). The Block is filled with the remainder of the Textflow.
- This process can be repeated with more Textflow Blocks.
- The returned Textflow handle can be supplied to *PDF_info_textflow( )* in order to determine the results of Block filling, e.g. the end condition or the end position of the text.

Note that the *fitmethod* property should be set to *clip* (this is the default anyway if *textflowhandle* is supplied). The basic code fragment for linking Textflow Blocks looks as follows:

```
p.fit_pdi_page(page, 0.0, 0.0, "");
tf = -1;

for (i = 0; i < blockcount; i++)
{
        String optlist = "encoding=winansi textflowhandle=" + tf;
        int reason;
        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_string(reason, "");
```

```
            /* end loop if all text was placed */
            if (result.equals("_stop"))
            {
                p.delete_textflow(tf);
                break;
            }
        }
```

Cookbook *A full code sample can be found in the Cookbook topic* `blocks/linked_textblocks`.

Block filling order. The Block functions *PDF_fill_*block()* process properties and Block contents in the following order:

- Background: if the *backgroundcolor* property is present and contains a color space keyword different from *None*, the Block area will be filled with the specified color.
- Border: if the *bordercolor* property is present and contains a color space keyword different from *None*, the Block border will be stroked with the specified color and linewidth.
- Contents: the supplied Block contents and all other properties except *bordercolor* and *linewidth* will be processed.
- Textline and Textflow Blocks: if neither text nor default text has been supplied, there won't be any output at all, not even background color or Block border.

Nested Blocks. Before Blocks can be filled the page containing the Blocks must have been placed on the output page before (since otherwise PPS wouldn't know the location of the Blocks after scaling, rotating, and translating the page). If the page only serves as a Block container without bringing static content to the new page you can place the imported page with the *blind* option.

For successful Block filling it doesn't matter how the imported page was placed on the output page:

- The page can be placed directly with *PDF_fit_pdi_page()*.
- The page can be placed indirectly in a table cell with *PDF_fit_table()*.
- The page can be placed as contents of a another PDF Block with *PDF_fill_pdfblock()*.

The third method, i.e. filling a PDF Block with another page containing Blocks, allows nested Block containers. This allows simple implementations of interesting use cases. For example, you can implement both imposition and personalization with a two-step Block filling process:

- The first-level Block container page contains several large PDF Blocks which indicate the major areas on the paper to be printed on. The arrangement of PDF Blocks reflects the intended post-processing of the paper (e.g. folding or cutting).
- Each of the first-level PDF Blocks is then filled with a second-level container PDF page which contains Text, Image, PDF, or Graphics Blocks to be filled with variable text for personalization.

With this method Block containers can be nested. Although Block nesting works to an arbitrary level, a nesting level of three or more will only rarely be required.

The second-level Block containers (e.g. a template page for a letter) may be identical or different for each imposed page. If they are identical the Blocks on the letter template must be filled before placing the letter template itself in the next first-level Block since PPS always uses the location of the most recent placement of the template page.

Cookbook A full code sample can be found in the Cookbook topic `blocks/nested_blocks`.

Block coordinates. The Rectangle coordinates of a Block refer to the PDF default coordinate system. When the page containing the Block is placed on the output page with PPS, several positioning and scaling options can be supplied to *PDF_fit_pdi_page( )*. These options are taken into account when the Block is being processed. This makes it possible to place a template page on the output page multiply, every time filling its Blocks with data. For example, a business card template may be placed four times on an imposition sheet. The Block functions will take care of the coordinate system transformations, and correctly place the text for all Blocks in all invocations of the page. The only requirement is that the client must place the page and then process all Blocks on the placed page. Then the page can be placed again at a different location on the output page, followed by more Block processing operations referring to the new position, and so on.

The Block Plugin displays the Block coordinates differently from what is stored in the PDF file. The plugin uses Acrobat's convention which has the coordinate origin in the upper left corner of the page, while the internal coordinates (those stored in the Block) use PDF's convention of having the origin at the lower left corner of the page. The coordinate display in the Properties dialog is also subject to the units specified in Acrobat (see »Block size and position«, page 362).

Spot colors in Block properties. To use a separation (spot) color in a Block property you can click the »...« button which will present a list of all HKS and Pantone spot colors. These color names are built into PPS and can be used without further preparations. For custom spot colors an alternate color can be defined in the Block Plugin. If no alternate color is specified in the Block properties, the custom spot color must have been defined earlier in the PPS application using *PDF_makespotcolor( )* or a suitable color option list. Otherwise Block filling will fail.

13.7 Block Properties

PPS and the Block Plugin support general properties which can be assigned to any type of Block. In addition there are properties which are specific to the Block types *Textline*, *Textflow*, *Image*, *PDF*, and *Graphics*.

Properties support the same data types as option lists except handles and action lists. The names of Block properties are generally identical to options for API functions such as *PDF_fit_textline( )*, *PDF_fit_image( )* (e.g., *fitmethod*, *charspacing*). In these cases the behavior is exactly the same as the one documented for the respective option.

13.7.1 Administrative Properties

Administrative properties apply to all Block types. Required entries will automatically be generated by the Block Plugin. Table 13.4 lists the administrative Block properties.

Table 13.4 Administrative properties

keyword	***possible values and explanation***
Description	*(String) Human-readable description of the Block's function, coded in PDFDocEncoding or Unicode (in the latter case starting with a BOM). This property is for user information only, and will be ignored by PPS.*
Locked	*(Boolean) If* `true`*, the Block and its properties can not be edited with the Block Plugin. This property will be ignored by PPS. Default:* `false`
Name	*(String; required) Name of the Block. Block names must be unique within a page, but not within a document. The three characters* `[ ] /` *are not allowed in Block names. Block names are restricted to a maximum of 125 characters.*
Subtype	*(Keyword; required) Depending on the Block type, one of* `Text`, `Image`, `PDF`, *or* `Graphics`. *Note that Textline and Textflow Blocks both have Subtype* `Text`*, but are distinguished by the* `textflow` *property.*
textflow	*(Boolean) Controls single- or multiline processing. This property is not available explicitly in the user interface of the Block Plugin, but will be mapped to Textline or Textflow Blocks, respectively (Default:* `false`*):* ***false*** *Textline Block: text spans a single line and will be processed with PDF_fit_textline( ).* ***true*** *Textflow Block: text can span multiple lines and will be processed with PDF_fit_textflow( ). In addition to the standard text properties Textflow-related properties can be specified (see Table 13.9).*
Type	*(Keyword; required) Always* `Block`

13.7.2 Rectangle Properties

Rectangle properties apply to all Block types. They describe the appearance of the Block rectangle itself. Required entries will automatically be generated by the Block Plugin. Table 13.5 lists the rectangle properties.

Table 13.5 Rectangle properties

keyword	***possible values and explanation***
background-color	*(Color) If this property is present and contains a color space keyword different from* None*, a rectangle will be drawn and filled with the supplied color. This may be useful to cover existing page contents. Default:* None
bordercolor	*(Color) If this property is present and contains a color space keyword different from* None*, a rectangle will be drawn and stroked with the supplied color. Default:* None
linewidth	*(Float; must be greater than 0) Stroke width of the line used to draw the Block rectangle; only used if* `bordercolor` *is set. Default: 1*
Rect	*(Rectangle; required) The Block coordinates. The origin of the coordinate system is in the lower left corner of the page. However, the Block Plugin displays the coordinates in Acrobat's notation, i.e., with the origin in the upper left corner of the page. The coordinates will be displayed in the unit which is currently selected in Acrobat, but will always be stored in points in the PDF file.*
Status	*(Keyword) Describes how the Block will be processed by PPS and the Preview feature (default:* `active`*):* ***active*** *The Block will be fully processed according to its properties.* ***ignore*** *The Block will be ignored.* ***ignoredefault*** *Like* `active`*, except that the* `defaulttext/image/pdf/graphics` *properties and options are ignored, i.e. the Block remains empty if no variable contents are available (especially in the Preview). This may be useful to make sure that the Block's default contents are not used for filling Blocks on the server side although the Block may contain default contents for generating Previews. It can also be used to disable the default contents for previewing a Block without removing the default contents from the Block properties.* ***static*** *No variable contents will be placed; instead, the Block's default text, image, PDF, or graphics contents will be used if available.*

13.7.3 Appearance Properties

Appearance properties specify formatting details:

- Table 13.6 lists transparency appearance properties for all Block types.
- Table 13.7 lists text appearance properties for Textline and Textflow Blocks.

Table 13.6 Transparency appearance properties for all Block types

keyword	***possible values and explanation***
blendmode	*(Keyword list; if used in PDF/A-1 mode it must have the value* `Normal`*) Name of the blend mode:* `None`, `Color`, `ColorDodge`, `ColorBurn`, `Darken`, `Difference`, `Exclusion`, `HardLight`, `Hue`, `Lighten`, `Luminosity`, `Multiply`, `None`, `Normal`, `Overlay`, `Saturation`, `Screen`, `SoftLight`. *Default:* `None`
opacityfill	*(Float; if used in PDF/A mode it must have the value 1) Opacity for fill operations in the range 0..1. The value 0 means fully transparent; 1 means fully opaque.*
opacitystroke	*(Float; if used in PDF/A mode it must have the value 1) Opacity for stroke operations in the range 0..1. The value 0 means fully transparent; 1 means fully opaque.*

Table 13.7 Text appearance properties for Textline and Textflow Blocks

keyword	possible values and explanation
charspacing	*(Float or percentage) Character spacing. Percentages are based on fontsize. Default: 0*
decoration-above	*(Boolean) If* `true`*, the text decoration enabled with the* `underline`*,* `strikeout`*, and* `overline` *options will be drawn above the text, otherwise below the text. Changing the drawing order affects visibility of the decoration lines. Default:* `false`
fillcolor	*(Color) Fill color of the text. Default:* `gray 0` *(=black)*
fontname[1]	*(String) Name of the font as required by PDF_load_font( ). The Block plugin will present a list of fonts available in the system. However, these font names may not be portable between macOS, Windows, and Unix systems. If* `fontname` *starts with an '@' character the font will be applied in vertical writing mode. The encoding for the text must be specified as an option for PDF_fill_textblock( ) when filling the Block unless the* `font` *option has been supplied.*
fontsize[1]	*(Float) Size of the font in points*
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`
overline	*(Boolean) Overline mode. Default:* `false`
shadow	*(Composite) Create a shadow effect (default: no shadow). The following subproperties are available:* ***fillcolor*** *(Color) Color of the shadow. Default:* {`gray 0.8`} ***offset*** *(List of 2 floats or percentages) The shadow's offset from the reference point of the text in user coordinates or as a percentage of the font size. Default:* {`5% -5%`}
strikeout	*(Boolean) Strikeout mode. Default:* `false`
strokecolor	*(Color) Stroke color of the text. Default:* `gray 0` *(=black)*
strokewidth	*(Float, percentage, or keyword; only effective if* `textrendering` *is set to stroke text) Line width for outline text (in user coordinates or as a percentage of the* `fontsize`*). The keyword* `auto` *or the equivalent value 0 uses a built-in default. Default:* `auto`
textrendering	*(Integer) Text rendering mode. Only the value 3 has an effect on Type 3 fonts (default: 0):* **0**  *fill text* **1**  *stroke text (outline)* **2**  *fill and stroke text* **3** *invisible text* **4**  *fill text and add it to the clipping path* **5**  *stroke text and add it to the clipping path* **6**  *fill and stroke text and add it to the clipping path* **7**  *add text to the clipping path (not for Blocks)*
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. This property is required in Textline and Textflow Blocks; it will be enforced by the Block Plugin.

13.7.4 Text Preparation Properties

Text preparation properties specify preprocessing steps for Textline and Textflow Blocks. Table 13.8 lists text preparation properties for Textline and Textflow Blocks.

Table 13.8 Text preparation properties for Textline and Textflow Blocks

keyword	possible values and explanation
charref	*(Boolean) If* true*, enable substitution of numeric and character entity references and glyph name references. Default: the global* charref *option*
escape-sequence	*(Boolean) If* true*, enable substitution of escape sequences in content strings, hypertext strings, and name strings. Default: the global* escapesequence *option*
features	*(List of keywords) Specifies which typographic features of an OpenType font will be applied to the text, subject to the* script *and* language *options. Keywords for features which are not present in the font will silently be ignored. The following keywords can be supplied:* ***_none*** *Apply none of the features in the font. As an exception, the* vert *feature must explicitly be disabled with the* novert *keyword.* ***<name>*** *Enable a feature by supplying its four-character OpenType tag name. Some common feature names are* liga*,* ital*,* tnum*,* smcp*,* swsh*,* zero*. The full list with the names and descriptions of all supported features can be found in Section 7.3.1, »Supported OpenType Layout Features«, page 164.* ***no<name>*** *The prefix* no *in front of a feature name (e.g.* noliga*) disables this feature.* *Default:* _none *for horizontal writing mode. In vertical writing mode* vert *will automatically be applied. The* readfeatures *option in PDF_load_font( ) is required for OpenType feature support.*
language	*(Keyword; only relevant if* script *is supplied) The text will be processed according to the specified language, which is relevant for the* features *and* shaping *options. A full list of keywords can be found in Section 7.4.2, »Script and Language«, page 172, e.g.* ARA *(Arabic),* JAN *(Japanese),* HIN *(Hindi). Default:* _none *(undefined language)*
script	*(Keyword; required if* shaping=true*) The text will be processed according to the specified script, which is relevant for the* features*,* shaping*, and* advancedlinebreaking *options. The most common keywords for scripts are the following:* _none *(undefined script),* latn*,* grek*,* cyrl*,* armn*,* hebr*,* arab*,* deva*,* beng*,* guru*,* gujr*,* orya*,* taml*,* thai*,* laoo*,* tibt*,* hang*,* kana*,* han*. The keyword* _auto *selects the script to which the majority of characters in the text belong, where* _latn *and* _none *are ignored. A full list of keywords can be found in Section 7.4.2, »Script and Language«, page 172. Default:* _none
shaping	*(Boolean) If* true*, the text will be formatted (shaped) according to the* script *and* language *options. The* script *option must have a value different from* _none *and the required shaping tables must be available in the font. Default:* false

13.7.5 Text Formatting Properties

Table 13.9 lists properties which can only be used for Textflow Blocks, with the exception of the *stamp* property which can also be used for Textline Blocks. They will be used to construct the initial option list for processing the Textflow (corresponding to the *optlist* parameter of *PDF_create_textflow( )*). Inline option lists for Textflows can not be specified with the plugin, but they can be supplied on the server as part of the text contents when filling the Block with *PDF_fill_textblock( )*, or in the Block's *defaulttext* property.

Table 13.9 Text formatting properties (mostly for Textflow Blocks)

keyword	possible values and explanation
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.*
advancedlinebreak	*(Boolean) Enable the advanced line breaking algorithm which is required for complex scripts. This is required for linebreaking in scripts which do not use space characters for designating word boundaries, e.g. Thai. The options* `locale` *and* `script` *will be honored. Default:* `false`
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*
avoidemptybegin	*(Boolean) If* `true`*, empty lines at the beginning of a fitbox will be deleted. Default:* `false`
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`
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.*
hortabsize	*(Float or percentage) Width of a horizontal tab[1]. The interpretation depends on the* `hortabmethod` *option. Default: 7.5%*

Table 13.9 Text formatting properties (mostly for Textflow Blocks)

keyword	possible values and explanation
lastalignment	*(Keyword) Formatting for the last line in a paragraph. All keywords of the alignment option are supported, plus the following (default:* auto*):* ***auto*** *Use the value of the alignment option unless it is justify. In the latter case* left *will be used.*
leading	*(Float or percentage) Distance between adjacent text baselines in user coordinates, or as a percentage of the font size. Default: 100%*
locale	*(Keyword) The locale which will be used for localized linebreaking methods if* advancedlinebreak=true*. The keywords consists of one or more components, where the optional components are separated by an underscore character '_' (the syntax slightly differs from NLS/POSIX locale IDs):* ► *A required two- or three-letter lowercase language code according to ISO 639-2 (see* www.loc.gov/standards/iso639-2*), e.g.* en*, (English),* de *(German),* ja *(Japanese). This differs from the* language *option.* ► *An optional four-letter script code according to ISO 15924 (see* www.unicode.org/iso15924/iso15924-codes.html*), e.g.* Hira *(Hiragana),* Hebr *(Hebrew),* Arab *(Arabic),* Thai *(Thai).* ► *An optional two-letter uppercase country code according to ISO 3166 (see* www.iso.org/iso/country_codes/iso_3166_code_lists*), e.g.* DE *(Germany),* CH *(Switzerland),* GB *(United Kingdom)* *Specifying a locale is not required for advanced line breaking: the keyword* _none *specifies that no locale-specific processing will be done. Default:* _none *Examples:* de_DE, en_US, en_GB
maxspacing **minspacing**	*(Float or percentage) The maximum or minimum distance between words (in user coordinates, or as a percentage of the width of the space character). The calculated word spacing is limited by the provided values (but the* wordspacing *option will still be added). Defaults:* minspacing=50%*,* maxspacing=500%
minlinecount	*(Integer) Minimum number of lines in the last paragraph of the fitbox. If there are fewer lines they will be placed in the next fitbox. The value 2 can be used to prevent single lines of a paragraph at the end of a fitbox (»orphans«). Default: 1*
nofitlimit	*(Float or percentage) Lower limit for the length of a line with the* nofit *method (in user coordinates or as a percentage of the width of the fitbox). Default: 75%*
parindent	*(Float or percentage) Left indent of the first line of a paragraph[1]. The amount will be added to* leftindent*. Specifying this option within a line will act like a tab. Default: 0*
rightindent **leftindent**	*(Float or percentage) Right or left indent of all text lines[1]. If* leftindent *is specified within a line and the determined position is to the left of the current text position, this option will be ignored for the current line. Default: 0*
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
shrinklimit	*(Percentage) Lower limit for compressing text with the* shrink *method; the calculated shrinking factor is limited by the provided value, but will be multiplied with the value of the* horizscaling *option. Default: 85%*
spreadlimit	*(Float or percentage) Upper limit for the distance between two characters for the* spread *method (in user coordinates or as a percentage of the font size); the calculated character distance will be added to the value of the* charspacing *option. Default: 0*
stamp	*(Keyword; Textline and Textflow Blocks) This option can be used to create a diagonal stamp within the Block rectangle. The text comprising the stamp will be as large as possible. The options* position*,* fitmethod*, and* orientate *(only* north *and* south*) will be honored when placing the stamp text in the box. Default:* none*.* ***ll2ur*** *The stamp will run diagonally from the lower left corner to the upper right corner.* ***ul2lr*** *The stamp will run diagonally from the upper left corner to the lower right corner.* ***none*** *No stamp will be created.*

Table 13.9 Text formatting properties (mostly for Textflow Blocks)

keyword	***possible values and explanation***
tabalignchar	*(Unichar) Unicode value of the character at which decimal tabs will be aligned. Default: the period character '.' (U+002E)*
tabalignment[2]	*(List of keywords) Alignment for tab stops. Each entry in the list defines the alignment for the corresponding entry in the ruler option (default:* `left`*):*
	center *Text will be centered at the tab position.*
	decimal *The first instance of* `tabalignchar` *will be left-aligned at the tab position. If no* `tabalignchar` *is found, right alignment will be used instead.*
	left *Text will be left-aligned at the tab position.*
	right *Text will be right-aligned at the tab position.*

1. In user coordinates, or as a percentage of the width of the fit box
2. Tab settings can be edited in the property subgroup `Ruler Tabs for hortabmethod=ruler` *in the Block properties dialog.*

13.7.6 Object Fitting Properties

Fitting properties are available for all Block types, although some properties are specific to a certain Block type. They control how the contents will be placed in the Block:

- Table 13.10 lists fitting properties for Textline, Image, PDF, and Graphics Blocks
- Table 13.11 lists fitting properties for Textflow Blocks (mostly related to aspects of vertical fitting).

The object fitting algorithm uses the Block rectangle as fitbox. Except for *fitmethod=clip* there will be no clipping; if you want to make sure that the Block contents do not exceed the Block rectangle avoid *fitmethod=nofit*.

Table 13.10 Fitting properties for Textline, Image, PDF, and Graphics Blocks

keyword	possible values and explanation
alignchar	*(Unichar or keyword; only for Textline Blocks) If the specified character is found in the text, its lower left corner will be aligned at the lower left corner of the Block rectangle. For horizontal text with* `orientate=north` *or* `south` *the first value supplied in the* `position` *option defines the position. For horizontal text with* `orientate=west` *or* `east` *the second value supplied in the* `position` *option defines the position. This option will be ignored if the specified alignment character is not present in the text. The value 0 and the keyword* `none` *suppress alignment characters. The specified* `fitmethod` *will be applied, although the text cannot be placed within the Block rectangle because of the forced positioning of* `alignchar`**.** *Default:* `none`
dpi	*(Float list; only for image Blocks) One or two values specifying the desired image resolution in pixels per inch in horizontal and vertical direction. With the value 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*
fitmethod	*(Keyword) Strategy to use if the supplied content doesn't fit into the Block rectangle:* `auto`, `clip`, `entire`, `meet`, `nofit` *or* `slice` *(default:* `meet`).
margin	*(Float list; only for Textline Blocks) One or two float values describing additional horizontal and vertical reduction of the Block rectangle. Default: 0*
orientate	*(Keyword) Specifies the desired orientation of the content when it is placed. Possible values are* `north`, `east`, `south`, `west`. *Default:* `north`
position	*(Float list) One or two values specifying the position of the reference point within the content. The position is specified as a percentage within the Block. Only for Textline Blocks: the keyword* `auto` *can be used for the first value in the list. It indicates* `right` *if the writing direction of the text is from right to left (e.g. for Arabic and Hebrew text), and* `left` *otherwise (e.g. for Latin text).* *Default: {0 0}, i.e. the lower left corner*
rotate	*(Float) Rotation angle in degrees by which the Block will be rotated counter-clockwise before processing begins. The reference point is center of the rotation. Default: 0*
scale	*(Float list; only for image, PDF, and Graphics Blocks) One or two values specifying the desired scaling factor(s) in horizontal and vertical direction. This option will be ignored if the* `fitmethod` *property has been supplied with one of the keywords* `auto`, `meet`, `slice`, *or* `entire`. *Default: 1*
shrinklimit	*(Float or percentage; only for Textline Blocks) The lower limit of the shrinkage factor which will be applied to fit text with* `fitmethod=auto`. *Default: 0.75*

Table 13.11 Fitting properties for Textflow Blocks

keyword	possible values and explanation
firstlinedist	*(Float, percentage, or keyword) The distance between the top of the Block rectangle and the baseline for the first line of text, specified in user coordinates, as a percentage of the relevant font size (the first font size in the line if* `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.*
fitmethod	*(Keyword) Strategy to use if the Block is too small for the Textflow:* **auto** `fontsize` *and* `leading` *are decreased until the text fits.* **clip** *Text is clipped at the Block margin (useful for linking Textflow Blocks).* **nofit** *Text runs beyond the bottom margin of the Block (useful for moving Blocks).* *Default:* `clip` *if the* `textflowhandle` *option is supplied, otherwise* `auto`
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 Block rectangle.* **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.*
linespread-limit	*(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`*.*
minfontsize	*(Float or percentage) Minimum allowed font size when text is scaled down to fit into the Block rectangle with* `fitmethod=auto` *when* `shrinklimit` *is exceeded. The limit is specified in user coordinates or as a percentage of the height of the Block. If the limit is reached the text will be created with the specified* `minfontsize` *as fontsize. Default: 0.1%*
orientate	*(Keyword) Specifies the desired orientation of the text when it is placed. Possible values are* `north`*,* `east`*,* `south`*,* `west`*. Default:* `north`
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 13.11 Fitting properties for Textflow Blocks

keyword	possible values and explanation	
verticalalign	*(Keyword) Vertical alignment of the text in the fitbox (default:* `top`*):*	
	top	*Formatting will start at the first line, and continue downwards. If the text doesn't fill the fitbox there may be whitespace below the text.*
	center	*The text will be vertically centered in the fitbox. If the text doesn't fill the fitbox there may be whitespace both above and below the text.*
	bottom	*Formatting will start at the last line, and continue upwards. If the text doesn't fill the fitbox there may be whitespace above the text.*
	justify	*The text will be aligned with top and bottom of the fitbox. In order to achieve this the leading will be increased up to the limit specified by* `linespreadlimit`*. The height of the first line will only be increased if* `firstlinedist=leading`*.*

13.7.7 Properties for default Contents

Properties for default contents specify how to fill the Block if no specific contents are provided. They are especially useful for the Preview feature since it will fill the Blocks with their default contents. Table 13.12 lists properties for default contents.

Table 13.12 Properties for default contents

keyword	possible values and explanation
default-graphics	*(String; only for graphics Blocks) Path name of a graphics file which will be used if no graphics is supplied by the client application.*[1]
defaultimage	*(String; only for image Blocks) Path name of an image which will be used if no image is supplied by the client application.*[1]
defaultpdf	*(String; only for PDF Blocks) Path name of a PDF document which will be used if no substitution PDF is supplied by the client application.*[1]
default-pdfpage	*(Integer; only for PDF Blocks) Page number of the page in the default PDF document. Default: 1*
defaulttext	*(String; only for Textline and Textflow Blocks) Text which will be used if no variable text is supplied by the client application*[2]

1. It is recommended to use file names without absolute paths, and use the `SearchPath` *feature in the PPS client application. This makes Block processing independent from platform and file system details.*
2. The text will be interpreted in `winansi` *encoding or Unicode.*

13.7.8 Custom Properties

Custom properties apply to Blocks of any type of Block, and will be ignored by PPS and the Preview feature. Table 13.13 lists the naming rules for custom properties.

Table 13.13 Custom Block properties for all Block types

keyword	possible values and explanation
any name not containing the three characters `[ ] /`	*(String, name, float, or float list) The interpretation of the values of custom properties is completely up to the client application; they will be ignored by PPS.*

13.8 Querying Block Names and Properties with pCOS

In addition to automatic Block processing with PPS, the integrated pCOS facility can be used to enumerate Block names and query standard or custom properties.

Cookbook *A full code sample for querying the properties of Blocks contained in an imported PDF can be found in the Cookbook topic* `blocks/query_block_properties`.

Finding the number and names of Blocks. The client code must not even know the names or number of Blocks on an imported page since these can also be queried. The following statement returns the number of Blocks on page with number *pagenum*:

```
blockcount = (int) p.pcos_get_number(doc, "length:pages[" + pagenum + "]/blocks");
```

The following statement returns the name of Block number *blocknum* on page *pagenum* (Block and page counting start at 0):

```
blockname = p.pcos_get_string(doc,
        "pages[" + pagenum + "]/blocks[" + blocknum + "]/Name");
```

The returned Block name can subsequently be used to query the Block's properties or fill the Block with text, image, PDF or graphics contents. If the specified Block doesn't exist an exception will be thrown. You can avoid this by using the *length* prefix to determine the number of Blocks and therefore the maximum index in the *blocks* array (keep in mind that the Block count will be one higher than the highest possible index since array indexing starts at 0).

Checking for the presence of a Block. For additional flexibility of the client application code you can check whether for the presence of a Block before attempting to fill it. This way the designer can move Blocks between pages without breaking the application which fills the Blocks.

The following code checks whether a Block with the name *foo* is present on a page:

```
/* pCOS object type "dictionary" means that the Block is present */
if (pcos_get_string(doc, "type:pages[" + pagenum + "]/blocks/" + "foo").equals("dict"))
{
        /* Block "foo" is present on the page */
}
```

Addressing Blocks by number or name. In the pCOS path syntax for addressing Block properties the following expressions are equivalent, assuming that the Block with number 6 has its *Name* property set to *foo:*

```
pages[...]/blocks[6]
pages[...]/blocks/foo
```

Querying Block coordinates. The two coordinate pairs *(llx, lly)* and *(urx, ury)* describing the lower left and upper right corner of a Block named *foo* can be queried as follows:

```
llx = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/rect[0]");
lly = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/rect[1]");
urx = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/rect[2]");
ury = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/rect[3]");
```

Note that these coordinates are provided in the default 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. The values queried with the pCOS pseudo object *rect* (all lowercase) take into account any relevant *CropBox/MediaBox* and *Rotate* entries and normalize the order of the coordinates. In contrast, the values queried with the native PDF key *Rect* cannot be directly used as new coordinates if a CropBox is present.

Note that the *topdown* option is not taken into account when querying Block coordinates.

Querying custom properties. Custom properties can be queried as in the following example, where the property *zipcode* is queried from a Block named *b1* on page *pagenum:*

```
zip = p.pcos_get_string(doc, "pages[" + pagenum + "]/blocks/b1/Custom/zipcode");
```

If you don't know which custom properties are actually present in a Block, you can determine the names at runtime. In order to find the name of the first custom property in a Block named *b1* use the following:

```
propname = p.pcos_get_string(doc, "pages[" + pagenum + "]/blocks/b1/Custom[0].key");
```

Use increasing indexes instead of 0 in order to determine the names of all custom properties. Use the *length* prefix to determine the number of custom properties.

Non-existing Block properties and default values. Use the *type* prefix to determine whether a Block or property is actually present. If the type for a path is 0 or *null* the respective object is not present in the PDF document. Note that for predefined properties this means that the default value of the property will be used.

Name space for custom properties. In order to avoid confusion when PDF documents from different sources are exchanged, it is recommended to use an Internet domain name as a company-specific prefix in all custom property names, followed by a colon ':' and the actual property name. For example, ACME corporation would use the following property names:

```
acme.com:digits
acme.com:refnumber
```

Since standard and custom properties are stored differently in the Block, standard PPS property names (as defined in Section 13.7, »Block Properties«, page 382) will never conflict with custom property names.

13.9 Creating and Importing Blocks programmatically

13.9.1 Creating PDFlib Blocks with POCA

PDFlib Blocks can be created programmatically with the POCA interface which is included in PPS. Using POCA the required PDF data structures for Block can be prepared and then supplied to the *blocks* option of *PDF_begin/end_page_ext()*. When creating the Block definitions the requirements in Section 13.10, »PDFlib Block Specification«, page 398, must be obeyed. The Block properties must be created according to the data types listed in Section 13.7, »Block Properties«, page 382.

Cookbook *A code sample for creating PDFlib Blocks with PPS can be found in the category* `blocks` *of the PDFlib Cookbook.*

The PDFlib Block specification contains an unfortunate redundancy in that the name of a Block is recorded twice: once in the main *Blocks* dictionary of a page, and again in the *Name* entry within a particular Block dictionary. These two names must be identical in order to avoid problems when filling the Block with PPS or previewing the Block with the Block Plugin. *PDF_begin/end_page_ext()* will therefore throw an exception if the dictionary provided with the *blocks* option contains a block definition which violates the »same block name« rule. The corresponding pairs are highlighted in blue in the code sample below.

The following code fragment demonstrates the use of POCA functions for creating the Block definition shown in Section , »Block dictionary keys«, page 399:

```
/* Create the Block dictionary */
blockdict = p.poca_new("containertype=dict usage=blocks");

/* ------------------------------------------------------
 * Create a Text Block
 * ------------------------------------------------------
 */
textblock = p.poca_new("containertype=dict usage=blocks type=name key=Type value=Block");

container1 = p.poca_new("containertype=array usage=blocks " +
        "type=integer values={70 640 300 700}");

p.poca_insert(textblock, "type=array key=Rect value=" + container1);
p.poca_insert(textblock, "type=name  key=Name value=job_title");
p.poca_insert(textblock, "type=name  key=Subtype value=Text");
p.poca_insert(textblock, "type=name  key=fitmethod value=auto");
p.poca_insert(textblock, "type=string  key=fontname value=Helvetica");
p.poca_insert(textblock, "type=float key=fontsize value=12");

/* Hook up the Block in the page's Block dictionary */
p.poca_insert(blockdict, "type=dict key=job_title direct=false value=" + textblock);

/* ------------------------------------------------------
 * Create an Image Block
 * ------------------------------------------------------
 */
imageblock = p.poca_new("containertype=dict usage=blocks " +
        "type=name key=Type value=Block");

container2 = p.poca_new("containertype=array usage=blocks " +
```

```
        "type=integer values={70 440 300 600}");

p.poca_insert(imageblock, "type=array key=Rect value=" + container2);
p.poca_insert(imageblock, "type=name  key=Name value=logo");
p.poca_insert(imageblock, "type=name  key=Subtype value=Image");
p.poca_insert(imageblock, "type=name  key=fitmethod value=auto");

/* Hook up the Block in the page's Block dictionary */
p.poca_insert(blockdict, "type=dict key=logo direct=false value=" + imageblock);

/* ------------------------------------------------------
 * Create a PDF Block
 * ------------------------------------------------------
 */
pdfblock = p.poca_new("containertype=dict usage=blocks " +
        "type=name key=Type value=Block");

container3 = p.poca_new("containertype=array usage=blocks " +
        "type=integer values={70 240 300 400}");

p.poca_insert(pdfblock, "type=array key=Rect value=" + container3);
p.poca_insert(pdfblock, "type=name key=Name value=pdflogo");
p.poca_insert(pdfblock, "type=name key=Subtype value=PDF");
p.poca_insert(pdfblock, "type=name key=fitmethod value=meet");

/* Hook up the Block in the page's Block dictionary */
p.poca_insert(blockdict, "type=dict key=pdflogo direct=false " + "value=" + pdfblock);


/* ------------------------------------------------------
 * Hook up the Block dictionary in the current page
 * ------------------------------------------------------
 */
p.end_page_ext("blocks=" + blockdict);


/* Clean up */
p.poca_delete(blockdict, "recursive");
```

13.9.2 Importing PDFlib Blocks

You can copy one ore more PDFlib Blocks from the input document to the current output page with *PDF_process_pdi()* and *action=copyallblocks* or *action=copyblock* as follows:

```
if (p.process_pdi(p, doc, 0, "action=copyallblocks block={pagenumber=1}") != 1)
{
        /* Error */
}
```

This way you can implement multi-level Block filling workflows. Keep in mind that Block names must be unique on each page, i.e. you cannot import multiple Blocks with the same name to the same page. Use the *outputblockname* suboption to rename Blocks upon copying.

13.10 PDFlib Block Specification

The Block syntax conforms to the PDF Reference which specifies an extension mechanism that allows applications to store private data attached to the data structures comprising a PDF page. A description of the PDFlib Block syntax is provided here. Users who create Blocks with the Block Plugin or PDFlib don't need this information.

PDF object structure for PDFlib Blocks. The page dictionary contains a *PieceInfo* entry which has another dictionary as value. The page dictionary should also contain the key *LastModified* which contains a time stamp for the creation or last modification of the Block structures. This dictionary contains the key *PDFlib* with an application data dictionary as value. The application data dictionary contains two standard keys listed in Table 13.14.

Table 13.14 Entries in a PDFlib application data dictionary

key	value
LastModified	*(Data string; required) The date and time when the Blocks on the page were created or most recently modified. This entry will be created by PDFlib when creating Blocks with the POCA interface.*
Private	*(Dictionary; required) A Block list (see Table 13.15)*

A Block list is a dictionary containing general information about Block processing, plus a list of all Blocks on the page. Table 13.15 lists the keys in a Block list dictionary.

Table 13.15 Entries in a Block list dictionary

key	value
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 13.17). The value of the* `Name` *key in the Block dictionary must be identical to the Block's name in this dictionary.*
BlockProducer[1]	*(String) Name of the software used to create the Blocks programmatically. This entry will be created by PDFlib when creating Blocks with the POCA interface.*
PluginVersion[1]	*(String) A string containing a version identification of the Block plugin used to create the Blocks.*
pdfmark[1]	*(Boolean) Must be* `true` *if the Block list has been generated by use of pdfmarks.*
Version	*(Number; required) The version number of the Block specification to which the file complies. This document describes version 10 of the Block specification.*

1. Exactly one of the keys `BlockProducer`, `PluginVersion` *and* `pdfmark` *must be present.*

Data types for Block properties. Properties support the same data types as option lists except handles and specialized lists such as action lists. Table 13.16 details how these types are mapped to PDF data types.

Table 13.16 Data types for Block properties

Data type	PDF type and remarks
boolean	*(Boolean)*
string	*(String)*

Table 13.16 Data types for Block properties

Data type	PDF type and remarks
keyword (name)	*(Name) It is an error to provide keywords outside the list of keywords supported by a particular property.*
float, integer	*(Number) While option lists support point and comma as decimal separators, PDF numbers require point.*
percentage	*(Array with two elements) The first element in the array is the number, the second element is a string containing a percent character.*
list	*(Array)*
color	*(Array with two or three elements) The first element in the array specifies a color space, and the second element specifies a color value. To specify the absence of color the respective property must be omitted. 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 PPS, 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.*
unichar	*(Text string) Unicode string in* `utf16be` *format, starting with the BOM U+FEFF*

Block dictionary keys. Block dictionaries may contain the keys in Table 13.17.

Table 13.17 Entries in a Block dictionary

property group	values
administrative properties	*(Some keys are required) Administrative properties according to Table 13.4*
rectangle properties	*(Some keys are required) Rectangle properties according to Table 13.5*
appearance properties	*(Some keys are required) Appearance properties for all Block types according to Table 13.6 and text appearance properties according to Table 13.7 for Textline and Textflow Blocks*
text preparation properties	*(Optional) Text preparation properties for Textline and Textflow Blocks according to Table 13.8*
text formatting properties	*(Optional) Text formatting properties for Textline and Textflow Blocks according to Table 13.9*
object fitting properties	*(Optional) Object fitting properties for Textline, Image, PDF, and Graphics Blocks according to Table 13.10, and fitting properties for Textflow Blocks according to Table 13.11*

Table 13.17 Entries in a Block dictionary

property group	***values***
properties for default contents	*(Optional) Properties for default contents according to Table 13.12*
Custom	*(Dictionary; optional) A dictionary containing key/value pairs for custom properties according to Table 13.13.*

A Revision History

Date	Changes
March 29, 2021	▸ Minor changes for PDFlib 9.3.1
July 14, 2020	▸ Updates for PDFlib 9.3.0
February 01, 2019	▸ Updates for PDFlib 9.2.0
February 01, 2018	▸ Updates for PDFlib 9.1.2
July 24, 2017	▸ Updates for PDFlib 9.1.1
December 15, 2016	▸ Updates for PDFlib 9.1.0
July 27, 2016	▸ Updates for PDFlib 9.0.7
November 23, 2015	▸ Updates for PDFlib 9.0.6
May 18, 2015	▸ Updates for PDFlib 9.0.5
December 16, 2014	▸ Updates for PDFlib 9.0.4
May 14, 2014	▸ Updates for PDFlib 9.0.3
December 17, 2013	▸ Updates for PDFlib 9.0.2
July 24, 2013	▸ Updates for PDFlib 9.0.1
March 12, 2013	▸ Updates for PDFlib 9.0.0
June 09, 2011	▸ Updates for PDFlib 8 VT Edition (internally 8.1.0)
December 09, 2010	▸ Various updates and corrections for PDFlib 8.0.2
September 22, 2010	▸ Various updates and corrections for PDFlib 8.0.1p7
April 13, 2010	▸ Various updates and corrections for PDFlib 8.0.1
December 07, 2009	▸ Updates for PDFlib 8.0.0
April 20, 2010	▸ Minor corrections for PDFlib 7.0.5
March 13, 2009	▸ Various updates and corrections for PDFlib 7.0.4
February 13, 2008	▸ Various updates and corrections for PDFlib 7.0.3
August 08, 2007	▸ Various updates and corrections for PDFlib 7.0.2
February 19, 2007	▸ Various updates and corrections for PDFlib 7.0.1
October 03, 2006	▸ Updates and restructuring for PDFlib 7.0.0
February 15, 2007	▸ Various updates and corrections for PDFlib 6.0.4
February 21, 2006	▸ Various updates and corrections for PDFlib 6.0.3; added Ruby section
August 09, 2005	▸ Various updates and corrections for PDFlib 6.0.2
November 17, 2004	▸ Minor updates and corrections for PDFlib 6.0.1 ▸ introduced new format for language-specific function prototypes in chapter 8 ▸ added hypertext examples in chapter 3
June 18, 2004	▸ Major changes for PDFlib 6
January 21, 2004	▸ Minor additions and corrections for PDFlib 5.0.3

Date	Changes
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

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

PDFlib GmbH

PDFlib GmbH
Franziska-Bilek-Weg 9
80339 München, Germany
www.pdflib.com
phone +49 • 89 • 452 33 84-0

Licensing contact
sales@pdflib.com

Support
support@pdflib.com *(please include your license number)*