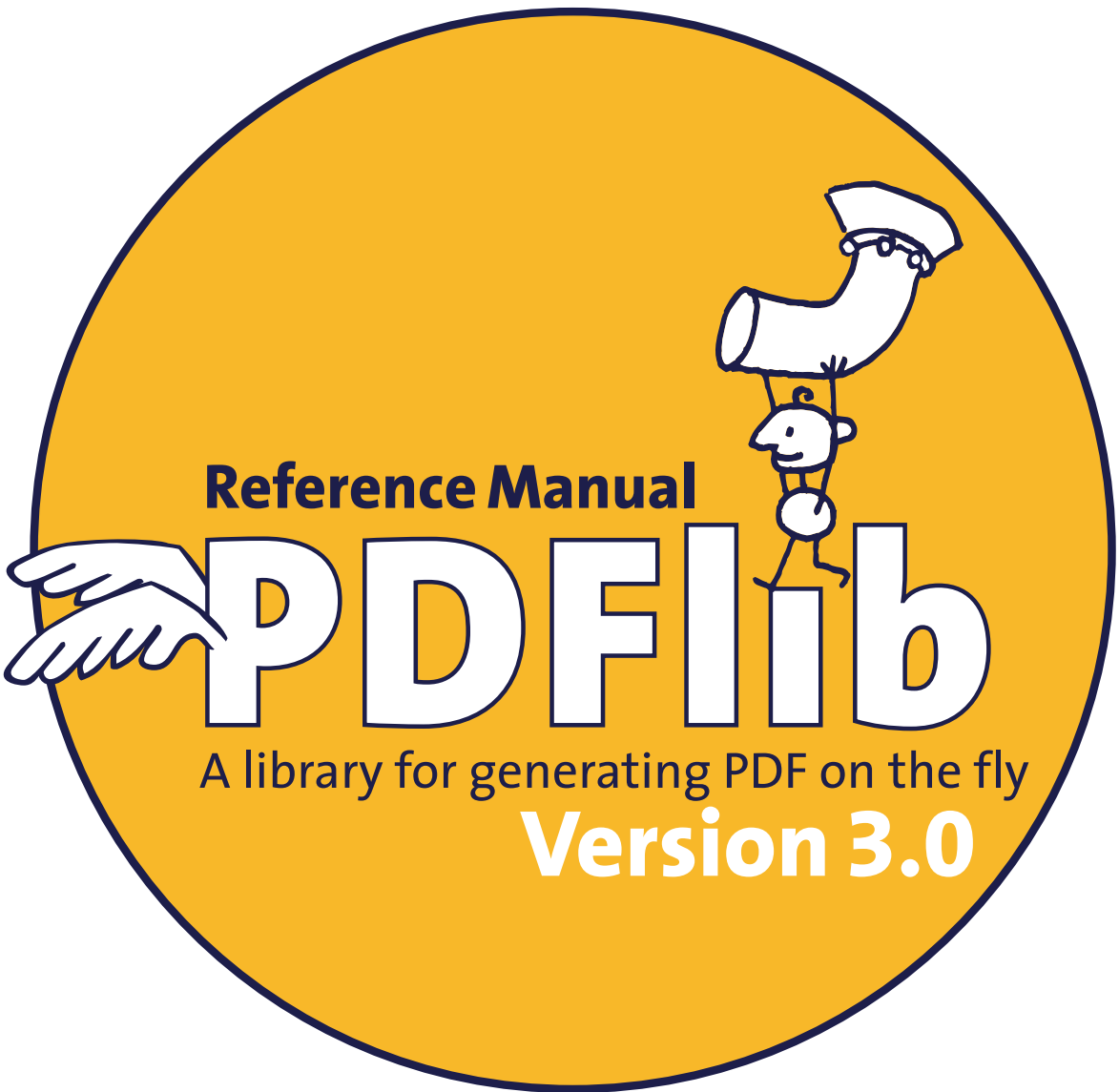
PDFlib GmbH München, Germany



www.pdflib.com

PDFlib GmbH
Tal 40, 80331 München, Germany
http://www.pdflib.com

phone +49 • 89 • 29 16 46 87
fax +49 • 89 • 29 16 46 86
Licensing contact: sales@pdflib.com
Support for PDFlib licensees: support@pdflib.com (please include your license number)
For other inquiries check the PDFlib mailing list (see http://www.pdflib.com for details)

Author: Thomas Merz (tm@pdflib.com)
Design and illustrations: Alessio Leonardi (alessio@leowol.de)
Quality control (manual): Katja Karsunke, Petra Porst (pp@pdflib.com)
Quality control (software): a cast of thousands

Revision history of this manual

Date	***Changes***
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. 01, 1999	► *Minor changes for PDFlib 1.0 (not publicly released)*
Aug. 10, 1998	► *Extensions for PDFlib 0.7 (only for a single customer)*
July 08, 1998	► *First attempt at describing PDFlib scripting support in PDFlib 0.6*
Feb. 25, 1998	► *Slightly expanded the manual to cover PDFlib 0.5*
Sept. 22, 1997	► *First public release of PDFlib 0.4 and this manual*

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

Contents

1 Introduction

1.1 PDFlib Programming

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

How can I use PDFlib? PDFlib is available on a variety of platforms, including Unix, Windows, MacOS, and EBCDIC-based systems such as IBM AS/400. Although PDFlib itself is written in the C language, its functions can be accessed from several other languages and programming environments which are called language bindings. The PDFlib language bindings cover all major Web application languages currently in use. The Application Programming Interface (API) is easy to learn, and is identical for all bindings. Currently the following bindings are supported:

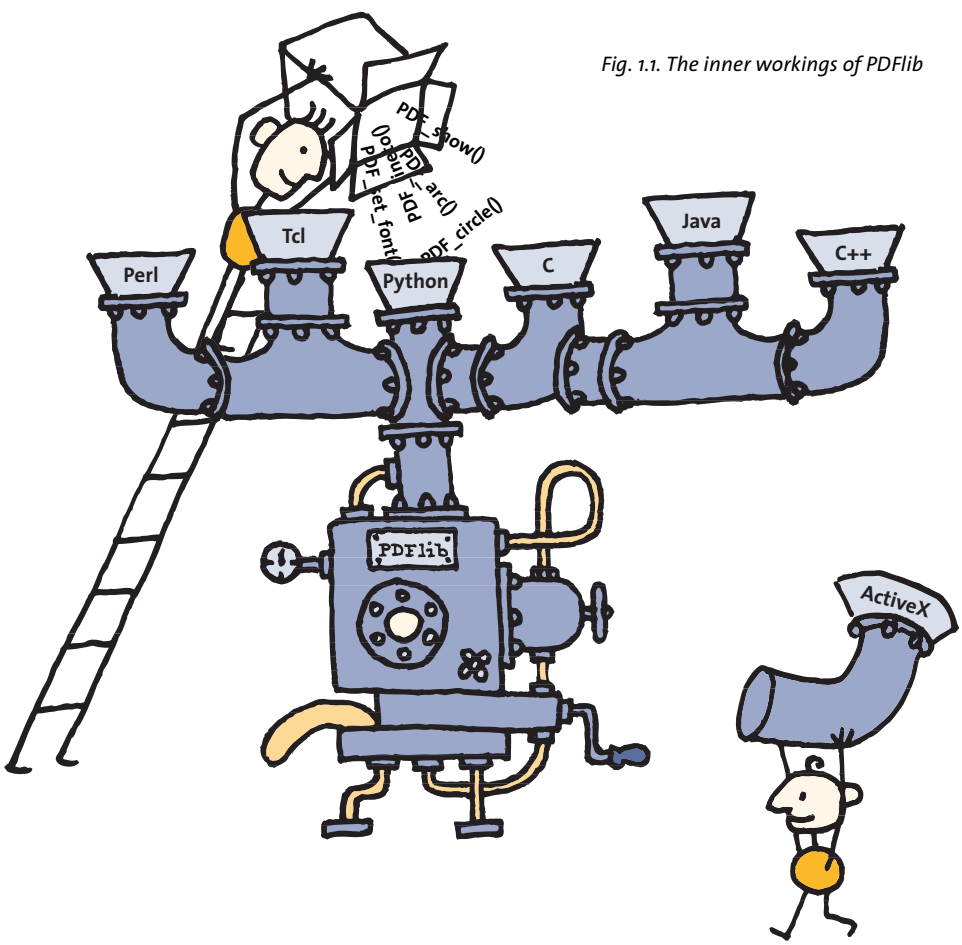


Fig. 1.1. The inner workings of PDFlib

- Active X/COM, providing access from Visual Basic, Windows Scripting Host and Active Server Pages with VBScript or JScript, as well as many other environments
- ANSI C
- ANSI C++
- Java
- Python
- Perl
- Tcl

Besides, there are a number of additional PDFlib bindings supplied by third parties, most notably PHP3.

What can I use PDFlib for? PDFlib's primary target is creating dynamic PDF on the World Wide Web. Similar to HTML pages dynamically generated on the Web server, you can use a PDFlib program for dynamically generating PDF reflecting user input or some other dynamic data, e.g. data retrieved from the Web server's database. The PDFlib approach offers several advantages with respect to creating PDF from PostScript files with Acrobat Distiller:

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

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

Requirements for using PDFlib. PDFlib makes PDF generation possible without wading through the 500+ page PDF specification. While PDFlib tries to hide technical PDF details from the user, a general understanding of PDF is useful. In order to make the best use of PDFlib, application programmers should ideally be familiar with the basic graphics model of PostScript (and therefore PDF). However, a reasonably experienced application programmer who has dealt with any graphics API for screen display or printing shouldn't have much trouble adapting to the PDFlib API as described in this manual.

About this manual. This manual describes the API implemented in PDFlib. It does not describe the process of building the library on specific platforms. The function interfaces described in this manual are believed to remain unchanged during future PDFlib development. Functions not described in this manual are unsupported, and should not be used. This manual does not attempt to explain Acrobat/PDF features or internals. Please refer to the Acrobat product literature, and the material cited at the end of this manual for further reference.

1.2 PDFlib Features

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

Table 1.1. PDFlib features for generating PDF

topic	features
PDF Documents	▸ PDF documents of arbitrary length, directly in memory (for Web servers) or on disk file ▸ Arbitrary page size–each page may have a different size ▸ Compression for text, vector graphics, image data, and file attachments ▸ Strict Acrobat 3 / PDF 1.2 mode optionally available
Vector graphics	▸ Common vector graphics primitives: lines, curves, arcs, rectangles, etc. ▸ Vector paths for stroking, filling, and clipping ▸ RGB color for stroking and filling objects
Fonts	▸ Text output in different fonts ▸ Text column formatting ▸ Underlined, overlined, and strikeout text ▸ Built-in font metrics for PDF's 14 base fonts ▸ PostScript font embedding (PFB and PFA file formats) ▸ Support for AFM and PFM font metrics files ▸ Library clients can retrieve character metrics for exact formatting ▸ Flexible font and metrics file configuration
Hypertext	▸ Page transition effects such as shades and mosaic ▸ Nested bookmarks ▸ PDF links, launch links (other document types), and Web links ▸ Document information: four standard fields (Title, Subject, Author, Keywords) plus user-defined info field (e.g., part number) ▸ File attachments and note annotations
Internationalization	▸ Unicode support (see below) ▸ Support for a variety of encodings (both built-in and user-defined) ▸ CID font and CMap support for Chinese, Japanese, and Korean text ▸ Support for the Euro character ▸ Support for international standards, e.g., ISO 8859-2
Unicode	▸ Unicode support for hypertext features: bookmarks (e.g., Greek or Russian), contents and title of text annotations, document information fields, attachment description, and author name ▸ Unicode encoding for Japanese, Chinese, and Korean text
Images	▸ Embed images in GIF, PNG, TIFF, JPEG, or CCITT file formats ▸ Images constructed by the client directly in memory ▸ Efficiently re-use image data, e.g., for repeated logos on each page ▸ Transparent (masked) images
Programming	▸ Language bindings for ActiveX, C, C++, Java, Perl, Python, Tcl ▸ Transparent Unicode handling for ActiveX, Java, and Tcl ▸ Thread-safe for deployment in multi-threaded server applications ▸ Configurable error handler and memory management for C and C++ ▸ Exception handling integrated with the host language's native exception handling

1.3 PDFlib Output and Compatibility

PDFlib output. PDFlib generates binary PDF output (although most items can also be generated in ASCII mode for debugging purposes). If the Zlib compression library is available (this is the case for all binary PDFlib distributions), the PDF output will be compressed with the Flate (also known as ZIP) compression algorithm. The compression can also be deactivated. Compression applies to potentially large items, such as raster image data and file attachments, as well as text and vector operators on page descriptions. The compression speed/output size trade-off can be controlled with a PDFlib parameter.

Acrobat 4 features. Generally, we strive to produce PDF documents which may be used with a wide variety of PDF consumers. PDFlib generates output compatible with Acrobat 3 and higher.

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

Table 1.2. PDFlib features which require Acrobat 4

topic	*remarks*
hypertext	► *File attachments are not recognized in Acrobat 3 (require full Acrobat 4)* ► *Different icons for notes are not recognized in Acrobat 3*
page size	► *Acrobat 4 extends the limits for acceptable PDF page sizes*
Unicode	► *Unicode hypertext doesn't work in Acrobat 3*
font	► *The Euro symbol is not supported in Acrobat 3* ► *CID fonts for Chinese, Japanese, and Korean require Acrobat 3J or Acrobat 4*
transparency	► *Transparency information is ignored in Acrobat 3*
JPEG images	► *Acrobat 3 supports only baseline JPEG images, but not the progressive flavor*
external images	► *Acrobat 4 (but not the free Acrobat Reader) support external image references via URL. Acrobat 3 (Reader and Exchange) is unable to display such referenced images.*

Acrobat 3 compatibility mode. Basically, if you don't use the above-mentioned Acrobat 4 features, the generated PDF files will be compatible to Acrobat 3 and 4. However, due to a very subtle compatibility issue with certain output devices, PDFlib also offers a strict Acrobat 3 compatibility mode. In order to understand the problem, we must distinguish between the actual Acrobat viewer version required by a certain PDF file, and the very first line in the file which may read *%PDF-1.2* or *%PDF-1.3* for Acrobat 3 and Acrobat 4-generated files, respectively. It's important to know that Acrobat 3 viewers open files starting with the *%PDF-1.3* line without any problem, provided the file doesn't use any Acrobat 4 feature. This is the basis of PDFlib's dual-version compatibility approach.

However, some PDF consumers other than Acrobat implement a much stricter way of version control: they simply reject all files starting with the *%PDF-1.3* line, regardless of whether the actual content requires a PDF 1.2 or PDF 1.3 interpreter. For example, some EfI RIPs for high-speed digital printing machines are known to (mis-)behave in this manner. In order to work around this problem, PDFlib offers a strict Acrobat 3 compatibility mode in which a *%PDF-1.3* header is emitted, and Acrobat 4 features are disabled.

Note again that it is not necessary to use PDFlib's strict Acrobat 3 compatibility mode only to make sure the PDF files can be read with Acrobat 3 – this will automatically be

the case if you refrain from using the above-mentioned Acrobat 4 features. The strict mode is only required for those rare situations where you have to deal with one of those broken PDF-enabled RIPs.

1.4 Features not implemented in PDFlib

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

Table 1.3. Features which are currently not implemented in PDFlib

topic	***remarks***
dealing with existing PDFs	*PDFlib generates new PDF documents, but doesn't integrate or manipulate existing PDF content.*
encryption	*Encryption requires all page contents to be cryptographically processed.*
thumbnails	*Thumbnails require a rasterizer for the page contents.*
linearization	*Linearization (Web optimization) requires a complex rewrite of the PDF file.*
EPS embedding	*Embedding EPS graphics requires a PostScript interpreter.*
font subsetting	*Font subsetting requires extended font processing.*
TrueType fonts	*Embedding TrueType fonts in PDF is a delicate and error-prone process. PDFlib currently doesn't support any kind of TrueType embedding.*

2 PDFlib Language Bindings

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

2.1 Overview of the PDFlib Language Bindings

2.1.1 What's all the Fuss about Language Bindings?

While the C programming language has been one of the cornerstones of systems and applications software development for decades, a whole slew of other languages have been around for quite some time which are either related to new programming paradigms (such as C++), open the door to powerful platform-independent scripting capabilities (such as Perl, Tcl, and Python), promise a new quality in software portability (such as Java), or provide the glue among many different technologies while being completely operating system specific (such as ActiveX/COM).

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

This goal gives rise to quite a new issue in software portability. Instead of porting a given program in a given language to many different platforms, we are trying to maintain a coherent programming interface across many different languages! Keeping this in mind is very important when dealing with the PDFlib API. The goal of multi-language portability also explains some properties of the interface which may be considered quirky in a pure C environment. In case you wonder about a particular interface feature, or some »obvious« enhancement to the interface comes to your mind, please take into a account the compatibility of your proposed enhancement to all supported environments.

Naturally, the question arises how to support so many languages with a single library. Fortunately, all modern language environments are extensible in some way or another. This includes support for extensions written in the C language in all cases. Looking closer, each environment has its own restrictions and requirements regarding the implementation of extensions. The facilities provided for extension developers are numerous, and differ significantly among the languages. Given the amount of changes occurring in actively developed software, and the number of supported languages, this may quickly result in a maintenance nightmare, especially when we take into account the number of supported platforms.

Fortunately enough, the task of writing language wrappers has been facilitated by a cute program called SWIG[1] (Simplified Wrapper and Interface Generator), written by Dave Beazley. SWIG is brilliant in design and implementation. With the help of SWIG,

early PDFlib versions could easily be integrated into the Perl, Tcl, and Python scripting languages, and even Java with a hacked-up version of SWIG. However, over time the requirements for the PDFlib language wrappers grew and grew, until finally it was necessary to manually fine-tune or partially rewrite the SWIG-generated wrapper code. For this reason the language wrappers are no longer generated automatically using SWIG. By the way, SWIG support for PDFlib was suggested and in its first incarnation implemented by Rainer Schaaf <*Rainer.Schaaf@t-online.de*>[1].

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

PDFlib scripting API. In order to avoid duplicating the PDFlib API reference manual for all supported languages, this manual is considered authoritative not only for the C binding but also for all other languages. Of course, the script programmer has to mentally adapt certain conventions and syntactical issues from C to the relevant language. However, translating C API calls to, say, Perl calls is a straightforward process. Alas, we were able to translate a C PDFlib application to Perl by simply deleting the include directives and adding a bunch of dollar signs to the variable names!

2.1.2 Availability and Special Considerations

Given the broad range of platforms and languages (let alone different versions of both) supported by PDFlib, it shouldn't be much of a surprise that not all combinations of platforms, languages, and versions thereof can be tested. However, we strive to make PDFlib work with the latest available versions of the respective environments. Table 2.1 lists the language/platform combinations we used for testing.

Table 2.1. Tested language and platform combinations

language	*Unix (Linux and others)*	*Windows*	*MacOS 8.6 PPC*
ActiveX	–	*ASP (MIIS 4, PWS);* *WSH (both with VBScript 5.0 and JScript 5.0);* *Visual Basic 6.0 SP3*	–
ANSI C	*GCC* *and other C compilers*	*Microsoft Visual C++ 6.0 SP3* *Watcom C 10.6*	*Metrowerks CodeWarrior 4*
ANSI C++	*GCC*	*Microsoft Visual C++ 6.0 SP3* *Watcom C 10.6*	*Metrowerks CodeWarrior 4*
Java	*Blackdown JDK 1.2.2 RC4* *IBM JDK 1.1.8*	*Sun JDK 1.1.8 and 1.2.2* *MS Visual J++ 6.0 SP3*	*MRJ 2.2, based on JDK 1.1.8*
Perl	*Perl 5.005_03* *and 5.005_63*	*ActivePerl build 522,* *based on Perl 5.005_03*	*MacPerl 5.2.0r4, based on Perl 5.004*
Python	*Python 1.5.2*	*Python 1.5.2*	*Python 1.5.2*
Tcl	*Tcl 8.3*	*Tcl 8.3*	*Tcl 8.3*

1. More information on SWIG can be found at http://www.swig.org

1. On a totally unrelated note, Rainer and his wonderful family live in a nice house close to the Alps – definitely a great place for biking!

2.1.3 The »Hello world« Example

Being a well-known classic, the »Hello, world!« example will be used for the first PDFlib program. It uses PDFlib to generate a one-page PDF file with some text on the page. In the following sections, the »Hello, world!« sample will be shown for all supported language bindings. The code for all language samples is contained in the PDFlib distribution.

2.1.4 Error Handling

PDFlib provides sophisticated means for dealing with different kinds of programming and runtime errors. In order to allow for smooth integration to the respective language environment, PDFlib's error handling is integrated into the language's native way of dealing with exceptions. Basically, C and C++ clients can install custom code which is called when an error occurs. Other language bindings use the existing exception machinery provided by all modern languages. More details on PDFlib's exception handling can be found in Section 3.1.4, »Error Handling«. The sections on error handling in this chapter cover the language-specific details for the supported environments.

2.1.5 Version Control

Taking into account the rapid development cycles of software in general, and Internet-related software in particular, it is important to allow for future improvements without breaking existing clients. In order to achieve compatibility across multiple versions of the library, PDFlib supports several version control schemes depending on the respective language. If the language supports a native versioning mechanism, PDFlib seamlessly integrates it so the client doesn't have to worry about versioning issues except making use of the language-supplied facilities. In other cases, when the language doesn't support a suitable versioning scheme, PDFlib supplies its own major and minor version number at the interface level. These may be used by the client in order to decide whether the given PDFlib implementation can be accepted, or should be rejected because a newer version is required.

2.1.6 Unicode Support

PDFlib supports Unicode for a variety of features (see Section 3.3.9, »Unicode Support« for details). The language bindings, however, differ in their native support for Unicode. If a given language binding supports Unicode strings, the respective PDFlib language wrapper is aware of the fact, and automatically deals with Unicode strings in the correct way.

2.1.7 Summary of the Language Bindings

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

Table 2.2. Summary of the language bindings

language	custom error handling	automatic Unicode conversion	version control	thread-safe
COM/ActiveX	COM exceptions	yes	Class ID and ProgID	yes (both-threading)
C	client-supplied error handler	–	manually	yes
C++	client-supplied error handler	–	manually	yes
Java	Java exceptions	yes	automatically	–
Perl	Perl exceptions	–	via package mechanism	–
Python	Python exceptions	–	manually	–
Tcl	Tcl exceptions	yes (requires Tcl 8.2 or above)	via package mechanism	–

2.2 ActiveX/COM Binding

2.2.1 How does the ActiveX/COM Binding work?

COM (Component Object Model)[1], developed by Microsoft, is a powerful mechanism for reusing software components regardless of the programming language on the client side (the user of the component) or the server side (the actual implementation of a component). In theory, COM is even a platform-independent binary standard which allows clients to communicate with servers within the same process, on the same machine, or a networked machine. In practice, however, COM is basically a standard for the Windows environment (although attempts have been made to port COM to other platforms).

ActiveX is built on Microsoft's COM technology, and used primarily to develop interactive content for the World Wide Web, although it can be used in desktop and other applications. The reusable software components are called ActiveX controls (formerly known as OLE controls or OCX). Although ActiveX burdens the developer with a variety of specific technologies, terms, and troublesome issues (such as type libraries, registration, an assortment of threading models, historical jettison, to name but a few), ActiveX users are rewarded with tight integration and almost universal usability (if you happen to live in the Windows universe).

Since PDFlib is pure component-ware, the library naturally lends itself to an ActiveX implementation for Windows deployment. The ActiveX implementation of PDFlib is built as a wrapper DLL around the core PDFlib DLL. The wrapper DLL calls the PDFlib core functions and is responsible for communicating with the underlying COM machinery, registration and type library issues, COM exception handling, memory management, and Unicode string conversions. This parallels other PDFlib bindings where we strive to make a strict distinction between core functionality and the language wrapper. The

1. See http://www.microsoft.com/com for more information about COM and ActiveX

PDFlib ActiveX wrapper can technically be characterized by as follows (don't worry if you don't understand some of the terms – they are not required for using PDFlib):

- PDFlib acts as a Win32 in-process ActiveX server component (also known as an automation server) without any user interface.
- PDFlib supports is a »both-threaded« component, i.e., it is treated as both an apartment-threaded as well as a free-threaded component. In addition, PDFlib aggregates a free-threaded marshaler. In simple terms, clients can use the PDFlib object directly (instead of going through a proxy/stub pair) which boosts performance.
- PDFlib is fully Unicode-aware.
- The PDFlib binary *pdflib_com.dll* is a self-registering DLL with a type library.
- PDFlib is stateless, i.e., method parameters are used instead of properties.
- PDFlib's dual interface supports both early and late binding.
- PDFlib supports rich error information.

Note PDFlib currently is not MTS-aware (Microsoft Transaction Server).

2.2.2 Availability and Special Considerations for ActiveX

PDFlib can be deployed in all environments that support ActiveX components. We will demonstrate our examples in several specific environments:

- Visual Basic[1]
- Active Server Pages (ASP)[2] with JScript[3]
- Windows Script Host (WSH) with Visual Basic Scripting Edition (VBScript)[4]

Active Server Pages and Windows Script Host both support JScript and VBScript. Since the scripts are nearly identical, however, we do not demonstrate all combinations here. In addition, there are many other ActiveX-aware development environments available – Java, Visual C++, Delphi, PowerBuilder, to name but a few. PDFlib also works in Visual Basic for Applications (VBA).

What the PDFlib ActiveX installer does. The installation program supplied for the PDFlib ActiveX component automatically takes care of all issues related to using PDFlib with ActiveX. For the sake of completeness, the following describes the runtime environment required for using PDFlib (this is taken care of by the installation routine):

- The PDFlib core and auxiliary DLLs are copied to the installation directory.
- The PDFlib ActiveX DLL must be registered with the Windows registry. The installer uses the self-registering PDFlib DLL to achieve the registration.
- If a licensed version of PDFlib is installed, the serial number is entered in the system.

While the »Runtime« installation option performs the above steps, the »Full« installation option additionally copies documentation and sample files to the installation directory.

1. Visual Basic is a commercial product of Microsoft. For more information see http://msdn.microsoft.com/vbasic/prodinfo.
2. Active Server Pages is a technology for executing server-side scripts in a variety of languages. It is available with Microsoft Web servers, and several other server products. More information about ASP can be found at http://msdn.microsoft.com/workshop/server/default.asp.
3. JScript is an extension of ECMAScript (see http://www.ecma.ch/stand/ecma-262.htm) which in turn is based on Netscape's JavaScript. For more information on JScript see http://msdn.microsoft.com/scripting/jscript/default.htm.
4. WSH is available in command-line (cscript.exe) and windowing flavors (wscript.exe). WSH is included in Microsoft Internet Explorer 5, Windows 98, and Windows 2000. For more information see http://msdn.microsoft.com/scripting.

Note When the PDFlib registry entries do not contain a valid serial string, PDFlib will work, but will stamp the generated pages with a diagonal line (the »nagger«). Developers who wish to get rid of the nagger during the evaluation phase or for prototype demos can request a temporary serial string.

Redistributing the PDFlib ActiveX component. Developers who obtained a redistributable runtime license and wish to redistribute the PDFlib ActiveX component along with their own product must do one of the following:

- Ship the complete PDFlib installation and run the PDFlib installer as part of their product's setup process;
- Integrate the files of the PDFlib »Runtime« installation option in their own installation, and take care of the necessary PDFlib registry keys. This can be accomplished by completing the entries in the supplied registration file template *pdflib.reg*, and using it during the installation process of their own product. In addition, *pdflib_com.dll* must be called for self-registration (e.g., using the *regsvr32.exe* utility).
- Supply their serial number at runtime using PDFlib's *set_parameter( )* function, supplying *serial* as first parameter, and the actual serial string as second parameter.

Special considerations for Visual Basic. When it comes to leveraging external ActiveX components, Visual Basic supports both early (compile-time) and late (run-time) binding. Although both types of bindings are possible with PDFlib, early binding is heavily recommended. It is achieved by performing the following steps:

- Create a reference from your VB project to PDFlib via »Project«, »References...«, and selecting the *pdflib_com* control.
- Declare object variables of type *PDFlib_com.PDF* instead of the generic type *Object*:

```
Dim oPDF As PDFlib_com.PDF
Set oPDF = CreateObject("PDFlib_com.PDF")   ' or: Set oPDF = New PDFlib_com.PDF
```

Creating a reference and using early binding has several advantages:

- VB can check the code for spelling errors.
- IntelliSense and context-sensitive help are available.
- The VB object browser shows all PDFlib methods along with their parameters and a short description.
- VB programs run much faster with early binding than with late binding.

PDFlib programming in Visual Basic is straightforward, with one exception. Due to a Microsoft-confirmed bug (pardon: an »issue«) in Visual Basic 6 several PDFlib functions cannot be used directly since VB erroneously overrides PDFlib method names with some built-in methods of VB. For example, the following cannot be successfully compiled in VB 6:

```
oPDF.circle 10, 10, 30
```

In order to work around this problem, Microsoft technical support came up with the following suggestion:

```
oPDF.[circle] 10, 10, 30
```

Putting the critical method name in brackets seems to do the trick. From all PDFlib functions only the following seem to be affected by this problem:

```
circle
scale
```

The data type *integer*, as used in the PDFlib ActiveX component, is a signed 32-bit quantity. In Visual Basic this corresponds to the *long* data type. Therefore, when the PDFlib API reference calls for an *int* type argument, Visual Basic programmers should translate this to *long* (although VB will correctly translate if *int* values are supplied).

Special considerations for Active Server Pages. You can improve the performance of COM objects such as *PDFlib_com* on Active Server Pages by instantiating the object outside the actual script code on the ASP page, effectively giving the object session scope instead of page scope. More specifically, instead of using *CreateObject* (as shown in the example in the next section)

```
<%@ LANGUAGE = "JavaScript" %>
<%
        var oPDF;
        oPDF = Server.CreateObject("PDFlib_com.PDF");
        if (oPDF.open_file("file.pdf") == -1)
            ...
```

use the *OBJECT* tag with the *RUNAT, ID,* and *ProgID* attributes to create the *PDFlib_com* object:

```
<OBJECT RUNAT=Server ID=oPDF ProgID="PDFlib_com.PDF"> </OBJECT>

<%@ LANGUAGE = "JavaScript" %>
<%
        if (oPDF.open_file("file.pdf") == -1)
            ...
```

You can boost performance even more by applying this technique to the *global.asa* file, and using the *Scope=Application* attribute, thereby giving the object application scope.

2.2.3 The »Hello world« Example for several ActiveX Environments

Since there is no single ActiveX example, but only specific usage patterns of the PDFlib ActiveX component, let's pick some ActiveX development environments to see how the »Hello world« example could be done. More specific information for these environments can be found in Section 2.2.2, »Availability and Special Considerations for ActiveX«.

The »Hello world« example in Visual Basic.

```
Attribute VB_Name = "hello"
'
' hello.bas
'
' PDFlib client: hello example in Visual Basic via ActiveX
' Important: the PDFlib ActiveX component must be referenced
' via "Project", "References..."!

Option Explicit

Sub main()
```

```
    Dim ret As Long, font As Long
    Dim oPDF As PDFlib_com.PDF

    Set oPDF = New PDFlib_com.PDF

    ' Open new PDF file
    ret = oPDF.open_file("hello_ax_vb.pdf")
    If (ret = -1) Then
        MsgBox "Couldn't open PDF file!"
        End
    End If

    oPDF.set_info "Creator", "hello.bas"
    oPDF.set_info "Author", "Thomas Merz"
    oPDF.set_info "Title", "Hello, world (ActiveX/VB)!"

    ' start a new page
    oPDF.begin_page 595, 842

    font = oPDF.findfont("Helvetica-Bold", "winansi", 0)

    oPDF.setfont font, 24

    oPDF.set_text_pos 50, 700
    oPDF.show "Hello, world!"
    oPDF.continue_text "(says ActiveX/VB)"

    oPDF.end_page             ' finish page
    oPDF.close                ' close PDF document

    set oPDF = Nothing

End Sub
```

The »Hello world« example in Active Server Pages (ASP) with JScript.

Note *Unlike the other examples we do not create a PDF output file in the ASP example. Instead, we generate the PDF data in memory and directly send it to the client via HTTP. This technique is much more appropriate to a Web server environment.*

```
<%@ LANGUAGE = "JavaScript" %>
<%
// hello.js
//
// PDFlib client: hello example for ActiveX with Active Server Pages and JScript
// Requires the PDFlib ActiveX component
//
        var font;
        var oPDF;

        oPDF = Server.CreateObject("PDFlib_com.PDF");

        if (oPDF == null) {
            Response.write("Couldn't create PDFlib object!<br>");
            Response.end();
        }

        Response.Expires = 0;
```

```
        Response.Buffer = true;
        Response.ContentType = "application/pdf";

        // Open new PDF file
        oPDF.open_file("")

        oPDF.set_info("Creator", "hello.js");
        oPDF.set_info("Author", "Thomas Merz");
        oPDF.set_info("Title", "Hello, world (Active X/ASP/JScript)!");

        // start a new page
        oPDF.begin_page(595, 842);

        font = oPDF.findfont("Helvetica-Bold", "winansi", 0);

        oPDF.setfont(font, 24);

        oPDF.set_text_pos(50, 700);
        oPDF.show("Hello, world!");
        oPDF.continue_text("(says ActiveX/ASP/JScript)");

        oPDF.end_page();
        oPDF.close();

        Response.BinaryWrite(oPDF.get_buffer());
        Response.End()

        oPDF = "";
%>
```

The »Hello world« example in Windows Script Host (WSH) with VBScript.

```
' hello.vbs
'
' PDFlib client: hello example for ActiveX with Windows Script Host and VBS
' Requires the PDFlib ActiveX component
'
Option Explicit
On Error Resume Next

Dim font
Dim oPDF

Set oPDF = CreateObject("PDFlib_com.PDF")

' Open new PDF file
if (oPDF.open_file("hello_ax_vbs.pdf") = -1) then
        WScript.Echo "Couldn't open PDF file!"
        WScript.Quit(1)
end if

oPDF.set_info "Creator", "hello.asp"
oPDF.set_info "Author", "Thomas Merz"
oPDF.set_info "Title", "Hello, world (Active X/VBS)!"

' start a new page
oPDF.begin_page 595, 842
```

```
font = oPDF.findfont("Helvetica-Bold", "winansi", 0)

oPDF.setfont font, 24

oPDF.set_text_pos 50, 700
oPDF.show "Hello, world!"
oPDF.continue_text "(says ActiveX/VBS)"

oPDF.end_page
oPDF.close
set oPDF = Nothing
```

2.2.4 Error Handling in ActiveX

Error handling for the PDFlib Active component is done according to COM conventions: when a PDFlib-internal exception occurs, a COM exception is raised and furnished with the PDFlib error code and a clear-text description of the error. In additon, *PDF_delete( )* is called internally. Table 2.3 lists all COM errors thrown by PDFlib along with the corresponding PDFlib exceptions. The COM exception may be caught and handled in the PDFlib client in whichever way the client environment supports for handling COM errors.

The error codes used in COM are 32-bit values with the highest bit set, which makes them look like very large negative numbers. PDFlib conforms to the COM conventions, and returns error codes in the range which is reserved for application-defined errors. More specifically, the error codes are constructed as follows:

```
COM error code = 0x80040000 + 0x200 + (PDFlib error code)
```

(The first hexadecimal number is equal to Visual Basic's *vbObjectError* constant, the second is the Microsoft-suggested offset for component-specific errors.)

Table 2.3 lists all PDFlib error names along with the decimal and hexadecimal error codes. A more detailed discussion of PDFlib's exception mechanism can be found in Section 3.1.4, »Error Handling«). Fortunately, ActiveX programmers need not deal with these error numbers directly since the *PDFlib_com* type library provides symbolic constants in the *PDFlib_com.Errors* class.

Table 2.3. COM error codes raised by PDFlib

PDFlib error name	decimal value	hexadecimal value	explanation
MemoryError	*-2147220991*	*&H80040201*	*not enough memory*
IOError	*-2147220990*	*&H80040202*	*input/output error, e.g. disk full*
RuntimeError	*-2147220989*	*&H80040203*	*wrong order of PDFlib function calls*
IndexError	*-2147220988*	*&H80040204*	*array index error*
TypeError	*-2147220987*	*&H80040205*	*argument type error*
DivisionByZero	*-2147220986*	*&H80040206*	*division by zero*
OverflowError	*-2147220985*	*&H80040207*	*arithmetic overflow*
SyntaxError	*-2147220984*	*&H80040208*	*syntactical error*
ValueError	*-2147220983*	*&H80040209*	*a value supplied as argument to PDFlib is invalid*
SystemError	*-2147220982*	*&H8004020A*	*PDFlib internal error*

Table 2.3. COM error codes raised by PDFlib

PDFlib error name	decimal value	hexadecimal value	explanation
NonfatalError	*-2147220981*	*&H8004020B*	*a non-fatal problem was detected*
UnknownError	*-2147220980*	*&H8004020C*	*other error*

Let's take a look at how our sample environments deal with exceptions. Real-world examples, of course, would take a more sophisticated approach in dealing with the error situation.

Error handling in Visual Basic. A Visual Basic program can detect when an error happens, and react upon the error. Catching Exceptions in Visual Basic is achieved with an *On Error GoTo* clause:

```
Sub main()
        Dim oPDF As PDFlib_com.PDF
        On Error GoTo ErrExit

        ...some PDFlib instructions...

        End

ErrExit:
        MsgBox Hex(Err.Number) & ":  " & Err.Description
End Sub
```

Note You can disable error handling in VBScript with the undocumented On Error GoTo 0 statement (i.e., using zero as address for the GoTo statement).

Error handling in JScript. JScript 5.0[1] adds structured exception handling to the language which looks similar to C++ or Java, with the difference that JScript exceptions cannot be typed, and only a single clause can deal with an exception. Detecting an exception and acting upon it is achieved with a *try ... catch* clause:

```
try {
        ...some PDFlib instructions...
} catch (exc) {
        Response.write("Error " + exc.number + ": " + exc.description + "<br>");
        Response.end();
}
```

Note Due to some problem with JScript's integer handling it's impossible to directly compare exception numbers with hexadecimal values. Comparison with decimal values, however, works fine.

Error handling in VBScript. Unfortunately, VBScript doesn't have any means for catching errors, but only for ignoring them. For this reason one has to periodically check the *ERR* object in order to see whether something went wrong in one of the previous calls to the ActiveX component. VBScript's missing *On Error GoTo* clause has the major drawback that the script code is either cluttered with calls to the error checking routine, or subsequent errors may happen between the first error and the next invocation of the error checking routine:

1. JScript 5.0 is available with Microsoft Internet Explorer 5.0 and Microsoft Internet Information Services 5.0

```
On Error Resume Next
Err.Clear

...some PDFlib instructions...

CheckPDFError

...more PDFlib instructions...


Sub CheckPDFError()
        If Err.number <> 0 then
            WScript.Echo "Error " & Hex(Err.number) & ": " & Err.description
            Err.Clear
        End If
end Sub
```

2.2.5 Version Control in ActiveX

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

PDFlib_com, being an ActiveX component, makes use of the class ID mechanism. The GUID for *PDFlib_com* is contained in its type library (which in turn is contained in *pdflib_com.dll*), and in the Windows registry.

Since PDFlib is registered under both the generic program identifier (ProgID) *PDFlib_com.PDF*, as well as a version-specific ProgID, users will rarely have to deal with the GUID directly.

2.2.6 Unicode Support in ActiveX

32-bit versions of ActiveX/COM support Unicode natively. The ActiveX language wrapper automatically converts all COM strings to Unicode or ISO Latin 1 (PDFDocEncoding), as appropriate. ActiveX's Unicode-awareness, however, may lead to subtle problems regarding 8-bit encodings (such as *winansi*) and Unicode characters in literal strings. More details on this issue can be found in Section 3.3.9, »Unicode Support«.

Unicode support in Visual Basic. Visual Basic supports Unicode internally. (VB's program editor, however, doesn't seem to be fully Unicode-aware). Unicode strings can be constructed from numerical values using the *ChrW* function:

```
Unicodetext = ChrW(&H39B) & ChrW(&H39F) & ChrW(&H393) & ChrW(&H39F) & ChrW(&H3A3)
```

Unicode support in JScript. JScript supports Unicode internally. Unicode characters can be written directly into string literals using a Unicode-aware text editor; entered with an escape sequence such as

```
Unicodetext = "\u039B\u039F\u0393\u039F\u03A3";
```

or constructed from numerical values using the *String.fromCharCode* method:

```
Unicodetext = String.fromCharCode(0x39B, 0x39F, 0x393, 0x39F, 0x3A3);
```

Unicode support in VBScript. VBScript supports Unicode internally. Similar to Visual Basic, Unicode strings can be constructed from numerical values using the *ChrW* function:

```
Unicodetext = ChrW(&H39B) & ChrW(&H39F) & ChrW(&H393) & ChrW(&H39F) & ChrW(&H3A3)
```

2.3 C Binding

2.3.1 How does the C Binding work?

In order to use the PDFlib C binding, you need to build a static or shared library (DLL on Windows), and you need the central PDFlib include file *pdflib.h* for inclusion in your PDFlib client source modules. The PDFlib distribution is prepared for building both static or dynamic versions of the library.

On Windows, using DLLs involves some issues related to the function calling conventions and export or import of DLL functions. The *pdflib.h* header file deals with these issues by defining appropriate macros for both the library itself as well as for PDFlib clients. This macro system is set up in a way that PDFlib clients don't need to take any special measures in order to get the required import statements from the header file. However, if you are using function pointers for accessing PDFlib functions (instead of direct calls) you must make sure that your function pointers are declared using the same calling conventions as dictated by *pdflib.h* (depending on whether the static or shared library is used), since otherwise your program will immediately crash.

2.3.2 Availability and Special Considerations for C

PDFlib itself is written in the ANSI C language, and assumes ANSI C clients as well as 32-bit platforms (at least). No provisions have been made to make PDFlib compatible with older C compilers, or 16-bit platforms.

2.3.3 The »Hello world« Example in C

```
/* hello.c
 *
 * PDFlib client: hello example in C
 *
 */

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

#include "pdflib.h"

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

    p = PDF_new();

    /* open new PDF file */
    if (PDF_open_file(p, "hello_c.pdf") == -1) {
        fprintf(stderr, "Error: cannot open PDF file hello_c.pdf.\n");
```

```
        exit(2);
    }

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

    PDF_begin_page(p, a4_width, a4_height);     /* start a new page */

    font = PDF_findfont(p, "Helvetica-Bold", "host", 0);

    PDF_setfont(p, font, 24);
    PDF_set_text_pos(p, 50, 700);
    PDF_show(p, "Hello, world!");
    PDF_continue_text(p, "(says C)");
    PDF_end_page(p);                            /* close page */

    PDF_close(p);                               /* close PDF document */

    exit(0);
}
```

2.3.4 Error Handling in C

C or C++ clients can install a custom error handler routine with *PDF_new2( )*. In case of an error this routine will be called with a pointer to the PDF structure, the error type and a descriptive string as arguments. A list of PDFlib error types can be found in Section 3.1.4, »Error Handling«. Macro definitions for the error types can be found in *pdflib.h*. These are constructed by prefixing the error name with *PDF_* (e.g., *PDF_MemoryError*). The opaque data pointer argument to *PDF_new2( )* is useful for multi-threaded applications which want to supply a handle to thread- or class-specific data in the *PDF_new2( )* call. PDFlib supplies the opaque pointer to the user-supplied error and memory handlers via a call to *PDF_get_opaque( )*, but doesn't otherwise use it.

An important task of the error handler is to clean up PDFlib internals using *PDF_delete( )* and the supplied pointer to the PDF object. *PDF_delete( )* will also close the output file if necessary. PDFlib functions other than *PDF_delete( )* should not be called from within a client-supplied error handler.

Except for non-fatal errors (type *NonfatalError*), client-supplied error handlers are expected to not return to the library function which raised the exception. This can be achieved by using the *setjmp( )/longjmp( )* facility.

The following code may be used as a starting point for developing a custom error handler:

```
void custom_errorhandler(PDF *p, int type, const char* shortmsg)
{
    char msg[256];

    sprintf(msg, "Application error: %s\n", shortmsg);

    (void) fprintf(stderr, msg);        /* Issue a warning message in all cases */

    switch (type) {
        case PDF_NonfatalError:
            return;
```

```
        case PDF_MemoryError:           /* you can act on specific errors here */
        case PDF_IOError:
        case PDF_RuntimeError:
        case PDF_IndexError:
        case PDF_TypeError:
        case PDF_DivisionByZero:
        case PDF_OverflowError:
        case PDF_SyntaxError:
        case PDF_ValueError:
        case PDF_SystemError:
        case PDF_UnknownError:
        default:

            if (p != NULL)              /* first allocation? */
                PDF_delete(p);          /* clean up PDFlib */

            exit(99);                   /* brutal way of saying good-bye */
    }
}
```

Obviously, the appropriate action when an error happens is completely application specific. The above sample doesn't even attempt to handle the error, but simply exits. A custom error handler can be installed in PDFlib by using *PDF_new2( )*.

2.3.5 Version Control in C

In the C language binding there are two basic versioning issues:

- Does the PDFlib header file in use for a particular compilation correspond to the PDFlib binary?
- Is the PDFlib library in use suited for a particular application, or is it too old?

The first issue can be dealt with by comparing the macros *PDFLIB_MAJORVERSION* and *PDFLIB_MINORVERSION* supplied in *pdflib.h* with the return values of the API functions *PDF_get_majorversion( )* and *PDF_get_minorversion( )* which return PDFlib major and minor version numbers.

The second issue can be dealt with by comparing the return values of the above-mentioned functions with fixed values corresponding to the needs of the application.

On Unix platforms the PDFlib library file name may contain version information if the platform supports it (see Appendix A, »Shared Libraries and DLLs«). In this case PDFlib leverages operating system support for library versioning.

2.3.6 Unicode Support in C

C developers must manually construct their Unicode strings according to Section 3.3.9, »Unicode Support«. For CJK encoding which may contain null characters, the *PDF_show2( )* functions etc. must be used, since their counterparts *PDF_show( )* etc. expect regular null-terminated C-style strings which don't support embedded null characters.

2.4 C++ Binding

2.4.1 How does the C++ Binding work?

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

Using the C++ object wrapper effectively replaces the *PDF_* prefix in all PDFlib function names with the more object-oriented *p->* approach. Keep this in mind when reading the PDFlib API descriptions.

2.4.2 Availability and Special Considerations for C++

Although the PDFlib C++ binding assumes an ANSI C++ environment, this is not strictly required by the implementation. In fact, we work around some issues related to non-ANSI-conforming compilers in *pdflib.hpp* and *pdflib.cpp*. It may be worthwhile to add namespace support to the PDFlib C++ wrapper, but this is currently not implemented due to restrictions in the namespace handling of some widely used compilers.

In most environments there are inherent issues related to C++ deployment in shared libraries which adversely affect portability. For this reason it is suggested to statically bind *pdflib.cpp* to your application, and use the generic PDFlib C library as a shared library (if shared libraries are to be used at all).

2.4.3 The »Hello world« Example in C++

```
// hello.cpp
//
// PDFlib client: hello example in C++
//
//

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

#include "pdflib.hpp"

int
main(void)
{
    PDF *p;             // pointer to the PDF class
    int font;

    p = new PDF();

    // Open new PDF file
    if (p->open("hello_cpp.pdf") == -1) {
        fprintf(stderr, "Error: cannot open PDF file hello_cpp.pdf.\n");
        exit(2);
    }

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

```
    // start a new page
    p->begin_page((float) a4_width, (float) a4_height);

    font = p->findfont("Helvetica-Bold", "host", 0);

    p->setfont(font, 24);

    p->set_text_pos(50, 700);
    p->show("Hello, world!");
    p->continue_text("(says C++)");
    p->end_page();                  // finish page

    p->close();                     // close PDF document
    delete p;

    exit(0);
}
```

2.4.4 Error Handling in C++

Error handling for PDFlib clients written in C++ works the same as error handling in C, so everything in Section 2.3.4, »Error Handling in C« applies to C++, too. In addition, a number of C++ peculiarities must be observed:

A C++ error handler can be supplied in the PDF constructor, which has the same signature as the *PDF_new2()* function. The C++ error handler must not be a class method since it will be called indirectly through a function pointer without any class association.

In the C++ binding, the *PDF* data type refers to a C++ class, not to the structure used in the C binding (this change is automatically accomplished via simple macro substitution in the header files). However, the C++ error handler lives on the client side, but has to deal with the PDFlib-internal C data structure. For this reason, C++ error handlers must use the (rather private) data type name *PDF_c* although the PDFlib API reference calls for the *PDF* data type.

Finally, a note for those brave folks who want to throw C++ exceptions in their client-supplied PDFlib error handler: don't do it! Since PDFlib is a C implementation, the error handler will be called from a C-style stack without any exception and stack unwinding information, so throwing a C++ exception in the error handler is likely to result in a crash. The correct way to do it is to install a C-style error handler, do a *longjmp()* to a C++ function, and throw the C++ exception from there (since we're now back on the C++ stack).

2.4.5 Version Control in C++

Version control for the C++ binding is identical to version control in the C binding (see Section 2.3.5, »Version Control in C«)

2.4.6 Unicode Support in C++

Unicode support for the C++ binding is identical to Unicode support in the C binding (see Section 2.3.6, »Unicode Support in C«).

2.5 Java Binding

2.5.1 How does the Java Binding work?

Starting with the Java[1] Development Kit (JDK) 1.1, Java supports a portable mechanism for attaching native language code to Java programs, the Java Native Interface (JNI)[2]. 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 simply 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 loading the shared library.

Taking into account PDFlib's stability and maturity (and the availability of source code), attaching the native PDFlib library to the Java VM doesn't impose any stability or security restrictions on your Java application, while at the same time offering the performance benefits of a native implementation. Regarding portability (at least on the server side), remember that PDFlib runs on many more platforms than the Java VM!

2.5.2 Availability and Special Considerations for Java

Obviously, for developing Java applications you will need the JDK which includes support for the JNI. For compiling the PDFlib-supplied JNI wrapper file, you will need the JNI include files for C which are part of the JDK (or SDK, if the vendor distinguishes between runtime and development environment).

The JDK has been ported to Linux [3] and many other Unix platforms. Apple's Java implementation, the MacOS Runtime for Java (MRJ)[4], version 2.0 and above, also supports the JNI.

In order to comply with a court ruling in a law suit against Sun, Microsoft also had to implement JNI support in their Java environment. Microsoft started shipping JNI support in Visual J++ 6.0 SP2, and the Java VM shipped in Internet Explorer 4.01 SP1a.

For the PDFlib binding to work, the Java VM must have access to the PDFlib shared library, the auxiliary libraries, and the PDFlib Java package.

The PDFlib Java package. In order to maintain a consistent look-and-feel for the Java developer, starting with version 3.0 PDFlib is organized as a Java package with the following package name:

```
com.pdflib.pdflib
```

This package is available in the *pdflib.jar* file. You can generate an abbreviated version of the PDFlib API reference (this manual) using the *javadoc* utility since the PDFlib class contains the necessary *javadoc* comments.

1. See http://java.sun.com
2. See http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html
3. See http://www.blackdown.org and http://www.ibm.com/developer/java
4. See http://devworld.apple.com/java

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

Unix. On Unix systems the library name supplied in the PDFlib Java class file will be decorated according to the system's naming conventions for the names of shared libraries (usually by prepending *lib* and appending *.so*). The library must be placed in one of the default locations for shared libraries, or in an appropriately configured directory (see Appendix A, »Shared Libraries and DLLs« for details).

Windows. On Windows systems the library name supplied in the PDFlib Java class file will be decorated according to the usual Windows naming conventions for DLLs (by appending *.dll*, resulting in *pdf_java.dll*). The DLL must be placed in the Windows system directory, the current directory, or a directory which is listed in the PATH environment variable.

Macintosh. On the Mac the library name supplied in the PDFlib Java class file is used without any change (i.e. *pdf_java*). The shared library is searched in the *Systems Extensions* folder, the *MRJ Libraries* folder within the *Extensions* folder, and the folder where the starting application lives (JBindery, for example). Note that the above naming not only relates to the file name, but also to the fragment name of the library which must be correctly set by the linker. Also, you may want to increase the amount of memory allocated to JBindery, especially if you want to process images with PDFlib.

2.5.3 The »Hello world« Example in Java

```
/* hello.java
 *
 * PDFlib client: hello example in Java
 */

import com.pdflib.pdflib;

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

        p = new pdflib();

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

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

        p.begin_page(595, 842);
```

```
        font = p.findfont("Helvetica-Bold", "host", 0);

        p.setfont(font, 18);

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

        p.close();
    }
}
```

2.5.4 Error Handling in Java

The Java binding installs a special error handler which translates PDFlib errors to native Java exceptions according to Table 2.4. The Java exceptions can be dealt with by applying the appropriate language constructs, i.e., by bracketing critical sections:

```
try {
        ...some PDFlib instructions...
} catch (Throwable e) {
        System.err.println("Exception caught:\n" + e);
}
```

Table 2.4. Java exceptions thrown by PDFlib

PDFlib error name	***Java exception***	***explanation***
MemoryError	*java.lang.OutOfMemoryError*	*not enough memory*
IOError	*java.io.IOException*	*input/output error, e.g. disk full*
RuntimeError	*java.lang.IllegalArgumentException*	*wrong order of PDFlib function calls*
IndexError	*java.lang.IndexOutOfBoundsException*	*array index error*
TypeError	*java.lang.ClassCastException*	*argument type error*
DivisionByZero	*java.lang.ArithmeticException*	*division by zero*
OverflowError	*java.lang.ArithmeticException*	*arithmetic overflow*
SyntaxError	*java.lang.RuntimeException*	*syntactical error*
ValueError	*java.lang.IllegalArgumentException*	*a value supplied as argument to PDFlib is invalid*
SystemError	*java.lang.InternalError*	*PDFlib internal error, or incompatible PDFlib library version*
NonfatalError	*java.lang.UnknownError*	*warnings (can be disabled)*
UnknownError	*java.lang.UnknownError*	*other error*

2.5.5 Version Control in Java

Version control for the Java binding is done transparently when loading the shared library. The wrapper code for loading the PDFlib shared library relies on the major and minor version numbers (which are queried from the PDFlib library). An exact match of both minor and major version number is required. If a PDFlib library with a non-matching version number is found, the wrapper code will raise a *java.lang.InternalError* exception.

2.5.6 Unicode Support in Java

Java supports Unicode natively. The Java language wrapper automatically converts all Java strings to Unicode or ISO Latin 1 (PDFDocEncoding), as appropriate. Java's Unicode-awareness, however, may lead to subtle problems regarding 8-bit encodings (such as *winansi*) and Unicode characters in literal strings. More details on this issue can be found in Section 3.3.9, »Unicode Support«.

Unicode characters can be written directly into code and string literals using a Unicode-aware text editor, or entered with an escape sequence such as

```
Unicodetext = "\u039B\u039F\u0393\u039F\u03A3";
```

2.6 Perl Binding

2.6.1 How does the Perl Binding work?

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

2.6.2 Availability and Special Considerations for Perl

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.

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

For the PDFlib binding to work, the Perl interpreter must have access to the PDFlib shared library, the auxiliary libraries, and the module file *pdflib_pl.pm*. In additon to the platform-specific methods described below you can add a directory to Perl's *@INC* search path using a command similar to the following:

```
use lib qw(/path/to/lib);
```

before the *use pdflib_pl* line. However, Perl will only use the path given in this command for the PDFlib wrapper library (e.g., *pdflib_pl.so.0)*, but not any required auxiliary libraries. These must in turn be made available via operating system specific methods (see Appendix A, »Shared Libraries and DLLs«).

Unix. On Unix systems both *pdflib_pl.so* and *pdflib_pl.pm* will be found if placed in the current directory, or the directory printed by the following Perl command:

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

Perl will also search the subdirectory *auto/pdflib_pl*. PDFlib's install mechanism will place the files in the correct directories. The PDFlib base shared library must also be accessible. Typical output of the above command looks like

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

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

If you don't have administrator privileges (e.g., on a machine run by your Internet Service Provider), you may run into special issues related to locating shared libraries. See Appendix A, »Shared Libraries and DLLs« for a more detailed description, and a possible solution of this problem.

Windows. On the Windows platform PDFlib supports the ActiveState port of Perl 5 to Windows, also known as ActivePerl.[1] PDFlib does not work with the Microsoft port of Perl 5, or other old ports which do not use the »object mode« for building the Perl executable. Both *pdflib_pl.dll* and *pdflib_pl.pm* will be found if placed in the current directory, or the directory printed by the following Perl command:

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

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

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

Macintosh. PDFlib supports the Macintosh port of Perl known as MacPerl[2]. Both the shared library *pdflib_pl* and *pdflib_pl.pm* will be found if placed in the current folder, or in one of the following folders:

```
<MacPerl>:lib
<MacPerl>:lib:MacPPC
```

where <*MacPerl*> denotes the Perl installation folder. In order to run the supplied samples, start Perl and open the script via »Script«, »Run Script«. It should be noted that the generated PDF output ends up in the Perl interpreter's folder if a relative file name is supplied (as in the sample scripts). You may want to increase the memory allocated to the Perl interpreter, especially if you want to process images with PDFlib.

2.6.3 The »Hello world« Example in Perl

```
#!/usr/bin/perl
# hello.pl
#
# PDFlib client: hello example in Perl
#

use pdflib_pl 3.0;

$p = PDF_new();

die "Couldn't open PDF file" if (PDF_open_file($p, "hello_pl.pdf") == -1);

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

PDF_begin_page($p, 595, 842);
```

1. See http://www.activestate.com
2. See http://www.macperl.com

```
$font = PDF_findfont($p, "Helvetica-Bold", "host", 0);

PDF_setfont($p, $font, 18.0);

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

PDF_delete($p);
```

2.6.4 Error Handling in Perl

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

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

2.6.5 Version Control in Perl

Perl's package mechanism supports a major/minor version number scheme for extension modules which is used by the PDFlib Perl binding. PDFlib applications written in Perl simply use the line

```
use pdflib_pl 3.0;
```

in order to make sure they will get the required library version (or a newer one).

2.6.6 Unicode Support in Perl

Perl developers must manually construct their Unicode strings according to Section 3.3.9, »Unicode Support«.

2.7 Python Binding

2.7.1 How does the Python Binding work?

Python[1] supports a mechanism for extending the language (interpreter) via native C libraries. The PDFlib wrapper for Python consists of a C wrapper file. The C module is used to build a shared library which is loaded at runtime by the Python interpreter. The shared library module is referred to from the Python script via an *import* statement.

2.7.2 Availability and Special Considerations for Python

The Python extension mechanism works by loading shared libraries at runtime.

In order to compile the PDFlib-supplied Python wrapper file, you will need to have the Python sources installed because the wrapper file needs the *Python.h* header file

1. See http://www.python.org

from the Python source file set. On the Mac, it suffices to select the »Developers Kit« installation option.

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

Unix. On Unix systems the PDFlib shared library for Python *pdflib_py.so* will be searched in the directories listed in the PYTHONPATH environment variable. The PDFlib base shared library must also be accessible.

Windows. On Windows systems the PDFlib shared library *pdflib_py.dll* will be searched in the directories listed in the PYTHONPATH environment variable.

Macintosh. On the Mac the PDFlib shared library *pdflib_py.ppc.slb* will be searched in the *Mac:Plugins* folder of the Python application folder.

2.7.3 The »Hello world« Example in Python

```
#!/usr/bin/python
# hello.py
#
# PDFlib client: hello example in Python
#

from sys import *
from pdflib_py import *

p = PDF_new()

if PDF_open_file(p, "hello_py.pdf") == -1:
    print 'Couldn\'t open PDF file!', "hello_py.pdf"
    exit(2);

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

PDF_begin_page(p, 595, 842)
font = PDF_findfont(p, "Helvetica-Bold", "host", 0)

PDF_setfont(p, font, 18.0)

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

PDF_delete(p);
```

2.7.4 Error Handling in Python

The Python binding installs a special error handler which translates PDFlib errors to native Python exceptions according to Table 2.5. The Python exceptions can be dealt with by applying the appropriate language constructs, i.e., by bracketing critical sections:

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

Table 2.5. Python exceptions thrown by PDFlib

PDFlib error name	Python exception	explanation
MemoryError	MemoryError	not enough memory
IOError	IOError	input/output error, e.g. disk full
RuntimeError	RuntimeError	wrong order of PDFlib function calls
IndexError	IndexError	array index error
TypeError	TypeError	argument type error
DivisionByZero	ZeroDivisionError	division by zero
OverflowError	OverflowError	arithmetic overflow
SyntaxError	SyntaxError	syntactical error
ValueError	ValueError	a value supplied as argument to PDFlib is invalid
SystemError	SystemError	PDFlib internal error
NonfatalError	RuntimeError	warnings (can be disabled)
UnknownError	RuntimeError	other error

2.7.5 Version Control in Python

We are currently not aware of any intrinsic versioning scheme available for Python. Currently PDFlib applications in Python must use manual version control.

2.7.6 Unicode Support in Python

Python developers must manually construct their Unicode strings according to Section 3.3.9, »Unicode Support«.

2.8 Tcl Binding

2.8.1 How does the Tcl Binding work?

Tcl[1] supports a mechanism for extending the language (interpreter) via native C libraries. The PDFlib wrapper for Tcl consists of a C wrapper file. The C module is used to build a shared library which is loaded at runtime by the Tcl interpreter.

In addition, the PDFlib Tcl binding leverages the idea of extension packages introduced in Tcl 7.5. All PDFlib functions are packed into a single Tcl extension package. The shared library module is referred to from the Tcl script via a *package* statement.

2.8.2 Availability and Special Considerations for Tcl

The Tcl extension mechanism works by loading shared libraries at runtime. For extending the Tcl interpreter with PDFlib, Tcl 8.0 or higher is required (because of its support for binary strings). Unicode support requires Tcl 8.2 or higher. The PDFlib wrapper code

1. See http://www.scriptics.com and http://www.tclconsortium.org

for Tcl may also be compiled for older versions of Tcl (down to 8.0). The supplied binaries, however, require Tcl 8.2 or higher.

In order to compile the PDFlib-supplied Tcl wrapper file, you will need to have the Tcl sources installed because the wrapper file needs the *tcl.h* and *tk.h* header files from the Tcl source file set. On Windows, the required headers will be available if you installed Tcl with the »Header and Library Files« option.

For the PDFlib binding to work, the Tcl shell must have access to the PDFlib shared library (the supplied test programs use *auto_path* to make the library available from the current directory; this facilitates testing) and the package index file *pkgIndex.tcl*:

Unix. On Unix systems the library name *pdflib_tcl.so* supplied in the *pkgIndex.tcl* file must be placed in one of the default locations for shared libraries, or in an appropriately configured directory (see Appendix A, »Shared Libraries and DLLs« for details). The PDFlib base shared library must also be accessible.

Windows. Unfortunately, Tcl doesn't itself produce a platform-specific decoration of the library name. If you compile the Tcl binding from source code, you must change the library name *pdflib_tcl.so* supplied in the *pkgIndex.tcl* file to the appropriate name *pdflib_tcl.dll* (this has already been done in the binary distribution). A library by this name will be searched in the Tcl shell's directory, the current directory, the Windows and Windows\system32 directories, and the directories listed in the PATH environment variable. The files *pkgIndex.tcl* and *pdflib_tcl.dll* will be searched for in the directories

```
C:\Program Files\Tcl 8.3\lib\tcl8.3\
C:\Program Files\Tcl 8.3\lib\tcl8.3\pdflib
```

Make sure to have the auxiliary DLLs accessible, too.

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

```
System:Extensions:Tool Command Language:tcl8.3
System:Extensions:Tool Command Language:tcl8.3:pdflib
```

In order to run the supplied samples, start the *Wish* application and use the »Source« menu command to locate the Tcl script. It should be noted that the generated PDF output ends up in the Tcl shell's folder if a relative file name is supplied (as in the sample scripts).

2.8.3 The »Hello world« Example in Tcl

```
#!/bin/sh
#
# hello.tcl
#
# PDFlib client: hello example in Tcl
#

# Hide the exec to TCL but not to the shell by appending a backslash\
exec tclsh "$0" ${1+"$@"}

# The lappend line is unnecessary if PDFlib has been installed
# in the Tcl package directory
```

```
lappend auto_path .

package require pdflib 3.0

set p [PDF_new]

if {[PDF_open_file $p "hello_tcl.pdf"] == -1} {
    puts stderr "Couldn't open PDF file!"
    exit
}

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

PDF_begin_page $p 595 842
set font [PDF_findfont $p "Helvetica-Bold" "host" 0 ]

PDF_setfont $p $font 18.0

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

PDF_delete $p
```

2.8.4 Error Handling in Tcl

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

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

2.8.5 Version Control in Tcl

Tcl's package mechanism supports a major/minor version number scheme for extension modules which is used by the PDFlib Tcl binding. PDFlib applications written in Tcl simply use the line

```
package require pdflib 3.0
```

in order to make sure they will get the required library version (or a newer one, which is ok for PDFlib).

2.8.6 Unicode Support in Tcl

Starting with version 8.2, Tcl supports Unicode natively. The Tcl language wrapper automatically converts all Tcl strings to Unicode or ISO Latin 1 (PDFDocEncoding), as appropriate. Tcl's Unicode-awareness, however, may lead to subtle problems regarding 8-bit encodings (such as *winansi*) and Unicode characters in literal strings. More details on this issue can be found in Section 3.3.9, »Unicode Support«.

Unicode characters can be written directly into code and string literals using a Unicode-aware text editor, or entered with an escape sequence such as

```
set Unicodetext "\u039B\u039F\u0393\u039F\u03A3"
```

3 PDFlib Programming Concepts

3.1 General Programming Issues

3.1.1 PDFlib Program Structure

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

PDFlib checks for several conditions in the ordering of API calls, but doesn't attempt to trap all kinds of illegal function call combinations. In the development phase it will be helpful to take a look at all warning messages generated by PDFlib, since these usually point to problems in the client's ordering of function calls. PDFlib will throw an exception if bad parameters are supplied by a library client.

3.1.2 Memory Management

Note This section applies to C and C++ PDFlib clients only. All other language bindings leverage the internal memory management of the language environment.

PDFlib dynamically allocates and frees lots of small and large memory chunks. The general strategy is to strictly separate PDFlib memory from client memory. In order to achieve this, data supplied by the client to PDFlib functions is copied into PDFlib memory space if the data is still needed after the call is finished. Consequently, PDFlib is responsible for freeing such memory when the data is no longer needed.

In order to allow for maximum flexibility, PDFlib's internal memory management routines (which are based on standard C *malloc/free*) may be replaced by external procedures provided by the client. These procedures will be called for all PDFlib-internal memory allocation or deallocation.

It is not reasonable to provide custom memory management routines from the scripting language bindings (since freeing the programmer from memory management chores is a major advantage of scripting languages). For this reason, custom memory management routines are only available for the C and C++ programmer. For all other language bindings memory management is handled in the respective wrapper code.

Memory Management in C. Memory management routines can be installed with a call to *PDF_new2( )*, and will be used in lieu of PDFlib's internal memory management routines. Either all or none of the following routines must be supplied:

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

These routines must adhere to the standard C *malloc/free/realloc* semantics, but may choose an arbitrary implementation. All routines will be supplied with a pointer to the calling PDF object. The only exception to this rule is that the very first call to the alloca-

tion routine will supply a PDF pointer of NULL. Client-provided memory allocation routines must therefore be prepared to deal with a NULL PDF pointer.

Using the *PDF_get_opaque( )* function, an opaque application specific pointer can be retrieved from the PDF object. The opaque pointer itself is supplied by the client in the *PDF_new2( )* call. The opaque pointer is useful for multi-threaded applications which may want to keep a pointer to thread- or class specific data inside the PDF object, for use in memory management or error handling routines.

The signatures of the memory management routines can be found in Section 4.2, »General Functions«.

Memory Management in C++. The PDF constructor accepts an optional error handler, optional memory management procedures, and an optional opaque pointer argument. Default NULL arguments are supplied in *pdflib.hpp* which will result in PDFlib's internal error and memory management routines becoming active.

Client-supplied memory management for the C++ binding works the same as with the C language binding. As with the error handler, the signatures of the memory management routines must be slightly changed to use *PDF_c* instead of *PDF* as their first argument.

Note User-supplied memory management routines are used (besides PDFlib) in the Zlib compression library, but not in the TIFF and PNG libraries.

3.1.3 Generating PDF Documents directly 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. Unix users can write the generated PDF to the *stdout* channel and consume it in a pipe process by supplying »–« as filename for *PDF_open_file( )*.

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

The active in-core PDF generation interface. In order to generate the PDF data in memory simply supply an empty filename to *PDF_open_file( )*, and retrieve the data with *PDF_get_buffer( )*. This is considered »active« mode since the client decides when he wishes to fetch the buffer contents.

Note C and C++ clients must neither touch nor 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_mem( )*) a callback function which will be called at unpredictable times by PDFlib whenever PDF data

is waiting to be consumed. However, 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 *PDF_set_parameter( )* and the *flush* parameter values detailed in Table 3.1.

Table 3.1. Controlling PDFlib's flushing strategy with the flush parameter

flush parameter	*flushing strategy*	*benefits*
none	*flush only once at the end of the document*	*complete PDF document can be fetched by the client in one chunk*
page	*flush at the end of each page*	*generating and fetching pages can be nicely interleaved*
content	*flush after all fonts, images, attached files, and pages*	*even better interleaving, since large items won't clog the buffer*
heavy	*always flush when the internal 32 KB document buffer is full*	*PDFlib's internal buffer will never grow beyond a fixed size*

3.1.4 Error 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, then, is to use conventional error reporting mechanisms (read: special function return codes) for function calls which may go wrong often times, and use a special exception mechanism for those rare occasions which don't warrant cluttering the code with conditionals. This is exactly the path that PDFlib goes: Some operations can be expected to go wrong rather frequently, for example:

- Trying to open an output file for which one doesn't have permission
- Using a font for which metrics information cannot be found
- Trying to open a corrupt image file

PDFlib signals such errors by returning a special value (usually –1) as documented in the API reference.

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

- running out of virtual memory
- not adhering to programming restrictions (e.g., closing a document before opening it)
- supplying wrong parameters to PDFlib API functions (e.g., trying to draw a circle with a negative radius, or supplying NULL pointers for required string arguments)

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

Obviously, the appropriate way to deal with an error heavily depends on the language used for driving PDFlib. For this reason, details on error handling are given in the language-specific sections in Chapter 2. Generally, we let C and C++ clients decide what to do by installing a custom error handler in PDFlib, or propagate the error to the language's native exception handling mechanism (all other language bindings). In the case of native language exceptions, the library client has the choice of catching exceptions and appropriately dealing with them, using the means of the respective language. The

implementation of raising exceptions is obviously language-specific, and part of the wrapper code.

For C and C++ clients which chose to not install their own error handler, the default action upon exceptions is to issue an appropriate message on the standard output channel, and exit on fatal errors. The PDF output file will be left in an inconsistent state! Since this may not be adequate for a library routine, for serious PDFlib projects it is strongly advised to leverage PDFlib's error handling facilities. A user-defined error handler may, for example, present the error message in a GUI dialog box, and take other measures instead of aborting. More information on implementing a custom error handler (for C and C++) and catching exceptions (for other language bindings) can be found in Chapter 2.

Runtime errors in PDFlib applications fall into one of several categories as shown in Table 3.2. The error handler will receive the type of PDFlib error along with a descriptive message as arguments, and present it to the user (for most language bindings), or perform custom operations if a user-supplied error handler was installed (for C and C++).

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

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

The suggested strategy is to enable warnings during the development cycle (and closely examine possible warnings), and disable warnings in a production system.

Table 3.2. PDFlib runtime errors

error name	***decimal value***	***explanation***
MemoryError	*1*	*not enough memory*
IOError	*2*	*input/output error, e.g. disk full*
RuntimeError	*3*	*wrong order of PDFlib function calls*
IndexError	*4*	*array index error*
TypeError	*5*	*argument type error*
DivisionByZero	*6*	*division by zero*
OverflowError	*7*	*arithmetic overflow*
SyntaxError	*8*	*syntactical error*
ValueError	*9*	*a value supplied as argument to PDFlib is invalid*
SystemError	*10*	*PDFlib internal error, or configuration problem (e.g., version mismatch)*
NonfatalError	*11*	*A non-fatal problem was detected. Non-fatal errors (warnings) can be suppressed using PDF_set_parameter( ).*
UnknownError	*12*	*other error*

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 in PDF lingo) has the origin in the lower left corner of the page, and uses the DTP point as unit:

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

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

Note *Hypertext functions, such as those for creating text annotations, links, and file annotations are not affected by user space transformations, and always use the default coordinate system instead.*

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

Table 3.3. Minimum and maximum page sizes supported by Acrobat 3 and 4

Acrobat viewer	minimum page size	maximum page size
Acrobat 3	1" = 72 pt = 2.54 cm	45" = 3240 pt = 114.3 cm
Acrobat 4	1/24" = 3 pt = 0.106 cm[1]	200" = 14400 pt = 508 cm

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

Common standard page size dimensions can be found in Table 3.4.[1] C macro definitions for these formats are available in *pdflib.h*.

Table 3.4. Common standard page size dimensions

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

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

3.2.2 Paths and Color

Graphics paths. A path is a shape made of an arbitrary number of straight lines, rectangles, or curves. A path may consist of several disconnected sections. Paths may be stroked or filled, or used for clipping. Stroking draws a line along the path, using client-supplied parameters for drawing. Filling paints the entire region enclosed by the path, using client-supplied parameters. The interior is determined by one of two algorithms. Clipping reduces the imageable area by replacing the current clipping area (which is the page size by default) with the intersection of the current clipping area and the path.

It's important to understand that 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.

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

Color. PDFlib clients may specify the colors used for filling and stroking the interior of paths and text characters. Colors may be specified as gray values between 0 and 1, or as RGB triples, i.e., three values between 0 and 1 specifying the percentage of red, green, and blue. The default value for stroke and fill colors is black, i.e. *(0, 0, 0)*.

3.2.3 Ordering constraints

Ordering constraints for path functions. For the sake of efficiency, PDF page descriptions must obey certain restrictions related to the ordering of path description, building, and using the path. In particular, none of the following functions must be used between the beginning of a path (i.e., one of the functions listed in Section 4.4.3, »Path Segment Functions«) and its natural demise (i.e., one of the functions listed in Section 4.4.4, »Path Painting and Clipping Functions«):

- all functions listed in Section 4.4.1, »General Graphics State Functions« (e.g., changing line width or linecap)
- all functions listed in Section 4.4.2, »Special Graphics State Functions«
- all functions listed in Section 4.5, »Color Functions« (e.g., changing the fill or stroke color)

These rules may easily be summarized as »don't change the appearance within a path description«.

3.3 Text Handling

3.3.1 The PDF Core Fonts

PDF viewers support a core set of 14 fonts which need not be embedded in any PDF file. Even when a font isn't embedded in the PDF file, PDF and therefore PDFlib need to know about the width of individual characters. For this reason, metrics information for the core fonts is already built into the PDFlib binary. However, the builtin metrics information is only available for the native host encoding (see below). Using another encoding than the host encoding requires metrics information files. Metrics files for the PDF core fonts are included in the PDFlib distribution in order to make it possible to use encodings other than the host encoding. The core fonts are the following:

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

3.3.2 Character Sets and predefined 8-Bit Text Encodings

PDF supports flexible encodings (tables which map numerical values to glyphs) for 8-bit text fonts. PDFlib includes provisions for supporting diverse encoding vectors for dealing with text. The predefined encoding vectors are referred to via symbolic names. Table 3.5 lists the symbolic encoding names supported by PDFlib (they are described in more detail below). The supported encodings can be arbitrarily mixed in one document. You may even use different encodings for a single font, although the need to do so will only rarely arise.

Table 3.5. Predefined character encodings supported by PDFlib

encoding	***description***
winansi	*Windows codepage 1252, a superset of ISO 8859-1*
macroman	*Mac Roman encoding, i.e., the default Macintosh character set*
ebcdic	*EBCDIC codepage 1047 as used on IBM S/390 systems*
builtin	*Original encoding used by non-text (symbol) or non-latin text fonts*
host	*macroman on the Mac, ebcdic on EBCDIC-based systems, and winansi on all others*

The winansi encoding. This encoding reflects the Windows ANSI character set, more specifically codepage 1252 including the three characters which Microsoft added for Windows 98 and Windows 2000 (*Euro*, *Zcaron*, and *zcaron*). The exact definition can be found in the C header files in the PDFlib source file set, or in the PDF specification [1]. The *winansi* encoding is a superset of ISO 8859-1 (Latin-1) and can therefore also be used on Unix systems.

Note Most PostScript fonts do not yet contain the three additional characters. They are supported by the core fonts in Acrobat 4, however.

The macroman encoding. This encoding reflects the MacOS character set, albeit with the old currency symbol at position 219, and not the Euro character as redefined by Apple (this incompatibility is dictated by the PDF specification). Also, this encoding does not include the Apple glyph and the mathematical symbols as defined in the MacOS character set. The exact definition can be found in the C header files in the PDFlib source file set, or in the PDF specification [1].

The ebcdic encoding. This encoding relates to the EBCDIC *(Extended Binary Coded Decimal Interchange Code)* defined by IBM and used on the IBM AS/400, S/390, and other midrange and mainframe systems. More specifically, PDFlib's *ebcdic* encoding uses the EBCDIC codepage 1047. As with all other PDFlib encodings, *ebcdic* encoding is always available for generating PDF output, and not only on native EBCDIC machines. The difference, however, is that on those machines the built-in metrics for the core fonts are sorted according to *ebcdic* encoding, and that *host* encoding (see below) also relates to *ebcdic* encoding.

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

```
EncodingScheme FontSpecific
```

while latin-text fonts will usually have the entry

```
EncodingScheme AdobeStandardEncoding
```

Fonts with the Adobe StandardEncoding can be reencoded to *winansi*, *macroman*, or *ebcdic* encodings, while fonts with *FontSpecific* encoding can't, and must use *builtin* encoding instead. PDFlib issues a warning message when an attempt is made to reencode symbol fonts.

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

The host encoding. Like *builtin*, the *host* encoding plays a special role since it doesn't refer to some fixed character set. Instead, *host* encoding will be mapped to *macroman* on the Mac, *ebcdic* on EBCDIC-based systems, and *winansi* on all others. The *host* encoding is primarily useful as a vehicle for writing platform-independent test programs (like those contained in the PDFlib distribution) or other encoding-wise simple applications. Assuming that PDFlib client programs are always encoded in the host's native encoding, such programs will always generate PDF text output with the »correct« encoding. Contrary to all other aspects of PDFlib, the concept of a *host* encoding is extremely non-portable. For this reason *host* encoding is not recommended for more sophisticated applications.

3.3.3 Custom Text Encodings

In addition to a number of predefined encodings (see Section 3.3.2, »Character Sets and predefined 8-Bit Text Encodings«) PDFlib supports user-defined 8-bit encodings in order to make PDFlib's font handling even more flexible. User-defined encodings are the way to go if you want to deal with some character set which is not internally available in PDFlib, such as ISO 8859-2 (Latin 2) or EBCDIC codepages different from the one supported internally in PDFlib. The following steps must be followed before a user-defined encoding can be leveraged in a PDFlib program:

- Generate a description of the encoding in a simple text format.
- Configure the encoding in the PDFlib resource file (see Section 3.3.7, »Resource Configuration and the UPR Resource File«).
- Provide a font (metrics and possibly outline file) that supports all characters used in the encoding. Of course, the characters in the font must use the correct PostScript glyph names as defined in the encoding table.

The encoding file simply lists glyph names and numbers line by line. As an example, the following excerpt shows the encoding definition for the ISO 8859-2 (Latin 2) encoding, also known as Windows codepage 1250:

```
% Encoding definition for PDFlib
% ISO 8859-2 (Latin-2)
% The Latin 2 character set supports the Slavic languages
% of Central Europe which use the Latin alphabet.

space       32  % 0x20
exclam      33  % 0x21
quotedbl    34  % 0x22
numbersign  35  % 0x23
...more glyph assignments...
udieresis   252 % 0xFC
yacute      253 % 0xFD
tcedilla    254 % 0xFE
dotaccent   255 % 0xFF
```

More formally, the contents of an encoding 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 a PostScript character name, followed by whitespace and a decimal character code in the decimal range 1–255. PDFlib does not support the use of the null character (code position 0) in user-defined encodings.
- Character codes which are not mentioned in the encoding file are assumed to be undefined.

The relationship between the name of the encoding file and the name of the actual encoding (to be used with *PDF_findfont( )*) is specified in PDFlib's resource file (see Section 3.3.7, »Resource Configuration and the UPR Resource File«). Sample encoding files are supplied with PDFlib.

Note *Similar to metrics information, custom encodings can be compiled into the PDFlib binary in order to improve performance.*

Finding character names. In order to write a custom encoding file you will have to find information about the exact definition of the character set to be defined by the encoding, as well as the exact glyph names used in the font files.[1] You must also ensure that a chosen font provides all necessary characters for the encoding. For example, the core fonts supplied with Acrobat do not support ISO 8859-2 (Latin 2). If you happen to have the FontLab[2] font editor (by the way, a great tool for dealing with all kinds of font and encoding issues), you may use it to find out about the encodings supported by a given font (look for »codepages« in the FontLab documentation). Actually, you may even use FontLab's encoding files since these are compatible with PDFlib's.

For the convenience of PDFlib users, the PostScript program *print_glyphs.ps* in the distribution fileset can be used to find the names of all characters contained in a font. In order to use it, enter the name of the font at the end of the PostScript file and send it

1. Useful raw material for writing encoding tables for a variety of standards and vendor-specific character sets can be found at ftp://ftp.unicode.org/Public/MAPPINGS; Information about the glyph names used in PostScript fonts can be found at http://partners.adobe.com/asn/developer/typeforum/unicodegn.html (although font vendors are not required to follow these recommendations).
2. See http://www.fontlab.com

(along with the font) to a PostScript Level 2 or 3 printer, or view it with a Level-2-compatible PostScript viewer. The program will print all characters in the font, sorted alphabetically by glyph name.

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

3.3.4 8-bit Encodings and Unicode-aware Language Bindings

The preceding discussion relates to 8-bit encodings only. PDFlib also supports 16-bit Unicode encoding for hypertext elements as discussed in Section 3.3.9, »Unicode Support«, and a wide range of locale-specific encodings for CJK fonts (see Section 3.3.8, »CID Font Support for Japanese, Chinese, and Korean Text«). This may give rise to problems in language bindings which are natively Unicode-aware (currently ActiveX, Java, and Tcl). In these environments strings passed to PDFlib functions may contain Unicode characters although the currently selected encoding is not Unicode-compatible, but one of the 8-bit encodings discussed in the preceding sections. Since a reasonable mapping is not possible in such cases, it is an error to pass Unicode characters to *PDF_show( ), PDF_show_xy( ),* and *PDF_continue_text( )* unless a Unicode-compatible CMap has been selected. (*PDF_show_boxed( )* doesn't process CID fonts at all, and cannot be used with Unicode encodings.)

3.3.5 Hypertext Encoding

PDF supports two methods for encoding hypertext elements such as bookmarks, annotations, and document information fields. Up to Acrobat 3, all hypertext strings had to be encoded with a special 8-bit encoding called PDFDocEncoding (PDFDocEncoding can not be used for text used on page descriptions). Starting with Acrobat 4, Unicode strings can be used for all hypertext elements. For more information on Unicode see Section 3.3.9, »Unicode Support«.

PDFDocEncoding (see Figure 3.1) is a superset of ISO 8859-1 (Latin 1) and therefore contains all ASCII characters in the lower part. Although PDFDocEncoding and the Windows codepage 1252 are quite similar, they differ substantially in the character range 0x80 – 0xA0.

Many clients will be able to directly use PDFDocEncoding. However, since the Mac encoding substantially differs from PDFDocEncoding, it is necessary to convert Mac strings to PDFDocEncoding when it comes to hypertext elements, and non-ASCII special characters are to be used. Mac special characters must be converted to Unicode before they can be used in hypertext elements. This conversion must be performed by the client.

Note Hypertext strings will automatically be converted to PDFDocEncoding on EBCDIC systems.

3.3.6 Font Outline and Metrics Files

PDF font embedding. PDF supports fonts outside the set of 14 core fonts in several ways. PDFlib is capable of embedding PostScript type 1 font descriptions into the generated PDF output. Alternatively, a font descriptor consisting of the character metrics and some general information about the font (without the actual character outline data) can be embedded. If a font is not embedded in a PDF document, Acrobat will take it from

Fig. 3.1. The PDFDocEncoding character set as defined in PDF 1.3 with hex and octal codes. Note the Euro character at position hexadecimal A0 = octal 240.

the target system if available, or construct a substitute font according to the font descriptor in the PDF. Table 3.6 lists different situations with respect to font usage, each of which poses different requirements on the necessary font and metrics files.

PDFlib supports the following file formats for font metrics and outline data on all platforms:

- The platform-independent AFM (Adobe Font Metrics) and the Windows-specific PFM (Printer Font Metrics) format for metrics information. Since PFM files do not describe the full character metrics but only the glyphs used in Windows (codepage 1252), they can only be used for the *winansi* or *builtin* encodings, while AFM-based font metrics can be rearranged to any encoding supported by the font.
- The platform-independent PFA (Printer Font ASCII) and the Windows-specific PFB (Printer Font Binary) format for PostScript Type 1 font outline information.

If you can get hold of a font file, but not the corresponding metrics file, you can try to generate the missing metrics using one of several freely available utilities. For example, the T1lib package[1] contains the *type1afm* utility for generating AFM metrics from PFA or PFB font files.

1. See http://www.neuroinformatik.ruhr-uni-bochum.de/ini/PEOPLE/rmz/t1lib/t1lib.html

Table 3.6. Different font usage situations and required metrics and outline files

font usage	*font metrics file required?*	*font outline file required?*
One of the 14 core fonts with PDFlib's host encoding[1,2]	*no*	*no*
One of the 14 core fonts with an encoding other than PDFlib's host encoding[2]	*yes (AFM files supplied with the PDFlib distribution)*	*no*
Non-core font which will not be embedded	*yes*	*no*
Non-core font which will be embedded	*yes*	*yes*
Additional font/encoding combinations for which the metrics have been compiled into PDFlib (see below)	*no*	*yes, if font embedding is requested*
Standard CID fonts[3]	*no*	*no*
Non-standard CID fonts	*(not supported)*	*(not supported)*

1. See Section 3.3.1, »The PDF Core Fonts« for a list of core fonts.
2. See Section 3.3.2, »Character Sets and predefined 8-Bit Text Encodings« for the definition of PDFlib's host encoding.
3. See Section 3.3.8, »CID Font Support for Japanese, Chinese, and Korean Text« for more information on CID fonts.

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

Performance notes. It is important to be aware of the impact of font handling issues on PDFlib's performance. Generally, the font metrics (either in-core or on file) are accessed whenever a certain font/encoding combination is used for the first time. Subsequent requests for the same combination will be satisfied from PDFlib's internal font cache without any further performance penalty. Regarding font handling performance, the following observations may be useful:

- Due to their small size and binary nature, PFM metrics files can be read much faster than the text-based AFM metrics files. However, they cannot be used for arbitrary encodings.
- AFM files contain much useful information about many aspects of font usage, and can be used for arbitrary encodings. However, although only the bare character metrics are required for PDFlib, the complete AFM file must be parsed in a time-consuming manner. For performance-critical applications it might be worthwhile to strip the unneeded data (e.g., the kerning information) from the AFM file. The AFM files supplied with PDFlib have been stripped in this way.
- For specific applications the performance can be improved very much by compiling the metrics information for the required font/encoding combinations into the PDFlib binary, thereby obviating the use for external metrics files at all. The *compile_metrics* utility supplied with PDFlib can be used for constructing a C header file with the required data. (The default metrics data built into PDFlib have also been generated with this utility.) *compile_metrics* requires re-compiling the PDFlib binary and is therefore only useful to C or C++ developers.

Use of TrueType fonts in PDF. Although PDF technically supports embedded TrueType fonts, this is a murky area of PDF technology. The internals of TrueType and their behavior in PDF are not clearly documented, which in practise gives rise to several problems (such as missing characters or unsearchable text).

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

Legal aspects of font embedding. It's important to note that mere possession of a font file may not justify embedding the font in PDF, even for holders of a legal font license. Many font vendors impose restrictions on the use of their fonts for non-print usage. Some type foundries completely forbid PDF font embedding, others offer special online or embedding licenses for their fonts, while still others allow font embedding provided the fonts are subsetted. Please check the legal implications of font embedding before attempting to embed fonts with PDFlib.

Note PDFlib currently doesn't implement font subsetting.

3.3.7 Resource Configuration and the UPR Resource File

In order to make PDFlib's font handling platform-independent and customizable, a configuration file can be supplied for describing the available fonts along with the names of their outline and metrics files, and the names of additional encoding files. In addition to the static configuration file, dynamic configuration can be accomplished at runtime by adding resources with *PDF_set_parameter( )*. For the configuration file we dug out a simple text format called »Unix PostScript Resource« (UPR) which came to life in the era of Display PostScript. However, we will take the liberty of extending the original UPR format for our purposes. The UPR file format as used by PDFlib will be described below.[1] There is an Adobe-supplied utility called *makepsres* floating around the Internet which can be used to automatically generate UPR files from PostScript font outline and metrics files.

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

- Lines can have a maximum of 255 characters.
- A backslash '\' escapes any character, including newline characters. This may be used to extend lines.
- The period character ' . ' serves as a section terminator, and must therefore be escaped when used at the start of any other line.
- All entries are case-sensitive.
- Comment lines may be introduced with a percent '%' character, and terminated by the end of the line.
- Whitespace is ignored everywhere.

UPR files consist of the following components:

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

```
PS-Resources-1.0
```

1. For those interested, the complete specification can be found in the book »Programming the Display PostScript System with X« (Appendix A), available at http://partners.adobe.com/asn/developer/PDFS/TN/DPS.refmanuals.TK.pdf

- A section listing all types of 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. This section exists for compatibility only, and is ignored by PDFlib.
- The optional directory line may be used as a shortcut for a directory prefix common to all resource files described in the file. The prefix will be added to all file names given in the UPR file. If present, the directory line starts with a slash character, immediately followed by the directory prefix. Note that the initial slash character is required on all platforms, and is not part of the directory name. Using the directory prefix a UPR file may, for example, point to some central PostScript font directory somewhere in the file system.

Note *Do not use the directory prefix for the (completely installed) PDFlib ActiveX component since it substitutes the UPR directory prefix mechanism with Windows registry entries.*

- A section for each of the resource categories listed at the beginning of the file. Each section starts with a line showing the resource category, followed by an arbitrary number of lines describing available resources. The list is terminated by a line with a single period character. Each resource data line contains the name of the resource (equal signs have to be quoted), an equal sign, and the corresponding relative or absolute file name for the resource. Relative file names will have the directory prefix applied, if one is present in the file. Using a double equal sign forces the file name to be interpreted absolute, i.e., the prefix is not used.

Supported resource categories. The resource categories currently supported in PDFlib are listed in Table 3.7. Other resource categories may be present in the UPR file for compatibility with Display PostScript installations, but they will silently be ignored.

Table 3.7. Resource categories supported in PDFlib

resource type name	*explanation*
FontAFM	*PostScript font metrics file in AFM format*
FontPFM	*PostScript font metrics file in PFM format*
FontOutline	*PostScript font outline file in PFA or PFB format*
Encoding	*Text file containing an 8-bit encoding table*

Sample UPR file. The following listing gives an example of a UPR configuration file as used by PDFlib. It describes the 14 PDF core fonts' metrics, plus metrics and outline files for two extra fonts, plus a custom encoding:

```
PS-Resources-1.0
FontAFM
FontPFM
FontOutline
Encoding
.
% Directory prefix example for Windows: /c:/psfonts
//usr/local/lib/fonts
FontAFM
Code-128=Code_128.afm
Courier=Courier.afm
Courier-Bold=Courier-Bold.afm
```

```
Courier-BoldOblique=Courier-BoldOblique.afm
Courier-Oblique=Courier-Oblique.afm
Helvetica=Helvetica.afm
Helvetica-Bold=Helvetica-Bold.afm
Helvetica-BoldOblique=Helvetica-BoldOblique.afm
Helvetica-Oblique=Helvetica-Oblique.afm
Symbol=Symbol.afm
Times-Bold=Times-Bold.afm
Times-BoldItalic=Times-BoldItalic.afm
Times-Italic=Times-Italic.afm
Times-Roman=Times-Roman.afm
ZapfDingbats=ZapfDingbats.afm
.
FontPFM
Foobar-Bold=foobb___.pfm
.
FontOutline
Code-128=Code_128.pfa
.
Encoding
Latin-2=latin-2.enc
.
```

Searching for the UPR resource file. If only the built-in resources are to be used (PDF core fonts with host encoding), a UPR configuration file is not required, since PDFlib contains all necessary resources.

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

- The PDFlib ActiveX component checks a registry entry to find the file *pdflib.upr* in the *fonts* subdirectory of the PDFlib installation directory.
- On Unix and Windows systems, the environment variable *PDFLIBRESOURCE* is examined and used as a resource file name.
- If no file name is found, the client-settable *resourcefile* parameter (which may be set using *PDF_set_parameter( )*) is examined and used as a resource file name, if set.
- If no file name is found, the file name *pdflib.upr* in the current directory is used.
- If this file can't be opened, an *IOError* is raised.
- If a resource file can be opened during any of the above steps, but a required resource category cannot be found, a *SystemError* is raised.

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

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

3.3.8 CID Font Support for Japanese, Chinese, and Korean Text

CJK support in Acrobat and PDF[1]. While Japanese font support was already available in Acrobat 3J, Acrobat 4 added full support for CID (Character ID) fonts[2] for Japanese, Chinese, and Korean (CJK) text even in the non-Japanese versions of the full Acrobat package as well as the free Acrobat Reader. In order to use CJK documents in Acrobat you must do one of the following:

- Use a localized CJK version of Acrobat.
- If you use any non-CJK version of the full Acrobat 4 product, select the Acrobat installer's option »Asian Language Support« (Windows) or »Language Kit« (Mac). The required CJK fonts will be installed from the Acrobat 4 product CD-ROM.
- If you're using Acrobat Reader 4, install one of the Asian Font Packs which are available on the Acrobat 4 product CD-ROM, or on the Web.[3]

CJK encodings and fonts. Historically, a wide variety of CJK encoding schemes has been developed by diverse standards bodies, companies, and other organizations. Fortunately enough, all prevalent encodings are supported by Acrobat and PDF by default. Acrobat 4 supports a wealth of different encoding schemes for CJK fonts. Since the concept of an encoding is much more complicated for CJK text than for Latin text, simple encoding vectors with 256 entries no longer suffice. Instead, PostScript and PDF use the concept of character collections and character maps (CMaps) for organizing the characters in a font. Conceptually, CMaps can be thought of as large encodings for CJK fonts.

Acrobat 4 supports a set of standard fonts for CJK text. These fonts are supplied with the Acrobat installation (or the Asian FontPack), and therefore don't have to be embedded in the PDF file (this parallels the use of the 14 core fonts for Latin text). These fonts contain all characters required for common encodings, and support both horizontal and vertical writing modes. The standard fonts and CMaps are documented in Table 3.8. As can be seen from the table, the default CMaps support most CJK encodings used on Mac, Windows, and Unix systems, as well as several other vendor-specific encodings. Detailed descriptions of the character encoding standards which are supported by these CMaps can be found in the PDF specification [1].

CJK font support in PDFlib. Having realized the similarity between core fonts/encoding vector on the one hand, and CJK standard fonts/CMaps on the other hand, it won't be much of a surprise that both Latin and CJK fonts can be chosen with the same PDFlib interface, using the CMap name in lieu of the encoding name, and taking into account that a given CJK font supports only a certain set of CMaps (see Table 3.8).

Let's take a look at an example:

```
font = PDF_findfont(p, "HeiseiMin-W3", "UniJIS-UCS2-H", 0)
```

This instruction locates one of the Japanese standard fonts, choosing the CMap for a Unicode-compatible *(UniJIS)* 2-byte encoding *(UCS2)* and horizontal writing mode (*H*). The *fontname* parameter must be the exact name of the font (strictly speaking, the value of the */CIDFontName* entry in the corresponding CID PostScript font file), without any

1. This is a good opportunity to praise Ken Lunde's seminal tome »CJKV information processing – Chinese, Japanese, Korean & Vietnamese Computing« (O'Reilly 1999, ISBN 1-56592-224-7), as well as his work at Adobe since he's one of the driving forces behind CJK support in PostScript and PDF.
2. See http://partners.adobe.com/asn/developer/typeforum/cidfonts.html for a wealth of resources related to CID fonts.
3. See http://www.adobe.com/prodindex/acrobat/cjkfontpack.html

Table 3.8. Predefined CMaps and Acrobat's default fonts for Japanese, Chinese, and Korean text

locale	*available standard fonts in Acrobat*	*supported CMaps (encodings)*
Simplified Chinese	*STSong-Light*	*GB-EUC-H, GB-EUC-V, GBpc-EUC-H, GBpc-EUC-V, GBK-EUC-H, GBK-EUC-V, UniGB-UCS2-H[1], UniGB-UCS2-V[1]*
Traditional Chinese	*MHei-Medium MSung-Light*	*B5pc-H, B5pc-V, ETen-B5-H, ETen-B5-V, ETenms-B5-H, ETenms-B5-V, CNS-EUC-H, CNS-EUC-V, UniCNS-UCS2-H[1], UniCNS-UCS2-V[1]*
Japanese	*HeiseiKakuGo-W5, HeiseiMin-W3*	*83pv-RKSJ-H, 90ms-RKSJ-H, 90ms-RKSJ-V, 90msp-RKSJ-H, 90msp-RKSJ-V, 90pv-RKSJ-H, Add-RKSJ-H, Add-RKSJ-V, EUC-H, EUC-V, Ext-RKSJ-H, Ext-RKSJ-V, H, V, UniJIS-UCS2-H[1], UniJIS-UCS2-V[1], UniJIS-UCS2-HW-H[1], UniJIS-UCS2-HW-V[1]*
Korean	*HYGoThic-Medium, HYSMyeongJo-Medium*	*KSC-EUC-H, KSC-EUC-V, KSCms-UHC-H, KSCms-UHC-V, KSCms-UHC-HW-H, KSCms-UHC-HW-V, KSCpc-EUC-H, UniKS-UCS2-H[1], UniKS-UCS2-V[1]*

1. Unicode-compatible CMap

encoding or writing mode suffixes. The *encoding* parameter is the name of one of the supported CMaps (the choice depends on the font) and will also indicate the writing mode (*see below*). PDFlib supports all of Acrobat 4's default CMaps, and will complain when it detects a mismatch between the requested font and the CMap. For example, asking PDFlib to use a Korean font with a Japanese encoding will result in an exception of type *PDF_ValueError*.

Although CID font embedding is technically possible in PDF 1.3, it is not practical due to the size of typical CID fonts, and due to the fact that most CJK font licenses do not permit embedding. For this reason the embedding parameter is not used for CID fonts, and must be 0.

PDFlib doesn't require any font-specific metrics information for CID fonts, and doesn't make any attempt to decode the client-supplied text strings, or verify whether they are correctly encoded with respect to the underlying CMap. For this reason calculating the extent of text with *PDF_stringwidth( )* and activating underline/overline/strikeout mode are not supported for CID fonts.

Note PDFlib currently only supports the standard CID fonts supplied with Acrobat (see Table 3.8). Neither custom CID fonts nor CJK TrueType fonts can be used. However, you can simulate bold fonts by rendering »fill and stroke« text (rendering mode 2, see textrendering parameter).

Horizontal and vertical writing mode. PDFlib supports both horizontal and vertical writing modes. The mode is selected along with the encoding by choosing the appropriate CMap name. CMaps with names ending in *-H* select horizontal writing mode, while the *-V* suffix selects vertical writing mode.

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

CJK text encoding in PDFlib. The client is responsible for supplying text such that its encoding matches the encoding requested for the CID font. PDFlib does not check whether the supplied text conforms to the requested encoding. Since several of the sup-

ported encodings may contain null characters in the text strings, C and C++ developers must take care not to use the *PDF_show( )* etc. functions, but instead *PDF_show2( )* etc. which allow for arbitrary binary strings along with a length parameter. For all other bindings, the text functions support binary strings, and *PDF_show2( )* etc. are not required. For multi-byte encodings, the high-order byte of a character must appear first.

PDFlib language bindings which are natively Unicode-aware automatically convert Unicode strings supplied to the library. For this reason only Unicode-compatible CMaps should be used with these language bindings (see also Section 3.3.9, »Unicode Support«).

Printing PDF documents with CJK text. Printing CJK documents gives rise to a number of issues which are outside the scope of this manual. However, we will supply some useful hints for the convenience of PDFlib users. If you have trouble printing CJK documents with Acrobat, consider one or more of the following:

- Printing CID fonts does not work on all PostScript printers. Native CID font support has only been integrated in PostScript version 2015, i.e. PostScript Level 1 and early Level 2 printers do not natively support CID fonts (unless the printer is equipped with the Type 0 font extensions). However, for early Level 2 devices the printer driver is supposed to take care of this by downloading an appropriate set of compatibility routines to pre-2015 Level 2 printers.
- Due to the large number of characters CID fonts consume very much printer memory (disk files for CID fonts typically are 5–10 MB in size). Not all printers have enough memory for printing such fonts. For example, in our testing we found we had to upgrade a Level 3 laser printer from 16 MB to 48 MB RAM in order to reliably print PDF documents with CID fonts.
- Non-Japanese PostScript printers do not have any Japanese fonts installed. For this reason, you must check »Download Asian Fonts« in Acrobat's print dialog.
- If you can't successfully print using downloaded fonts, check »Print as Image« in Acrobat's print dialog. This instructs Acrobat to send a bitmapped version of the page to the printer (300 dpi, though).

3.3.9 Unicode Support

Starting with version 4, Acrobat fully supports the Unicode standard. This is a large character set which covers all current and many ancient languages and scripts in the world, and has significant support in many applications and operating systems.[1] PDFlib supports the Unicode standard for the following features:

- bookmarks (see Figure 3.2)
- contents and title of note annotations (see Figure 3.2)
- standard and user-defined document information field contents (but not user-defined field names)
- description and author of file attachments
- CJK text on page descriptions, provided a Unicode-compatible encoding is used (see Section 3.3.8, »CID Font Support for Japanese, Chinese, and Korean Text«)

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

1. See http://www.unicode.org for more information about the Unicode standard

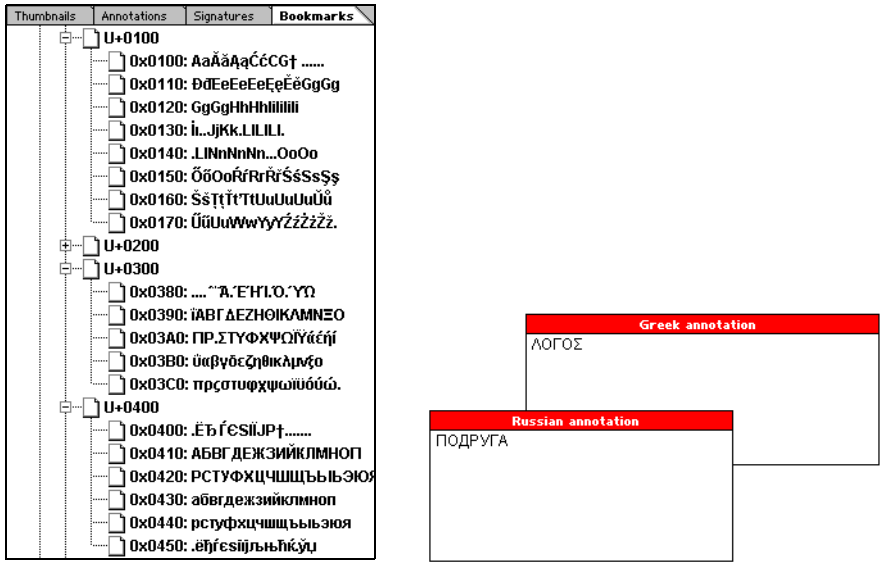


Fig. 3.2. Unicode bookmarks (left) and Unicode text annotations (right)

- Unicode support for the actual page descriptions is only available for CJK fonts.
- The usability of Unicode-enhanced PDF documents heavily depends on the Unicode support available on the target system. Unfortunately, most systems today are far from being fully Unicode-enabled in their default configurations. Although Windows NT and MacOS support Unicode internally, availability of appropriate Unicode fonts is still an issue.
- Acrobat on Windows is unable to handle more than one script in a single annotation. This seems to be related to an OS-specific issue (restrictions of the text edit widget used in Acrobat's implementation of the annotation feature).

Unicode encoding for hypertext elements. PDFlib supports a dual-encoding approach with respect to all text supplied by the client for one of the Unicode-enabled hypertext functions. PDF expects Unicode hypertext according to the following rules (these are also known as big-endian UTF-16 serialization with signature):

- In order to distinguish »regular« 8-bit encoded text strings from 16-bit Unicode strings, the Unicode Byte Order Mark (BOM) is used as a sentinel at the beginning of the string. The BOM consists of the following two byte values which must be the first 16-bit character in all Unicode strings for hypertext:

  ```
  hex (FE, FF) = octal (376, 377)
  ```

- Subsequent characters in the Unicode string are encoded with 2 bytes each, where the high order byte occurs first in the linear ordering (big-endian byte ordering, unlike the little-endian ordering used on Windows/Intel systems).
- Since Unicode strings may contain null characters, the usual C convention for strings cannot be used. For this reason, all non-Unicode-aware PDFlib language bindings expect Unicode strings to be terminated with a Unicode null character, i.e., two null bytes.

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

```
\0376\0377\003\233\003\237\003\223\003\237\003\243\0\0
```

or in hexadecimal notation:

```
\xFE\xFF\x03\x9B\x03\x9F\x03\x96\x03\x9F\x03\xA3\0\0
```

Clients of non-Unicode-aware language bindings (see below) must manually wrap Unicode hypertext with BOM and double-null as described above.

Unicode encoding for page descriptions. PDF allows Unicode-encoded text on document pages (as opposed to hypertext as discussed above). Unfortunately, this holds only true for CID fonts, but not regular Type 1 PostScript fonts. In order to place Unicode-conforming Chinese, Japanese, or Korean text on a page, a Unicode-compatible CMap must be used. These are easily identified by the *Uni* prefix in their name, and are marked in Table 3.8. These CMaps, however, only support the characters required for the respective locale, but not other Unicode characters.

Unicode text on page descriptions must be supplied »as is«, i.e., it must not be wrapped with BOM and double-null like hypertext. In addition, clients of the C and C++ language bindings (except when the ANSI string class is used in the latter case) must take care not to use the standard text functions *(PDF_show( )*, *PDF_show_xy( )*, and *PDF_continue_text( ))* when the text may contain embedded null characters. In such cases the alternate functions *PDF_show2( )* etc. must be used. This is not a concern for all other language bindings since the PDFlib language wrappers internally call *PDF_show2( )* etc. in the first place.

PDFlib language bindings and Unicode. The following PDFlib language bindings are fully Unicode-aware, and automatically convert Unicode strings to the format expected by PDFlib:

- ActiveX/COM
- Java
- Tcl (requires Tcl 8.2 or above)

Automatic conversion means these bindings will internally distinguish the following cases, and apply the appropriate conversion:

- 8-bit strings, i.e., characters from U+0000 to U+00FF are interpreted as PDFDocEncoding (for hypertext) or 8-bit characters according to the current encoding (for page descriptions)
- Unicode strings for hypertext functions will be wrapped with BOM and double-null
- Unicode strings for page descriptions will be supplied without any conversion. This requires a Unicode-compatible CMap to be selected with *PDF_findfont( )*.

The developer generally need not care about the encoding specifics detailed above, but can simply use Unicode text as supported by the environment. (More details on Unicode usage from within the supported languages can be found in the manual section for the respective binding in Chapter 2). However, there's a subtle issue related to literal Unicode characters embedded in ActiveX, Java, or Tcl source code which we will try to explain with a Java example.

Java's native support for Unicode strings is just fine for PDF's hypertext elements, but can be dangerous with respect to page descriptions and non-Unicode-compliant 8-bit encodings. For example, while most characters in the Windows codepage 1252 are compatible with Unicode, not all are (more specifically, the range 0x80-0x9F). Consider the following attempt to show the endash character with PDFlib's Java binding:

```
p.show("–")        // Literal character 0x96 in the code. Wrong, do not use!
```

When this snippet is compiled under Unix with the Latin-1 character set, it will work just fine. However, when it is compiled under Windows with codepage 1252, the literal endash character (0x96 in codepage 1252) will be translated to the corresponding Unicode character (0x2013 in this example), which is unsuited for a 8-bit PDF encoding such as *winansi*. In order to prevent this problem, rewrite the above code snippet as follows:

```
p.show("\u0096")    // Safe way of selecting characters outside Latin-1
```

This will pass the intended character code 0x96 to PDFlib, which will correctly interpret it according to the chosen encoding vector (although the Java compiler will be fooled into believing it deals with the *Unicode* character 0x0096, which doesn't actually exist).

3.3.10 Text Variations and Text Box Formatting

Underline, overline, and strikeout text. PDFlib can be instructed to put lines below, above, or in the middle of text. The stroke width of the bar and its distance from the baseline are calculated based on the font's metrics information. In addition, the current values of the horizontal scaling factor and the text matrix are taken into account when calculating the width of the bar. *PDF_set_parameter( )* can be used to switch the underline, overline, and strikeout feature on or off. The current stroke color is used for drawing the bars. The current linecap and dash parameters are ignored, however. Aesthetics alert: in most fonts underlining will touch descenders, and overlining will touch diacritical marks atop ascenders.

Note The underline, overline, and strikeout features are not supported for CID fonts.

Text rendering modes. PDFlib supports several rendering modes which affect the appearance of text. This includes outline text and the ability to use text as a clipping path. Text can also be rendered invisibly which may be useful for placing text on scanned images in order to make the text accessible to searching and indexing, while at the same time assuring it will not be visible directly. The rendering modes are described in Table 3.9. They can be set with *PDF_set_value( )*.

Table 3.9. Values for the text rendering mode

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

Text box formatting. While PDFlib offers the *PDF_stringwidth( )* function for performing text width calculations, many clients need easy access to text box formatting and justifying, e.g. to fit a certain amount of text into a given column. Although PDFlib offers such features, you shouldn't think of PDFlib as a full-featured text and graphics layout engine. The *PDF_show_boxed( )* function is an easy-to-use method for text box formatting with a number of formatting options. Text may be laid out in a rectangular box either left-aligned, right-aligned, centered, or fully justified. This function justifies by adjusting the inter-word spacing (the last line will be left-aligned only). However, advanced text processing features such as hyphenation are not available – PDFlib simply breaks text lines at existing whitespace characters.

ASCII newline characters *(0x0A)* in the supplied text are recognized, and force a new paragraph. CR/NL combinations are treated like a single newline character. Other formatting characters are currently not supported.

The following requirements and restrictions for PDFlib's box formatting shall be noted:

- Contiguous blanks in the text should be avoided.
- Due to restrictions in PDF's word spacing support, the space character must be available at code position 0x20 in the encoding. Although this is the case for many encodings, it implies that justification will not work with EBCDIC encoding.
- The simplistic formatting algorithm may fail for unsuitable combinations of long words and narrow columns.
- Since the bottom part of the box is used as a baseline, descenders in the last line may extend beyond the box area.
- It's currently not possible to feed the text in multiple portions into the box formatting routine. However, you can retrieve the text position after calling *PDF_show_boxed( )* with the *textx* and *texty* parameters.
- Support for text box formatting is not available for CID fonts.

3.4 Image Handling

3.4.1 Image File Formats

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

PNG images. If the freely available *libpng* library (which in turn requires the Zlib compression library) is available, PDFlib supports all flavors of PNG images (»Portable Network Graphics«).[1] If PNG images contain transparency information, the transparency is retained in the generated PDF (see Section 3.4.5, »Image Masks and Transparency«).

Note *All binary distributions of PDFlib support the PNG image file format.*

JPEG images. All versions of PDF directly support the »baseline« flavor of JPEG compression which accounts for the vast majority of available JPEG files. In addition, Acrobat 4 and PDF 1.3 support progressive JPEG compression. PDFlib correctly deals with

1. See http://www.w3.org/Graphics/PNG and http://www.cdrom.com/pub/png for more information on the PNG image file format and the libpng library

baseline and progressive JPEG images, but raises an exception for progressive images if run in Acrobat 3 compatibility mode. PDFlib also applies a workaround which is necessary for correctly processing Photoshop-generated CMYK JPEG files.

GIF images. PDFlib contains internal GIF handling code. Regular and interlaced GIFs may be used. Multi-frame GIF images are not supported.

TIFF images. Sam Leffler's TIFFlib[1] can be plugged into PDFlib in order to support many TIFF compression and encoding flavors. PDFlib tries to preserve the existing compressed data for certain kinds of TIFF compression schemes (pass-through), such as CCITT compression. TIFF images which make use of other compression schemes will first be decompressed, and then recompressed with the Flate/Zlib algorithm (if Zlib is available). Images of the first kind will result in much better performance than those of the second kind. However, PDFlib cannot check the integrity of the compressed image data in this case. Incomplete or corrupt image data may result in error or warning messages when using the PDF in Acrobat (e.g., »Read less image data than expected«).

Note All binary distributions of PDFlib support the TIFF image file format.

Note PDFlib does not support all kinds of TIFF images, e.g., color-separated images or those using the CIE color space.

CCITT images. Raw Group 3 or Group 4 fax compressed image data. Note that this actually means raw CCITT-compressed image data, *not* TIFF files using CCITT compression. Raw CCITT compressed image files are usually not supported in end-user applications, but can only be generated with specialized software.

Raw data. Uncompressed (raw) image data may be useful for some special applications, e.g., constructing a color ramp directly in memory. The nature of the image is deduced from the number of color components: 1 component implies a grayscale image, 3 components an RGB image, and 4 components a CMYK image.

If PDFlib is configured for compression, most image data will be output in the generated PDF in ZIP (Flate)-compressed format (this behavior can be changed by setting the *compress* parameter). Notable exceptions are JPEG, raw CCITT, and CCITT-compressed TIFF images, which always retain their original compression scheme.

3.4.2 Embedding Images in PDF

Embedding raster images with PDFlib is easy to accomplish. First, the image file has to be opened with a PDFlib function which does a brief analysis of the image parameters. The *PDF_open_image_file( )* function returns a handle which serves as an image descriptor. This handle can be used in a call to *PDF_place_image( )*, along with positioning and scaling parameters:

```
if ((image = PDF_open_image_file(p, "jpeg", "image.jpg", "", 0)) == -1) {
        fprintf(stderr,"Error: Couldn't read image.\n");
} else {
        PDF_place_image(p, image, 0.0, 0.0, 1.0);
        PDF_close_image(p, image);
}
```

1. See http://www.libtiff.org

The call to *PDF_close_image( )* may or may not be required, depending on whether the same image will be used again in the same document (see below).

3.4.3 Re-using Image Data

It should be emphasized that PDFlib supports an important PDF optimization technique for using repeated raster images.

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

3.4.4 Memory Images and External Image References

While the majority of image data for use with PDFlib will be pulled from some disk file on the local file system, other image data sources are also supported.

For performance reasons supplying existing image data directly in memory may be preferable over opening a disk file. PDFlib supports in-core image data for certain image file formats.

PDFlib also supports an experimental feature which isn't recommended for general-use PDF files, but may offer advantages in certain environments. While almost all PDF documents are completely self-contained (the only exception being non-embedded fonts), it is also possible to store only a reference to some external data source in the PDF file instead of the actual image data, and rely on Acrobat to fetch the required image data when needed. This mechanism works similar to the well-known image references in HTML documents. Usable external image sources include data files in the local file system, and URLs. It is important to note that while file references work in Acrobat 3 and 4, URL references only work in Acrobat 4 (full product). PDF documents which include image URLs are neither usable in Acrobat 3 nor Acrobat Reader 4!

The *PDF_open_image( )* interface can be used for both in-memory image data and external references.

3.4.5 Image Masks and Transparency

Transparency in PDF. Transparency has been missing from PostScript and PDF for quite a long time. Only with PDF 1.3 (and PostScript 3) Adobe integrated some limited support for transparency into languages and applications. While image masks (painting solid color through a bitmap mask) are an old feature of both PostScript and PDF, Acrobat 4 added the feature of masking particular pixels of an image. This offers the following opportunities:

- Masking by position: an image may carry the intrinsic information »print the foreground only, but not the background«. This is often used in catalog images.
- Masking by color value: pixels of a certain color (or from a color range – but not arbitrary sets of colors) are not painted, but the previously painted part of the page shines through instead. In TV and video technology this is also known as bluescreen-

ing, and is most often used for combining the weather man and the map into one image.

It is important to note that PDF supports binary transparency only: there is no alpha channel or variable opacity (»blend this image with the background«) but only a binary decision (»print either the image pixel, or the background pixel«). Binary transparency may be considered »poor man's alpha channel«. Another important restriction is that in PDF the mask is always attached to the image; it's not possible to use an image first with a mask, and the same image a second time without a mask, or with a different mask.

Viewing and printing PDF files with transparency. Equally important as PDF's intrinsic limitations with respect to transparency are the practical limitations when it comes to using PDF files with transparency in the viewer application. The following restrictions should be noted:

- Transparency only works in PDF 1.3/Acrobat 4 – older viewers will completely ignore transparency information, and display or print the whole image (overpainting the background).
- Printing transparent images to PostScript Level 1 or 2 doesn't work, even with Acrobat 4 (since transparency support only appeared in PostScript 3, and can't easily be emulated). Acrobat prints the base image without the mask.
- If an image is masked by position Acrobat 4 viewers will only honour the clipping up to a certain image size, and display the whole image otherwise. It appears from experimentation that the following limit applies:

  ```
  width x height x components < 1024 K
  ```

  Images above this limit are displayed without applying the mask. The limit in a typical PostScript 3 printer seems to be lower, resulting in PostScript errors when trying to print PDF documents with large masked images.
- Ghostscript 6.0 does not support masked images in PDF.

Transparency support in PDFlib. PDFlib supports both masking by position and by color value (only single color values, but no ranges). Transparency information can be applied implicitly or explicitly.

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

- GIF image files may contain a single transparent color value which is respected by PDFlib.
- PNG image files may contain several flavors of transparency information, or a full alpha channel. PDFlib tries to preserve as much as possible from this information: single transparent color values are retained; if multiple color values with an attached alpha value are given, only the first one with an alpha value below 50 percent is used; a full alpha channel is ignored.

The explicit case requires two steps, both of which involve image operations. First, an image must be prepared for later use as a binary transparency mask. This is accomplished by using the standard image file function with an additional parameter:

```
mask = PDF_open_image_file(p, "png", filename, "mask", 0)
```

In order to be usable as a mask, an image must have only a single color component and a bit depth of 1, i.e., only plain bitmaps are suitable as a mask. This rules out JPEG, GIF, and TIFF image files (which either use 8 bit in all cases, or are currently treated as 8-bit-images in PDFlib internally) and leaves PNG and in-memory images as possible candidates for constructing a mask. Pixel values of 0 in the mask will result in the corresponding area of the image being painted, while pixel values of 1 result in the background shining through.

In the second step this mask is applied to another image which itself is acquired through one of the usual image functions:

```
image = PDF_open_image_file(p, type, filename, "masked", mask)
if (image != -1)
    PDF_place_image(p, image, x, y, scale);
```

Note the different use of the optional string parameter for *PDF_open_image_file( )*: *mask* for defining a mask, and *masked* for applying a mask to another image. The integer parameter is unused in the first step, and carries the mask descriptor in the second step.

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

PDFlib doesn't make any provisions for painting solid color through a mask (like PostScript's *imagemask* operator), since this is a special case of the general masking mechanism. You can achieve this effect by applying the required mask to an auxiliary image constructed in memory with *PDF_open_image( )* (a solid rectangle of the requested color).

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

```
image = PDF_open_image_file(p, "gif", "pete.gif", "ignoremask", 0)
```

4 PDFlib API Reference

The API reference documents all supported PDFlib functions. A few functions are not supported in certain language bindings since they are not necessary. These cases are mentioned in appropriate notes.

4.1 Data Types and Naming Conventions

PDFlib data types. The exact syntax to be used for a particular language binding may actually vary slightly from the C syntax shown here in the reference. This especially holds true for the PDF document parameter *(PDF *** in the API reference) which has to be supplied as the first argument to almost all PDFlib functions in the C binding, but not those bindings which hide the PDF document parameter in an object created by the language wrapper.

Table 4.1 details the use of the PDF document type and the string type in all language bindings. The data types *integer*, *long*, and *float* are not mentioned since there is an obvious mapping of these types in all bindings. Please refer to the respective language section and the examples in Chapter 2 for more language-specific details.

Table 4.1. Data types in the language bindings.

language binding	*PDF document parameter required?*	*function names use PDF_ prefix?*	*string data type*[1]
ActiveX	*no*	*no*	*BSTR (string) or byte( )* [2]
C (and API reference)	*yes*	*yes*	*const char **
C++	*no*	*no*	*string or const char ** [2]
Java	*no*	*no*	*String or byte[ ]* [2]
Perl	*yes*	*yes*	*string*
Python	*yes*	*yes*	*string*
Tcl	*yes*	*yes*	*string or byte array* [2, 3]

1. C or C++ NULL string values and empty strings are considered equivalent.
2. If not type string it is mentioned in the function description.
3. Byte arrays are only available in Tcl 8.1 and above.

Naming conventions for PDFlib functions. In the C binding, all PDFlib functions live in a global namespace and carry the common *PDF_* prefix in their name in order to minimize namespace pollution. In contrast, several language bindings hide the PDF document parameter in an object created by the language wrapper. For these bindings, the function name given in this API reference must be changed by omitting the *PDF_* prefix and the *PDF ** parameter used as first argument. For example, the API description

```
PDF_open_file(PDF *p, const char *filename);
```

translates into the following when the function is used from Java:

```
p.open_file(String filename);
```

Here, *p* is an object of type *pdflib* created earlier.

4.2 General Functions

4.2.1 Setup Functions

Note *Users of the ActiveX and Java bindings can ignore the functions in this section (but not the parameters).*

Table 4.2 lists relevant parameters and values for this section.

Table 4.2. Parameters and values for the setup functions

set or get	*val or par*	*key*	*explanation*
set	*par.*	*FontAFM* *FontPFM* *FontOutline* *Encoding*	*The corresponding resource file line as it would appear for the respective category in a UPR file (see Section 3.3.7, »Resource Configuration and the UPR Resource File«)*
set	*par.*	*compatibility*	*Set PDFlib's compatibility mode to one of the strings »1.2« or »1.3« for Acrobat 3 or 4. The default is 1.3. This parameter must be set before the first call to PDF_open_*( ). Setting compatibility to »1.2« will make Acrobat 4 features unavailable. Note that strict Acrobat 3 compatibility mode is not required for generating Acrobat 3 compatible files, but only in very specific circumstances related to PDF-enabled RIPs (see Section 1.3, »PDFlib Output and Compatibility«).*
set	*par.*	*prefix*	*Resource file name prefix as used in a UPR file (see Section 3.3.7, »Resource Configuration and the UPR Resource File«). The prefix can only be set once.*
set	*par.*	*resourcefile*	*Relative or absolute file name of the PDFlib UPR resource file. The resource file will be loaded at the next attempt to access resources. The resource file name can only be set once. This call should occur before the first page.*
set	*par.*	*serial*	*Set the serial string for the ActiveX edition (see Section 2.2.2, »Availability and Special Considerations for ActiveX«*
set	*par.*	*warning*	*Enable or suppress warnings (nonfatal exceptions). Possible values are true and false, default value is true.*
set	*value*	*compress*	*Set the compression parameter to a value from 0–9. Default value is 6. Do not change this parameter within a page description. This parameter does not affect certain types of precompressed image data, such as JPEG and CCITT-compressed TIFF.* *0 no compression* *1 best speed* *9 best compression*

void PDF_boot(void)
void PDF_shutdown(void)
Boot and shut down PDFlib, respectively. Recommended for the C language binding, although currently not required. For all other language bindings booting and shutting down is accomplished automatically by the wrapper code, and these functions are not available.

int PDF_get_majorversion(void)
int PDF_get_minorversion(void)
Returns the PDFlib major and minor version number, respectively.

Note Both functions are not available in the ActiveX, Java, Perl, and Tcl bindings because these supply their own versioning schemes.

PDF *PDF_new(void)

Create a new PDF object, using PDFlib's internal default error handling and memory allocation routines. *PDF_new( )* returns a handle to a PDF object which is to be used in subsequent PDFlib calls. The contents of the PDF structure are considered private to the library; only pointers to the PDF structure are used at the API level.

The data type used for the opaque PDF object handle varies among language bindings. This doesn't really affect PDFlib clients, since all they have to do is pass the PDF handle as the first argument to all functions.

This function does not return any error code. If it doesn't succeed due to unavailable memory, a PDFlib exception is raised.

Note This function is not available in the C++ binding since it is hidden in the PDF constructor. In the ActiveX and Java bindings this function is automatically called by the wrapper code, and therefore also not available.

PDF *PDF_new2(
void (*errorhandler)(PDF *p, int type, const char *msg),
void* (*allocproc)(PDF *p, size_t size, const char *caller),
void* (*reallocproc)(PDF *p, void *mem, size_t size, const char *caller),
void (*freeproc)(PDF *p, void *mem),
void *opaque)

Create a new PDF object. Returns a pointer to the opaque PDF data type which is required as the *p* argument for all other functions. Unlike *PDF_new( )*, the caller may optionally supply own procedures for error handling and memory allocation. The function pointers for the error handler, the memory procedures, or both may be NULL. PDFlib will use default routines in these cases. Either all three memory routines must be provided, or none.

Note In the C++ binding this function is indirectly available via the PDF constructor. Not all function arguments must be given since default values of NULL are supplied. In all bindings other than C and C++ this function is automatically called by the wrapper code, and therefore not available.

void PDF_delete(PDF *p)

Delete a PDF object and free all remaining PDFlib-internal resources. Although not necessarily required for single-document generation, deleting the PDF object is heavily recommended for all server applications when they are done producing PDF. This function must only be called once for a given PDF object. *PDF_delete( )* should also be called from client-supplied error handlers for cleanup.

Note In the C++ binding this function is indirectly available via the PDF destructor. In the ActiveX and Java bindings this function is automatically called by the wrapper code, and therefore not available.

void *PDF_get_opaque(PDF *p)

Return the opaque application pointer stored in PDFlib which has been supplied in the call to *PDF_new2( )*. PDFlib never touches the opaque pointer, but supplies it unchanged

to the client. This may be used in multi-threaded applications for storing private thread-specific data within the PDF object.

Note This function is only available in the C and C++ bindings.

4.2.2 Document and Page Functions

Table 4.3 lists relevant parameters and values for this section.

Table 4.3. Parameters and values for the document and page functions

set or get	val or par	key	explanation
set	par.	flush	Set PDFlib's flushing strategy to none, page, content, or heavy. The default is page. See Section 3.1.3, »Generating PDF Documents directly in Memory« for an explanation of the flushing strategies.
set	value	pagewidth pageheight	Change the page size dimensions of the current page. These parameters must only be used within a page description. The parameters must be given as strings.

int PDF_open_file(PDF *p, const char *filename)
Open a new PDF file associated with *p*, using the supplied *filename*. PDFlib will attempt to open a file with the given name, and close the file when the PDF document is finished. This function returns -1 on error, and 1 otherwise.

The special file name »–« can be used for generating PDF on the stdout channel (this obviously does not apply to environments which don't support the notion of a stdout channel, such as the MacOS and ActiveX).

If *filename* is NULL or empty the PDF document will be generated in memory instead of on file. The result must be fetched by the client with the *PDF_get_buffer( )* function. *PDF_open_file( )* will always succeed in this case, and never return the -1 error value.

Note In the C++ binding this function is hidden in the overloaded open( ) call.

int PDF_open_fp(PDF *p, FILE *fp)
Open a new PDF file associated with *p*, using the supplied file handle. The function returns -1 on error, and 1 otherwise.

On MacOS and Windows the *fp* file handle must have been opened in binary mode, which is necessary for PDF output. On Windows PDFlib changes the output mode of the supplied file handle to binary mode itself.

Note This function is only available in the C and C++ bindings. In the C++ binding, it is hidden in the overloaded open( ) call.

void PDF_open_mem(PDF *p, size_t (*writeproc)(PDF *p, void *data, size_t size))
Open a new PDF document in memory, without writing to a disk file. The user-supplied *writeproc* callback function will be called by PDFlib in order to submit (portions of) the generated PDF data. The callback function must return the number of bytes written. If the return value doesn't match the *size* argument supplied by PDFlib, an exception will be thrown, and PDF generation stops. The frequency of *writeproc* calls is configurable with the *flush* parameter. The default value of the flush parameter is *page* (see Section 3.1.3, »Generating PDF Documents directly in Memory« for details).

Note NoteThis function is only available in the C and C++ bindings. In the C++ binding it is hidden in the overloaded open() call.

char * PDF_get_buffer(PDF *p, long *size)
Fetch the full or partial buffer containing the generated PDF data. This function must only be called between page descriptions (i.e., after *PDF_end_page( )* and before *PDF_begin_page( )*), or after *PDF_close( )* and before *PDF_delete( )* (the latter is not required by all language bindings). This function must only be called if an empty filename has been supplied to *PDF_open_file( )*. It returns a buffer full of binary PDF data for consumption by the client. The *size* parameter is only used for C and C++ clients, and points to a memory location where the length of the returned data in bytes will be stored. In all other language bindings an object of appropriate length will be returned, and the *size* parameter must be omitted.

If this function is called between page descriptions, it will return the PDF data generated so far. If it is called after *PDF_close( )* it returns the complete PDF document. If there is only a single call to this function which happens after *PDF_close( )* the return value is guaranteed to contain the complete PDF document in a contiguous buffer.

*Note This function does not return a string of text characters, but a sequence of binary bytes. Both data types may be identical in some language bindings, but different in others. In ActiveX binary data corresponds to a variant array of unsigned bytes (i.e., a variant of type VT_ARRAY | VT_UI1); in Java it corresponds to an array of bytes (i.e., byte[ ]); in Tcl 8.1 and above it corresponds to a byte array; in C++ it corresponds to a char * even if ANSI C++ strings are used.*

void PDF_close(PDF *p)
Finish the generated PDF document, free all document-related resources, and close the output file if the PDF document has been opened with *PDF_open_file( )*. This function must be called when the client is done generating pages, regardless of the method used to open the PDF document.

When the document was generated in memory (as opposed to on file), the document buffer will still be kept after this function is called (so that it can be fetched with *PDF_get_buffer( )*), and will be freed in the next call to *PDF_open( )*, or in *PDF_delete( )*.

void PDF_begin_page(PDF *p, float width, float height)
Start a new page in the PDF file. The *width* and *height* parameters are the dimensions of the new page in points. Acrobat's page size limits are documented in Section 3.2.1, »Coordinate Systems«. A list of commonly used page formats can be found in Table 3.4. Note that there are C convenience definitions for some common page formats (see Section 4.8, »Page Size Formats«). The page size can be changed after calling *PDF_begin_page( )* with the *pagewidth* and *pageheight* parameters. In order to produce landscape pages use *width* > *height*.

void PDF_end_page(PDF *p)
Must be used to finish a page description.

4.2.3 Parameter Handling Functions

PDFlib maintains a number of internal parameters which are used for controlling PDFlib's operation and the appearance of the PDF output. Four functions are available

for setting and retrieving both numerical and string parameters. All parameter strings (both keys and values) are case-sensitive.The descriptions of available parameters can be found in the respective sections.

float PDF_get_value(PDF *p, const char *key, float modifier)
Get the numerical value of some internal PDFlib parameter *key*, in some cases characterized by the *modifier*. For parameters where the description doesn't mention *modifier*, it is ignored and must be 0.

void PDF_set_value(PDF *p, const char *key, float value)
Set some numerical PDFlib parameter *key* to *value*.

char * PDF_get_parameter(PDF *p, const char *key, float modifier)
Get the string value of some PDFlib parameter *key*. , in some cases characterized by the *modifier*. For parameters where the description doesn't mention *modifier*, it is ignored and must be 0.

Note C and C++ clients must neither touch nor free the returned string.

void PDF_set_parameter(PDF *p, const char *key, const char *value)
Set the string value of some PDFlib parameter *key* to *value*.

4.3 Text Functions

4.3.1 Font Handling Functions

Table 4.4 lists relevant parameters and values for this section.

Table 4.4. Parameters and values for the text functions

set or get	*val or par*	*key*	*explanation*
get	*value*	*font*	*Return the identifier of the current font which must have been previously set with PDF_setfont( ). This function must only be called within a page description.*
get	*par.*	*fontname*	*The name of the current font which must have been previously set with PDF_setfont( ). This function must only be called within a page description. The font must have been set before calling this function.*
get	*value*	*fontsize*	*Return the size of the current font which must have been previously set with PDF_setfont( ). This function must only be called within a page description.*
get	*value*	*capheight*	*Return the cap height of the font identified by the modifier. The cap height is the height of the capital letter H in most fonts. The value is measured in em, i.e., it must be multiplied by the desired font size.*
get	*value*	*ascender*	*Return the ascender of the font identified by the modifier. The ascender is the height of the capital letter d in most fonts. The value is measured in em, i.e., it must be multiplied by the desired font size.*
get	*value*	*descender*	*Return the descender of the font identified by the modifier. The descender is the distance from the baseline to the bottom of the letter p in most fonts. It is usually negative. The value is measured in em, i.e., it must be multiplied by the desired font size.*

int PDF_findfont(PDF *p, const char *fontname, const char *encoding, int embed)
Prepare the font *fontname* for later use with *PDF_setfont( )*. The metrics will be loaded from memory or from an external metrics file.

For 8-bit fonts, *encoding* is one of *builtin*, *macroman*, *winansi*, *ebcdic*, or *host* (see Section 3.3.2, »Character Sets and predefined 8-Bit Text Encodings«), or the name of a user-defined encoding (see Section 3.3.3, »Custom Text Encodings«). Note that in order to use arbitrary encodings, you will need metrics information for the font (see Section 3.3.6, »Font Outline and Metrics Files«).

Alternatively, *encoding* can be the name of one of the built-in CMaps if *fontname* describes a CID font (see Section 3.3.8, »CID Font Support for Japanese, Chinese, and Korean Text«). In this case metrics information is not required.

Case is significant for both fontname and encoding.

Note CID fonts are not supported in the Acrobat 3 compatibility mode.

If the *embed* parameter has the value 0, only general font information is included in the PDF output. If *embed = 1*, the font outline file must be available in addition to the metrics information, and the actual font definition will be embedded in the PDF output. However, the font file will only be checked when this function is called, but not yet used, since font embedding is done at the end of the generated PDF file. The *embed* parameter must be 0 for CID fonts.

If the requested font/encoding combination cannot be used due to configuration problem (e.g., a font, metrics, or encoding file could not be found, or a mismatch was detected), an exception of type *PDF_RuntimeError* will be raised. Otherwise, the value returned by this function can be used as font argument to other font-related functions. *PDF_findfont( )* can safely be called outside of page descriptions.

Note The returned number – the font handle – doesn't have any significance to the user other than serving as an argument to PDF_setfont( ) and related functions. In particular, requesting the same font/encoding combination in different documents may result in different font handles.

void PDF_setfont(PDF *p, int font, float fontsize)
Set the current font in the given *fontsize*. The font descriptor must have been retrieved via *PDF_findfont( )*. This function must only be called within a page description. The font must be set on each page before drawing any text. Font settings will not be retained across pages.

4.3.2 Text Output Functions

Note All text supplied to the functions in this section must match the encoding selected with PDF_findfont( ). This applies to 8-bit text as well as Unicode or other encodings selected via a CMap.

Table 4.4 lists relevant parameters and values for this section.

void PDF_show(PDF *p, const char *text)
Print *text* in the current font and font size at the current text position. Both font (via *PDF_setfont( )*) and current point (via *PDF_moveto( )* or another text output function) must have been set before. The current point is moved to the end of the printed text. In the C and C++ bindings *text* must not contain null characters.

Table 4.5. Parameters and values for the text functions

set or get	*val or par*	*key*	*explanation*
set	*value*	*leading*	*Set the leading, which is the distance between baselines of adjacent lines of text. The leading is used for PDF_continue_text( ) and set to the value of the font size when a new font is selected using PDF_setfont( ). Setting the leading equal to the font size results in dense line spacing. However, ascenders and descenders of adjacent lines will generally not overlap.*
set	*value*	*textrise*	*Set the text rise parameter. The text rise specifies the distance between the desired text position and the default baseline. Positive values of text rise move the baseline up. The text rise always relates to the vertical coordinate. This may be useful for superscripts and subscripts. The text rise is set to the default value of 0 at the beginning of each page.*
set	*value*	*horizscaling*	*Set the horizontal text scaling to the given percentage, which must be greater than 0. Text scaling shrinks or expands the text by a given percentage. The text scaling is set to the default of 100 at the beginning of each page. Text scaling always relates to the horizontal coordinate.*
set	*value*	*text-rendering*	*Set the current text rendering mode to one of the values given in Table 3.9. The text rendering parameter is set to the default of 0 (= solid fill) at the beginning of each page.*
set	*value*	*charspacing*	*Set the character spacing, i.e., the shift of the current point after placing the individual characters in a string. The spacing is given in text space units. It is reset to the default of 0 at the beginning of each page. In order to spread the characters apart use positive values for horizontal writing mode, and negative values for vertical writing mode.*
set	*value*	*wordspacing*	*Set the word spacing, i.e., the shift of the current point after placing individual words in a text line. In other words, the current point is moved horizontally after each ASCII space character (0x20). Since fonts with multi-byte encodings don't have an ASCII space character they are not affected by the word spacing. The spacing value is given in text space units. It is reset to the default of 0 at the beginning of each page.*
get	*value*	*textx* *texty*	*The x or y coordinate, respectively, of the current text position.*
set	*par.*	*underline* *overline* *strikeout*	*Set the current underline, overline, and strikeout modes, which are retained until they are explicitly changed. Theses modes can be set independently from each other, and are reset to false at the beginning of each page (see Section 3.3.10, »Text Variations and Text Box Formatting«).* *true* *underline/overline/strikeout text (does not work for CID fonts)* *false* *do not underline/overline/strikeout text*

void PDF_show2(PDF *p, const char *text, int len)
Same as *PDF_show( ),* but with explicit string length in bytes for strings which may contain null characters. If *len = 0* a null-terminated string is assumed as in *PDF_show( )*.

Note This function is only available for the C and C++ bindings, and is not required for the other language bindings.

void PDF_show_xy(PDF *p, const char *text, float x, float y)
Print *text* in the current font at position *(x, y)*. The font must have been set before. The current point is moved to the end of the printed text. In the C and C++ bindings *text* must not contain null characters.

void PDF_show_xy2(PDF *p, const char *text, int len, float x, float y)
Same as *PDF_show_xy( ),* but with explicit string length in bytes for strings which may contain null characters. If *len* = *0* a null-terminated string is assumed as in *PDF_show_xy( )*.

Note This function is only available for the C and C++ bindings, and is not required for the other language bindings.

void PDF_continue_text(PDF *p, const char *text)
Move to the next line and print *text*. The start of the next line is determined by the leading parameter and the most recent call to *PDF_show_xy( )* or *PDF_set_text_pos( )*. The current point is moved to the end of the printed text. In the C and C++ bindings *text* must not contain null characters. This function should not be used in vertical writing mode.

void PDF_continue_text2(PDF *p, const char *text, int len)
Same as *PDF_continue_text( ),* but with explicit string length in bytes for strings which may contain null characters. If *len* = *0* a null-terminated string is assumed as in *continue_text( )*.

Note This function is only available for the C and C++ bindings, and is not required for the other language bindings.

int PDF_show_boxed(PDF *p, const char *text, float x, float y, float width, float height, const char *mode, const char *reserved)
Format the supplied *text* into a rectangular column. *mode* selects the horizontal alignment mode as discussed below.

If *width* = *0* and *height* = *0*, *mode* can attain one of the values *left*, *right*, or *center*, and the text will be formatted according to the chosen alignment with respect to the point *(x, y)*, with *y* denoting the position of the baseline. This function does not check whether the submitted parameters result in some text being clipped at the page edges. It returns the value 0 in this case.

If *width* or *height* is different from *0*, *mode* can attain one of the values *left*, *right*, *center*, *justify,* or *fulljustify*. The supplied text will be formatted into a text box defined by the lower left corner *(x, y)* and the supplied *width* and *height*. If the text doesn't fit into a line, a simple line-breaking algorithm is used to break the text into the next available line, using existing space characters for possible line-breaks. While the *left*, *right*, and *center* modes align the text on the respective line, *justify* aligns the text on both left and right margins. According to common practice the very last line in the box will only be left-aligned in *justify* mode, while in *fulljustify* mode all lines (including the last one if it contains at least one space character) will be left- and right-aligned. *fulljustify* is useful if the text is to be continued in another column.

This function returns the number of characters which could not be processed since the text didn't completely fit into the column. If the text did actually fit, it returns 0.

The current font must have been set before calling this function. The current values of font, font size, horizontal spacing, and leading are used for the text. The *reserved* parameter must be NULL or empty. This function cannot be used with CID fonts.

It is safe to use *PDF_continue_text( )* afterwards if *mode* = *left* or *justify*.

float PDF_stringwidth(PDF *p, const char *text, int font, float size)
Return the width of *text* in an arbitrary font and size which has been selected with *PDF_findfont( )*. The width calculation takes the current values of the following text parameters into account: horizontal scaling, character spacing, and word spacing. In the C and C++ bindings *text* must not contain null characters.

This function cannot be used with CID fonts. If the current font is a CID font, this function returns 0 regardless of the *text* and *size* arguments.

float PDF_stringwidth2(PDF *p, const char *text, int len, int font, float size)
Same as *PDF_stringwidth( ),* but with explicit string length in bytes for strings which may contain null characters. If *len* = 0 a null-terminated string is assumed as in *PDF_stringwidth( )*.

Note This function is only available for the C and C++ bindings, and is not required for the other language bindings.

void PDF_set_text_pos(PDF *p, float x, float y)
Set the current text position to *(x, y)*. The text position is set to the default value of *(0, 0)* at the beginning of each page.

Note The current point for graphics output and the current text position are maintained separately.

4.4 Graphics Functions

4.4.1 General Graphics State Functions

Note Don't use general graphics state functions within a path description (see Section 3.2, »Page Descriptions«).

void PDF_setdash(PDF *p, float b, float w)
Set the current dash pattern to *b* black and *w* white units. *b* and *w* must be non-negative numbers. In order to produce a solid line, set *b* = *w* = 0. The dash parameter is set to solid at the beginning of each page.

void PDF_setpolydash(PDF *p, float *darray, int length)
Set a more complicated dash pattern. The array of the given length contains alternating values for black and white dash lengths. The array values must be non-negative, and not all zero. In order to produce a solid line, choose *length* = 0 and *darray* = *NULL* or an empty array. The array length must be less than or equal to 8; otherwise the array will be truncated. The dash parameter is set to a solid line at the beginning of each page.

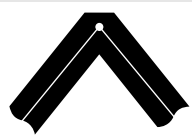
Note The length parameter is only required for the C and C++ language bindings. Other language bindings simply supply the array as argument, and the language wrapper will automatically determine its length.

void PDF_setflat(PDF *p, float flatness)
Set the flatness to a value between 0 and 100 inclusive. The *flatness* parameter describes the maximum distance (in device pixels) between the path and an approximation constructed from straight line segments. The flatness parameter is set to the default value

of 0 at the beginning of each page, which means that the device's default flatness is used.

void PDF_setlinejoin(PDF *p, int linejoin)
Set the linejoin parameter to a value of 0, 1, or 2. The *linejoin* parameter specifies the shape at the corners of paths that are stroked, as shown in Table 4.6. The linejoin parameter is set to the default value of 0 at the beginning of each page.

Table 4.6. Values of the linejoin parameter

value	*description (from the PDF specification)*	*examples*
0	*Miter joins: the outer edges of the strokes for the two segments are continued until they meet. If the extension projects too far, as determined by the miter limit, a bevel join is used instead.*	
1	*Round joins: a circular arc with a diameter equal to the line width is drawn around the point where the segments meet and filled in, producing a rounded corner.*	
2	*Bevel joins: the two path segments are drawn with butt end caps (see the discussion of linecap parameter), and the resulting notch beyond the ends of the segments is filled in with a triangle.*	

void PDF_setlinecap(PDF *p, int linecap)
Set the linecap parameter to a value 0, 1, or 2. The linecap parameter controls the shape at the ends of open paths with respect to stroking, as shown in Table 4.7. The linecap parameter is set to the default value of 0 at the beginning of each page.

Table 4.7. Values of the linecap parameter

value	*description (from the PDF specification)*	*examples*
0	*Butt end caps: the stroke is squared off at the endpoint of the path.*	
1	*Round end caps: a semicircular arc with a diameter equal to the line width is drawn around the endpoint and filled in.*	
2	*Projecting square end caps: the stroke extends beyond the end of the line by a distance which is half the line width and is squared off.*	

void PDF_setmiterlimit(PDF *p, float miter)
Set the miter limit to a value greater than or equal to 1. The *miterlimit* parameter is set to the default value of 10 at the beginning of each page.

void PDF_setlinewidth(PDF *p, float width)
Set the current line width to *width* units in the user coordinate system. The linewidth parameter is set to the default value of 1 at the beginning of each page.

4.4.2 Special Graphics State Functions

All graphics state parameters are restored to their default values at the beginning of a page. The default values are documented in the respective function descriptions. Functions related to the text state are listed in Section 4.3, »Text Functions«.

All transformation functions (*PDF_translate( )*, *PDF_scale( )*, *PDF_rotate( )*, *PDF_skew( )*, and *PDF_concat( ))* change the coordinate system used for drawing future objects. They do not affect existing objects on the page at all.

void PDF_save(PDF *p)
Save the current graphics state. The graphics state contains parameters that control all types of graphics objects. Saving the graphics state is not required by PDF; it is only necessary if the application wishes to return to some specific graphics state later (e.g., a custom coordinate system) without setting all relevant parameters explicitly again. The following items are subject to save/restore:

- graphics parameters: clipping path, coordinate system, current point, flatness, line cap style, dash pattern, line join style, line width, miter limit;
- color parameters: fill and stroke colors;
- text parameters: character spacing, word spacing, horizontal scaling, leading, font, font size, rendering mode, text rise;

The saved graphics state does not include the values of the fillrule, underline, overline, and strikeout parameters.

void PDF_restore(PDF *p)
Restore the most recently saved graphics state. The corresponding graphics state must have been saved on the same page. Pairs of *PDF_save( )* and *PDF_restore( )* may be nested.

Note Although the PDF specification doesn't limit the nesting level of save/restore pairs, applications must keep the nesting level below 10 in order to avoid printing problems caused by restrictions in the PostScript output produced by PDF viewers, and to allow for additional save levels required by PDFlib internally.

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

void PDF_scale(PDF *p, float sx, float sy)
Scale the coordinate system by *sx* and *sy*. This function may also be used for achieving a reflection (mirroring) by using a negative scaling factor.

Note Due to limitations in the Acrobat viewers, PDFlib must output coordinates with absolute values above 32.767 as integers. This may affect output accuracy in rare cases (when very small scaling factors and very large coordinates are used).

void PDF_rotate(PDF *p, float phi)
Rotate the user coordinate system by *phi* degrees. Angles are measured counterclockwise from the positive x axis of the current coordinate system.

void PDF_skew(PDF *p, float alpha, float beta)
Skew the coordinate system by the angles of *alpha* and *beta* degrees. Skewing (or shearing) distorts the coordinate system by the given angles in x and y direction. Angles are measured counterclockwise from the positive x axis of the current coordinate system. Both *alpha* and *beta* must be different from 90° and 270°.

void PDF_concat(PDF *p, float a, float b, float c, float d, float e, float f)
Concatenate a matrix to the current transformation matrix (CTM) for text and graphics. This function allows for the most general form of transformations. Unless you are familiar with transformation matrices, the use of *PDF_translate( ), PDF_scale( ), PDF_rotate( ),* and *PDF_skew( )* is suggested of this function. The CTM is reset to the default identity matrix *(1, 0, 0, 1, 0, 0)* at the beginning of each page. The six floating point values make up the matrix in the same way as in PostScript and PDF (see references). In order to avoid degenerate transformations, *a*d* must not be equal to *b*c*.

4.4.3 Path Segment Functions

Table 4.8 lists relevant parameters and values for this section.

Table 4.8. Parameters and values for the path segment functions

set or get	*val or par*	*key*	*explanation*
get	*value*	*currentx currenty*	*The x or y coordinate, respectively, of the current point*

void PDF_moveto(PDF *p, float x, float y)
Set the current point to *(x, y)*. The current point is set to the default value of *undefined* at the beginning of each page.

Note The current point for graphics output and the current text position are maintained separately.

void PDF_lineto(PDF *p, float x, float y)
Add a straight line from the current point to *(x, y)* to the current path. The current point must be set before using this function. The point *(x, y)* becomes the new current point.

The line will be centered around the »ideal« line, i.e. half of the linewidth (as determined by the value of the linewidth parameter) will be painted on each side of the line connecting both endpoints. The behavior at the endpoints is determined by the value of the linecap parameter.

void PDF_curveto(PDF *p, float x1, float y1, float x2, float y2, float x3, float y3)
Add a Bézier curve from the current point to *(x3, y3)* to the current path, using *(x1, y1)* and *(x2, y2)* as control points. The endpoint of the curve becomes the new current point.

void PDF_circle(PDF *p, float x, float y, float r)
Add a circle with center *(x, y)* and radius *r* to the current path. The point *(x + r, y)* becomes the new current point. Elliptical curves can be constructed by applying non-uniform scaling factors before drawing the circle.

void PDF_arc(PDF *p, float x, float y, float r, float start, float end)
Add a circular arc segment with center *(x, y)* and radius *r* to the current path, extending from *start* to *end* degrees. Angles are measured counterclockwise from the positive x axis of the current coordinate system. Before drawing the arc segment, this function implicitly moves the current point to the beginning of the arc. The endpoint of the arc becomes the new current point.

void PDF_rect(PDF *p, float x, float y, float width, float height)
Add a rectangle with lower left corner *(x, y)* and the supplied *width* and *height* to the current path. Setting the current point is not required before using this function. The point *(x, y)* becomes the new current point. The lines will be centered around the »ideal« line, i.e. half of the linewidth (as determined by the value of the linewidth parameter) will be painted on each side of the line connecting the respective endpoints.

void PDF_closepath(PDF *p)
Close the current path, i.e., add a line from the current point to the starting point of the path. The current point must be set before using this function.

4.4.4 Path Painting and Clipping Functions

Table 4.9 lists relevant parameters and values for this section.

Table 4.9. Parameters and values for the general graphics state functions

set or get	*val or par*	*key*	*explanation*
set	*par.*	*fillrule*	*Set the current fill rule to winding or evenodd. The fill rule is used by PDF viewers to determine the interior of shapes for the purpose of filling or clipping. Since both algorithms yield the same result for simple shapes, most applications won't have to change the fill rule. The fill rule is reset to the default of winding at the beginning of each page.*

void PDF_stroke(PDF *p)
Stroke (draw) the current path with the current line width and the current stroke color. This operation clears the path.

void PDF_closepath_stroke(PDF *p)
Close the current path and stroke it with the current line width and the current stroke color. This operation clears the path.

void PDF_fill(PDF *p)
Fill the interior of the current path with the current fill color. The interior of the path is determined by one of two algorithms (see *PDF_setfillrule( )*). Open paths are implicitly closed before being filled. This operation clears the path.

void PDF_fill_stroke(PDF *p)
Fill and stroke the path with the current fill and stroke color, respectively.

void PDF_closepath_fill_stroke(PDF *p)
Close the path, fill, and stroke it.

void PDF_endpath(PDF *p)
End the current path.

void PDF_clip(PDF *p)
Use the current path as the clipping path for future operations. The clipping path is set to the default value of the page size at the beginning of each page. This operation clears the path. The clipping path is subject to *PDF_save( )/PDF_restore( )*.

4.5 Color Functions

All color functions expect gray and color values in the inclusive range 0–1. The values are interpreted according to additive color mixture, i.e., 0 means no color and 1 means full intensity. Therefore, a gray value of 0 and RGB values with *(r, g, b) = (0, 0, 0)* mean black; a gray value of 1 and RGB values with *(r, g, b) = (1, 1, 1)* mean white. RGB or gray values in the range 0–255 must be scaled to the range 0–1 by dividing by 255.

Note Don't use color functions within a path description (see Section 3.2, »Page Descriptions«).

void PDF_setgray_fill(PDF *p, float gray)
Set the current fill color to the *gray* value. The gray fill parameter is set to the default value of 0 at the beginning of each page.

void PDF_setgray_stroke(PDF *p, float gray)
Set the current stroke color to the *gray* value. The gray stroke parameter is set to the default value of 0 at the beginning of each page.

void PDF_setgray(PDF *p, float gray)
Set the current fill and stroke color to the *gray* value. The gray parameter is set to the default value of 0 at the beginning of each page.

void PDF_setrgbcolor_fill(PDF *p, float red, float green, float blue)
Set the current fill color to the supplied RGB values. The rgbcolor fill parameter is set to the default value of *(0, 0, 0)* at the beginning of each page.

void PDF_setrgbcolor_stroke(PDF *p, float red, float green, float blue)
Set the current stroke color to the supplied RGB values. The rgbcolor stroke parameter is set to the default value of *(0, 0, 0)* at the beginning of each page.

void PDF_setrgbcolor(PDF *p, float red, float green, float blue)
Set the current fill and stroke color to the supplied RGB values. The rgbcolor parameter is set to the default value of *(0, 0, 0)* at the beginning of each page.

4.6 Image Functions

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

Table 4.10 lists relevant parameters and values for this section.

Table 4.10. Parameters and values for the image functions

set or get	*val or par*	*key*	*explanation*
get	*value*	*imagewidth* *imageheight*	*Get the width or height, respectively, of an image in pixels. The modifier is the integer handle of the selected image.*
get	*value*	*resx* *resy*	*Get the horizontal or vertical resolution of an image, respectively. The modifier is the integer handle of the selected image.* *If the value is positive, the return value is the resolution in pixels per inch (dpi). If then return value is negative it can be used to find the aspect ratio of non-square pixels, but doesn't have any absolute meaning. If the return value is zero, the resolution of the image is unknown.*

int PDF_open_image_file(PDF *p,
const char *type, const char *filename, const char *stringparam, int intparam)

Open and analyze a raster graphics file in one of the supported file formats as determined by the *type* parameter. The *type* parameter may attain the following values: *png*, *gif*, *jpeg*, *tiff* (case is significant for all parameters). The returned image handle, if not -1, may be used in subsequent image-related calls.

PDFlib will open the image file with the given name, process the contents, and close it before returning from this call. Although images can be placed multiply within a document (see *PDF_place_image( )*), the actual image file is not kept open after this call.

The *stringparam* and *intparam* parameters are used for additional image attributes according to Table 4.11. If *stringparam* is unused, it must be NULL or an empty string, and *intparam* must be 0.

Note The returned image handles cannot be reused across multiple PDF documents.

Note Masked images and progressive JPEGs are not supported in Acrobat 3 compatibility mode.

Table 4.11. The stringparam and intparam parameters of PDF_open_image_file( )

stringparam	*explanation and possible intparam values*
mask	*Create a mask from this image. The returned image handle may be used in subsequent calls for opening another image and supplied this image for the »masked« parameter. The intparam parameter is ignored in this case, and must be 0.*
masked	*Use the image descriptor given in intparam as a mask for this image. The intparam parameter is an image handle which has been retrieved with a previous call to PDF_open_image( ) with the »mask« parameter.*
ignoremask	*Ignore any transparency information which may be present in the image file.*
invert	*Invert black and white for 1-bit TIFF images. This is mainly intended as a workaround for certain TIFF images which are interpreted differently by different applications.*

int PDF_open_CCITT(PDF *p,
const char *filename, int width, int height, int BitReverse, int K, int BlackIs1)

Open an image file with raw CCITT G3 or G4 compressed bitmap data (this is different from a TIFF file which contains CCITT-compressed image data!). The returned image handle, if not -1, may be used in subsequent image-related calls. However, since PDFlib is unable to analyze CCITT images, all relevant parameters have to be passed to *PDF_open_CCITT( )* by the client. The parameters have the following meaning (apart from *filename*, *width*, and *height*, which are obvious):

BitReverse: If 1, do a bitwise reversal of all bytes in the compressed data.

K: CCITT compression parameter for encoding scheme selection. It has to be set as follows: -1 indicates G4 encoding, 0 indicates one-dimensional G3 encoding (G3-1D), 1 indicates mixed one- and two-dimensional encoding (G3, 2-D) as supported by PDF.

BlackIs1: If this parameter has the value 1, 1-bits are interpreted as black and 0-bits as white. Most CCITT images don't use such a black-and-white reversal, i.e., most images use *BlackIs1 = 0*.

int PDF_open_image(PDF *p, const char *type, const char *source, const char *data, long length, int width, int height, int components, int bpc, const char *params)

This versatile interface can be used to work with image data in several formats and from several data sources. The returned image handle, if not -1, may be used in subsequent image-related calls.

The *type* parameter denotes the kind of image data or compression. It can attain the values *jpeg*, *ccitt*, or *raw* (see Section 3.4.1, »Image File Formats«); the *source* parameter denotes where the image data comes from, and can attain the values *fileref*, *url*, or *memory* (see Section 3.4.4, »Memory Images and External Image References«). The relationship among the *source*, *data*, and *length* parameters is explained in Table 4.12.

Table 4.12. Values of the source, data, and length parameters of PDF_open_image( )

source	*data*	*length*
fileref	*string with a platform-independent file name (see [1])*[1]	*unused, should be 0*
url	*string with an image URL conforming to RFC 1738*[1]	*unused, should be 0*
memory	*Binary bytes containing image data; the image data is compressed according to the type parameter*	*length of (compressed) image data in bytes.*

1. data is not a string in Java, ActiveX, and C++, which makes it a little bit clumsy to pass filenames or URLs.

*Note The data parameter is not a string of text characters, but a sequence of binary bytes which is the same data type in some language bindings, but not in others. In ActiveX it corresponds to a variant array of unsigned bytes (i.e., a variant of type VT_ARRAY | VT_UI1); in Java it corresponds to an array of bytes (i.e. byte[ ]); in Tcl 8.1 and above it corresponds to a byte array; in C++ it corresponds to a char * even if ANSI C++ strings are used.*

Note Images referenced via external files or URLs are not supported in Acrobat 3 compatibility mode.

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

If *type* is *raw*, length must be equal to *[width x components x bpc/ 8] x height* bytes, with the bracketed term adjusted upwards to the next integer, and this exact amount of data must be supplied. The image samples are expected in the standard PostScript/PDF ordering. Even if *bpc* is not 8, each pixel row begins on a byte boundary, and color values must be packed from left to right within a byte. Image samples are always interleaved, i.e., all color values for the first pixel are supplied first, followed by all color values for the second pixel, and so on. If *components = 1* and *bpc = 1*, *params* may be *mask* in order to use this image as an image mask.

If *type* is *ccitt*, CCITT-compressed image data is expected. In this case, *params* is examined. For CCITT images two parameters as described for *PDF_open_CCITT( )* can be supplied in the *params* string as follows:

```
/K -1 /BlackIs1 true
```

Supported values for */K* are -1, 0, or 1, the default value is 0. Supported values for */BlackIs1* are *true* and *false*; the default value is *false*. The default values will be used if a NULL or empty *params* string is supplied. BitReverse cannot be supplied in this string. Instead, a special notion is used: if *length* is negative, the image data will be reversed.

If *params* is not used, it must be NULL or empty. The client is responsible for the memory pointed to by the *data* argument. The memory may be freed by the client immediately after this call.

Note *Unlike PDF_open_image_file( ) this function doesn't retrieve the image parameters from the image data, but relies on correct values being supplied by the client. Passing incorrect values will result in error messages when trying to open the PDF document in Acrobat.*

void PDF_close_image(PDF *p, int image)
Close the image. This only affects PDFlib's associated internal image structure. If the image has been opened from file, the actual image file is not affected by this call since it has already been closed at the end of the corresponding *PDF_open_image_file( )* call. An image handle cannot be used any more after having been closed with this function, since it cuts PDFlib's internal association with the image.

void PDF_place_image(PDF *p, int image, float x, float y, float scale)
Place the supplied image (which must have been retrieved with one of the *PDF_open_*( )* functions) on the current page. The lower left corner of the image is placed at *(x, y)* on the current page, and the image is scaled by the supplied scaling factor. Non-uniform scaling may be achieved with *PDF_scale( )*, optionally bracketing the sequence with *PDF_save( )* and *PDF_restore( )*.

PDFlib never changes the number of pixels in an 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. If the user coordinate system hasn't been scaled, this means that the image will be imported at 72 dpi. Resolution values which may be contained in the original image file will be ignored; the user is responsible for scaling the coordinate system appropriately (beware of non-square pixels!).

This function can be called an arbitrary number of times on arbitrary pages, as long as the image handle has not been closed with *PDF_close_image( )*. *PDF_place_image( )* must only be used within page descriptions, i.e., between *PDF_begin_page( )* and *PDF_end_page( )*.

4.7 Hypertext Functions

4.7.1 Document Open Action

Table 4.13 lists relevant parameters and values for this section.

Table 4.13. Keys and numerical values for PDF_set_value( )

set or get	val or par	key	explanation
set	par.	openaction	Set the open action, i.e., the zoom factor for the first page of the document. Possible values are retain, fitpage, fitwidth, fitheight, fitbbox. The meaning of these values is explained in Table 4.18. The default is retain. This parameter can be set once at an arbitrary time before PDF_close( ).

4.7.2 Bookmarks

int PDF_add_bookmark(PDF *p, const char *text, int parent, int open)
Add a PDF bookmark with the supplied *text* that points to the current page. The text may be encoded with PDFDocEncoding or Unicode. This function must not be called before starting the first page of the document with *PDF_begin_page( )*.

This function returns an identifier for the bookmark just generated. This identifier may be used as the *parent* parameter in subsequent calls. In this case, a new bookmark will be generated which is a subordinate of the given parent. In this way, arbitrarily nested bookmarks can be generated. If *parent = 0* a new top-level bookmark will be generated. If the *open* parameter has a value of 0, child bookmarks will not be visible. If *open = 1*, all children will be folded out.

This function must only be called within a page description.

4.7.3 Document Information Fields

void PDF_set_info(PDF *p, const char *key, const char *value)
Fill document information field *key* with *value*. The value can be encoded with PDFDocEncoding or Unicode, while the *key* must be encoded with PDFDocEncoding. *key* may be any of the four standard information field names, or up to one custom field name (see Table 4.14). If a custom field name is used, it must consist of printable ASCII characters except any of the following: blank *' '*, *%*, *(,)*, *<, >*, *[,]*, *{, }*, */*, and *#*.

Table 4.14. Values for the document information field key

key	explanation
Subject	Subject of the document
Title	Title of the document
Creator	Creator of the document
Author	Author of the document
(any custom name)	User-defined field name. PDFlib supports one additional field which may be arbitrarily named.

4.7.4 Page Transitions

PDF files may specify a page transition in order to achieve special effects which may be useful for presentations or »slide shows«. In Acrobat, these effects cannot be set document-specific or on a page-by-page basis, but only for the full screen mode. PDFlib, however, allows setting the page transition mode and duration for each page separately. Table 4.13 lists relevant parameters and values for this section.

Table 4.15. Keys and numerical values for PDF_set_value( )

set or get	val or par	key	explanation
set	par.	transition	Set the page transition effect for the current and any subsequent pages until until the transition is changed again. The transition type strings givenbelow are supported. type may also be NULL to reset the transition effect. The default transition is replace, i.e., no special effect. split — Two lines sweeping across the screen reveal the page blinds — Multiple lines sweeping across the screen reveal the page box — A box reveals the page wipe — A single line sweeping across the screen reveals the page dissolve — The old page dissolves to reveal the page glitter — The dissolve effect moves from one screen edge to another replace — The old page is simply replaced by the new page (default)
set	value	duration	Set the page display duration in seconds for the current page. The default duration is one second.

4.7.5 File Attachments

void PDF_attach_file(PDF *p, float llx, float lly, float urx, float ury, const char *filename, const char *description, const char *author, const char *mimetype, const char *icon)

Add a file attachment annotation at the rectangle specified by its lower left and upper right corners in default user space coordinates. *description* and *author* may be encoded in PDFDocEncoding or Unicode. *mimetype* is the MIME type of the file and will be used by Acrobat for launching the appropriate program when the file attachment annotation is activated. The *icon* parameter controls the display of the unopened file attachment in Acrobat, as shown in Table 4.16.

Note *PDF file attachments are only supported in Acrobat 3, and are not supported in Acrobat 3 compatibility mode. Moreover, Acrobat Reader is unable to deal with file attachments and will display a question mark instead. File attachments only work in the full Acrobat software.*

Table 4.16. Icon names for file attachments

icon name	icon appearance	icon name	icon appearance
graph		pushpin	
paperclip		tag	

4.7.6 Note Annotations

void PDF_add_note(PDF *p, float llx, float lly, float urx, float ury,
const char *contents, const char *title, const char *icon, int open)

Add a note annotation at the rectangle specified by its lower left and upper right corners in default user space coordinates. *contents* and *title* may be encoded with PDFDocEncoding or Unicode. The *icon* parameter controls the display of the unopened note attachment in Acrobat, as shown in Table 4.17. The annotation will be opened if *open = 1*, and closed if *open = 0*.

Note *Different note icons are only available in Acrobat 4, and are not supported in Acrobat 3 compatibility mode (the icon parameter must be NULL in this case). Acrobat 3 viewers (and apparently Unix versions of Acrobat 4) will display the »note« type icon regardless of the supplied icon parameter.*

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

Table 4.17. Icon names for note annotations

icon name	*icon appearance*	*icon name*	*icon appearance*
comment		*newparagraph*	
insert		*key*	
note		*help*	
paragraph			

4.7.7 Links

void PDF_add_pdflink(PDF *p,
float llx, float lly, float urx, float ury, const char *filename, int page, const char *dest)

Add a file link annotation to the PDF file *filename* at the rectangle specified by its lower left and upper right corners in default user space coordinates. *page* is the physical page number of the target page. *dest* specifies the destination zoom. It can attain one of the values specified in Table 4.18.

void PDF_add_locallink(PDF *p,
float llx, float lly, float urx, float ury, int page, const char *dest)

Add a link annotation with a target page in the current document at the rectangle specified by its lower left and upper right corners in default user space coordinates. *page* is the physical page number of the target page, and may be a previously generated page, or a page in the same document that will be generated later (after the current page). However, the application must make sure that the target page will actually be generated; PDFlib will issue a warning message otherwise. *dest* specifies the destination zoom. It can attain one of the values specified in Table 4.18.

Table 4.18. Values for the dest parameter of PDF_add_pdflink( ) and PDF_add_locallink( )

dest	*explanation*
retain	*Retain the zoom factor which was in effect when the link was activated.*
fitpage	*Fit the complete page to the window.*
fitwidth	*Fit the page width to the window.*
fitheight	*Fit the page height to the window.*
fitbbox	*Fit the page's bounding box (the smallest rectangle enclosing all objects) to the window.*

void PDF_add_launchlink(PDF *p, float llx, float lly, float urx, float ury, const char *filename)
Add a launch annotation (arbitrary file type) at the rectangle specified by its lower left and upper right corners in default user space coordinates. *filename* is the name of the file which will be launched upon clicking the link.

void PDF_add_weblink(PDF *p, float llx, float lly, float urx, float ury, const char *url)
Add a weblink annotation at the rectangle specified by its lower left and upper right corners in default user space coordinates. *url* is a Uniform Resource Identifier encoded in 7-bit ASCII specifying the link target. It can point to an arbitrary (Web or local) resource.

void PDF_set_border_style(PDF *p, const char *style, float width)
Set the border style for all kinds of annotations. These settings are used for all annotations until a new style is set. At the beginning of a document the annotation border style is set to a default of a solid line with a width of 1. Possible values of the style parameter are *solid* and *dashed*. If *width = 0* the links will be invisible.

void PDF_set_border_color(PDF *p, float red, float green, float blue)
Set the border color for all kinds of annotations. At the beginning of a document the annotation border color is set to *(0, 0, 0)*.

void PDF_set_border_dash(PDF *p, float b, float w)
Set the border dash style for all kinds of annotations (see *PDF_setdash( )*). At the beginning of a document the annotation border dash style is set to a default of *(3, 3)*. However, this default will only be used when the border style is explicitly set to *dashed*.

4.8 Page Size Formats

<format>_width, <format>_height, *where format is one of*
a0, a1, a2, a3, a4, a5, a6, b5, letter, legal, ledger, p11x17;
These macro definitions provide page width and height values for the most common page formats which may be used in calls to *PDF_begin_page( )*.

Note *These values are only supplied for the C and C++ bindings. Other language clients may use the values provided in Table 3.4.*

5 The PDFlib License

PDFlib is available under two different licensing terms which are substantially different, and meet the needs of different developer groups. Please take the time to read the short summaries below in order to decide which one applies to your development.

5.1 The »Aladdin Free Public License«

This license applies to the main PDFlib package, but not to the ActiveX edition and any EBCDIC editions (both of which are only available under the terms of the commercial PDFlib license). The complete text of the license agreement can be found in the file *aladdin-license.pdf*. In short and non-legal terms:

- you may develop free software with PDFlib, provided you make your source code available
- you may develop software for your own use with PDFlib as long as you don't sell it
- you may redistribute PDFlib non-commercially
- you may redistribute PDFlib on digital media for a fee if the complete contents of the media are freely redistributable.

Note that only the text in the file *aladdin-license.pdf* is considered to completely describe the licensing conditions. Project managers please note: using PDFlib in your commercial projects is not covered by the Aladdin license, and effectively means jeopardizing your project through unlicensed software.

5.2 The Commercial PDFlib License

A commercial PDFlib license is required for all uses of the software which are not explicitly covered by the Aladdin Free Public License, for example:

- shipping a commercial product which contains PDFlib
- distributing (free or commercial) software based on PDFlib when the source code is not made available
- implementing commercial Web services with PDFlib

Details of the commercial PDFlib license including options, pricing, and support can be found in the file *PDFlib-license.pdf* in the PDFlib distribution. Please contact us if you're interested in obtaining a commercial PDFlib license, or have any questions:

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

Licensing contact: sales@pdflib.com
Support for PDFlib licensees: support@pdflib.com (include your license number)
For other inquiries check the PDFlib mailing list (see http://www.pdflib.com for details).

6 References

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

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

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

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

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

German edition: Thomas Merz, Mit Acrobat ins World Wide Web.
Effiziente Erstellung von PDF-Dateien und ihre Einbindung ins Web.
Mit CD-ROM. ISBN 3-9804943-1-4, Thomas Merz Verlag 1998
80331 München, Tal 40, fax +49 • 89 • 29 16 46 86
http://www.pdflib.com

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

A Shared Libraries and DLLs

The details of building and using shared libraries, also known as shared objects or dynamic link libraries (DLLs), are among the most frequently asked questions of PDFlib users. PDFlib is in most cases used as a shared library, and uses several auxiliary shared libraries itself (these are zlib, libpng, and TIFFlib). For your convenience, we collected some useful information about shared libraries in this appendix.

Shared Libraries on Unix Systems

The many faces of shared libraries on Unix. Most problems with shared libraries are related to the variety of methods, options, and calls invented by Unix system vendors for implementing shared library support. In order to facilitate building and using shared libraries on a wealth of Unix systems, PDFlib leverages GNU libtool[1]. This is a collection of shell-scripts which attempt to »do the right thing« in order to build and use shared libraries on Unix systems.

While libtool support is completely integrated into the PDFlib configuration machinery, it is suggested to take a look at libtool and the corresponding documentation if you want to learn more about shared libraries.

Building shared libraries. Although we do not even attempt to completely cover the intricate details of shared libraries here, the hints given below may be helpful for PDFlib users. Examples for Linux and other Unix systems are shown in brackets.

- On many systems a compiler flag (Linux: *-fPIC*) must be used for modules which are intended to be linked into a shared library (so-called position-independent code, or PIC).
- Similarly, most systems require a special linker flag for shared libraries (Linux: *-shared*).
- The naming conventions for shared libraries vary (Linux and most others: *.so*, HP-UX: *.sl*)
- The system may or may not support a versioning system for shared libraries. Some systems require a version number to be included in the shared library file name, others at least tolerate it. Still others refuse to load libraries with version numbers in their names. The version number is often appended to the file name suffix with or without an additional dot (Linux: *lib<name>.so.5*, BSDI: *lib<name>.so5*). The system may or may not consider version numbers when loading shared libraries.

The PDFlib configure script and GNU libtool try to take care of all these issues by constructing suitable Makefiles. In case of problems try to locate as much information as possible regarding the above issues, and compare with the generated Makefiles.

Using shared libraries. Once you managed to correctly build your shared library, you are not yet done – you must make sure that the run-time linker (which loads and runs your program) is able to access the library:

- In order to actually find shared libraries, a variety of mechanisms is deployed. The most common is an environment variable (Linux, Solaris, and many others: *LD_LIBRARY_PATH*, HP-UX: *SHLIB_PATH*, AIX: *LIBPATH*). It contains a colon-separated list of

1. See http://www.gnu.org/software/libtool/libtool.html

directories which are searched for shared libraries. Failing that, a cache file (see below) is consulted, and then some set of default system directories (Linux: */usr/lib* and */lib)*. Setting an environment variable doesn't require *root* privilege, and can be useful for testing. Library paths can also be hard-coded in the executable file using a special linker option (Solaris: *-R*).

- In order to prepare the cache consulted by the run-time linker, a special program (Linux: *ldconfig*) must be invoked. This program scans all relevant locations for shared libraries and sets up a cache file with the known libraries (Linux: */etc/ld.so.cache*). Usually this program is invoked at boot time, and requires *root* privilege. This technique is useful for permanently installing a shared library on a system.

The PDFlib configure script and GNU libtool emit some instructions explaining the required steps for using a shared library after the build process is completed. You may recognize some of the above information in these instructions. Of course, the details vary among systems.

In order to find out the shared libraries required by a program or another shared library, a special utility (Linux: *ldd)* can be invoked. It informs about the necessary libraries for running a given program, and tries to locate these on the system. This is convenient for the analysis of shared library related problems.

Note If you find yourself fiddling with shared library related problems because you cannot install the libraries due to a lack of administrator priviledges, take a look at the .libs subdirectory and the library wrapper scripts created by libtool. These items, along with the commands issued for the test and install targets will give you an idea of libtool's library deployment.

Note On a few systems name clashes with existing libraries will occur when trying to install the PDFlib libraries (Digital Unix is an example of such a system). In this case you can change the name of the generated PDFlib library via a configure option.

To share or not to share? Note that while most Unix systems support shared libraries, not all do. According to the libtool documentation, building shared libraries is not supported on the following systems:

```
alpha-dec-osf2.1
i*86-*-bsdi3.1
i*86-*-bsdi3.0
i*86-*-bsdi2.1
i*86-pc-cygwin
m68k-next-nextstep3
m68k-sun-sunos4.1.1
mips-sgi-irix5.3
powerpc-ibm-aix4.1.5.0
```

PDFlib's *configure* mechanism will therefore build static versions of the library on those systems. This implies that most language bindings will be unavailable.

Special issues with Internet Service Providers. You may happen to find yourself in a difficult situation when trying to implement a PDFlib-based solution on a Unix machine run by your Internet Service Provider (ISP): you are allowed to install your own scripts and programs, but don't have the privileges required to change the run-time environment of the Web server. This may especially hurt you with respect to the *LD_LIBRARY_PATH* (or other) environment variable used to locate shared libraries.

For example, a PDFlib-based Perl script will first apply a Perl-specific search strategy for locating the PDFlib Perl wrapper library, and find it. This library, in turn, requires the core PDFlib library and possibly additional libraries. However, these libraries may not be found because you can't manipulate the Web server's or Perl interpreter's environment in which your script is run.

If you can't find any other solution to this problem, the following hint may be useful: since the Perl library search mechanism manages to find at least the first library, do not rely on the stacked shared library approach, but instead build a single shared library consisting of the Perl wrapper, the core PDFlib, and possibly additional required libraries – all linked into a single shared library which will be found by the Perl interpreter. Note that PDFlib's configure mechanism currently doesn't support such a configuration, so you will have to hand-tune the Makefiles.

In addition to Perl, this problem and the suggested solution may apply to other language bindings and environments, too.

Library versioning scheme used by libtool. If the operating system supports a versioning scheme for shared libraries libtool will make use of it, and create versioned libraries for PDFlib. It is very important to note that library version numbers are different from software version numbers – don't expect PDFlib's major and minor version numbers to show up in library file names! Library version numbers rather identify the binary programming interface exposed by the library. A table with the PDFlib version numbers and the corresponding interface (libtool) numbers can be found in the distribution.

Windows DLLs

DLLs (Dynamic Link Libraries) form one of the cornerstones of the Windows architecture. Building and using DLLs is very well understood, and generally doesn't pose any problems. The major exception to this rule is the cluttering of the Windows directory with all kinds of DLLs installed by every vendor and his dog. The PDFlib ActiveX component tries to avoid this issue by installing all required DLLs into a single application-specific directory. If you want to move PDFlib DLLs around your system, it may be useful to know the order in which Windows searches for DLLs:

- The current directory
- Windows 95/98: the Windows system directory
- Windows NT/2000: the 32-bit Windows system directory (*system32*)
- Windows NT/2000: the 16-bit Windows system directory (*system*)
- The Windows directory
- The directories listed in the *PATH* environment variable

Note The PDFlib ActiveX edition takes care of these issues through a private installation directory and custom registry entries.

Shared Libraries on the Macintosh

Shared libraries on the MacOS are fully supported on PowerPC machines via the Code Fragment Manager (*CFM*). 68K Macs require an extension called *CFM68K* which will not be further discussed here. A file type of *shlb* is generally used for shared libraries. The system looks for shared libraries in the following locations:

- The application folder
- The *Extensions* folder in the active system folder

B Summary of PDFlib Functions

This appendix summarizes the syntax of all PDFlib functions, and refers to the page in this manual where the function is discussed in detail.

General Functions

Function prototype	*page*
void PDF_boot(void)	*67*
void PDF_shutdown(void)	*67*
int PDF_get_majorversion(void)	*67*
int PDF_get_minorversion(void)	*67*
*PDF *PDF_new(void)*	*68*
*PDF *PDF_new2(void (*errorhandler)(PDF *p, int type, const char *msg), void* (*allocproc)(PDF *p, size_t size, const char *caller), void* (*reallocproc)(PDF *p, void *mem, size_t size, const char *caller), void (*freeproc)(PDF *p, void *mem), void *opaque)*	*68*
*void PDF_delete(PDF *p)*	*68*
*void *PDF_get_opaque(PDF *p)*	*68*
*int PDF_open_file(PDF *p, const char *filename)*	*69*
*int PDF_open_fp(PDF *p, FILE *fp)*	*69*
*void PDF_open_mem(PDF *p, size_t (*writeproc)(PDF *p, void *data, size_t size))*	*69*
*char * PDF_get_buffer(PDF *p, long *size)*	*70*
*void PDF_close(PDF *p)*	*70*
*void PDF_begin_page(PDF *p, float width, float height)*	*70*
*void PDF_end_page(PDF *p)*	*70*
*float PDF_get_value(PDF *p, const char *key, float modifier)*	*71*
*void PDF_set_value(PDF *p, const char *key, float value)*	*71*
*void PDF_set_parameter(PDF *p, const char *key, const char *value)*	*71*

Text Functions

Function prototype	*page*
*int PDF_findfont(PDF *p, const char *fontname, const char *encoding, int embed)*	*72*
*void PDF_setfont(PDF *p, int font, float fontsize)*	*72*
*void PDF_show(PDF *p, const char *text)*	*72*
*void PDF_show2(PDF *p, const char *text, int len)*	*73*
*void PDF_show_xy(PDF *p, const char *text, float x, float y)*	*73*
*void PDF_show_xy2(PDF *p, const char *text, int len, float x, float y)*	*74*
*void PDF_continue_text(PDF *p, const char *text)*	*74*
*void PDF_continue_text2(PDF *p, const char *text, int len)*	*74*
*int PDF_show_boxed(PDF *p, const char *text, float x, float y, float width, float height, const char *mode, const char *reserved)*	*74*
*float PDF_stringwidth(PDF *p, const char *text, int font, float size)*	*75*
*float PDF_stringwidth2(PDF *p, const char *text, int len, int font, float size)*	*75*
*void PDF_set_text_pos(PDF *p, float x, float y)*	*75*

Graphics Functions

Function prototype	page
void PDF_setdash(PDF *p, float b, float w)	75
void PDF_setpolydash(PDF *p, float *darray, int length)	75
void PDF_setflat(PDF *p, float flatness)	75
void PDF_setlinejoin(PDF *p, int linejoin)	76
void PDF_setlinecap(PDF *p, int linecap)	76
void PDF_setmiterlimit(PDF *p, float miter)	76
void PDF_setlinewidth(PDF *p, float width)	76
void PDF_save(PDF *p)	77
void PDF_restore(PDF *p)	77
void PDF_translate(PDF *p, float tx, float ty)	77
void PDF_scale(PDF *p, float sx, float sy)	77
void PDF_rotate(PDF *p, float phi)	77
void PDF_skew(PDF *p, float alpha, float beta)	78
void PDF_concat(PDF *p, float a, float b, float c, float d, float e, float f)	78
void PDF_moveto(PDF *p, float x, float y)	78
void PDF_lineto(PDF *p, float x, float y)	78
void PDF_curveto(PDF *p, float x1, float y1, float x2, float y2, float x3, float y3)	78
void PDF_circle(PDF *p, float x, float y, float r)	78
void PDF_arc(PDF *p, float x, float y, float r, float start, float end)	79
void PDF_rect(PDF *p, float x, float y, float width, float height)	79
void PDF_closepath(PDF *p)	79
void PDF_stroke(PDF *p)	79
void PDF_closepath_stroke(PDF *p)	79
void PDF_fill(PDF *p)	79
void PDF_fill_stroke(PDF *p)	79
void PDF_closepath_fill_stroke(PDF *p)	79
void PDF_endpath(PDF *p)	80
void PDF_clip(PDF *p)	80

Color Functions

Function prototype	page
void PDF_setgray_fill(PDF *p, float gray)	80
void PDF_setgray_stroke(PDF *p, float gray)	80
void PDF_setgray(PDF *p, float gray)	80
void PDF_setrgbcolor_fill(PDF *p, float red, float green, float blue)	80
void PDF_setrgbcolor_stroke(PDF *p, float red, float green, float blue)	80
void PDF_setrgbcolor(PDF *p, float red, float green, float blue)	80

Image Functions

Function prototype	page
int PDF_open_image_file(PDF *p, const char *type, const char *filename, const char *stringparam, int intparam)	81
int PDF_open_CCITT(PDF *p, const char *filename, int width, int height, int BitReverse, int K, int BlackIs1)	81
int PDF_open_image(PDF *p, const char *type, const char *source, const char *data, long length, int width, int height, int components, int bpc, const char *params)	82
void PDF_close_image(PDF *p, int image)	83
void PDF_place_image(PDF *p, int image, float x, float y, float scale)	83

Hypertext Functions

Function prototype	page
int PDF_add_bookmark(PDF *p, const char *text, int parent, int open)	84
void PDF_set_info(PDF *p, const char *key, const char *value)	84
void PDF_attach_file(PDF *p, float llx, float lly, float urx, float ury, const char *filename, const char *description, const char *author, const char *mimetype, const char *icon)	85
void PDF_add_note(PDF *p, float llx, float lly, float urx, float ury, const char *contents, const char *title, const char *icon, int open)	86
void PDF_add_pdflink(PDF *p, float llx, float lly, float urx, float ury, const char *filename, int page, const char *dest)	86
void PDF_add_locallink(PDF *p, float llx, float lly, float urx, float ury, int page, const char *dest)	86
void PDF_add_launchlink(PDF *p, float llx, float lly, float urx, float ury, const char *filename)	87
void PDF_add_weblink(PDF *p, float llx, float lly, float urx, float ury, const char *url)	87
void PDF_set_border_style(PDF *p, const char *style, float width)	87
void PDF_set_border_color(PDF *p, float red, float green, float blue)	87
void PDF_set_border_dash(PDF *p, float b, float w)	87

Parameters and Values

category	set or get	val or par.	key
setup	set	par.	FontAFM, FontPFM, FontOutline, Encoding, prefix, resourcefile, compatibility, serial, warning
	set	value	compress
document	set	value	pagewidth, pageheight, flush
text	set	value	leading, textrise, horizscaling, textrendering, charspacing, wordspacing
	get	value	textx, texty, font, fontsize, capheight, ascender, descender
	set	par.	underline, overline, strikeout
	get	par.	fontname
graphics	set	par.	fillrule
	get	value	currentx, currenty
image	get	value	imagewidth, imageheight, resx, resy
hypertext	set	par.	openaction, transition, duration
	set	value	duration

Index

E

F

G

H

I

R

S

T

U

V

W

Z