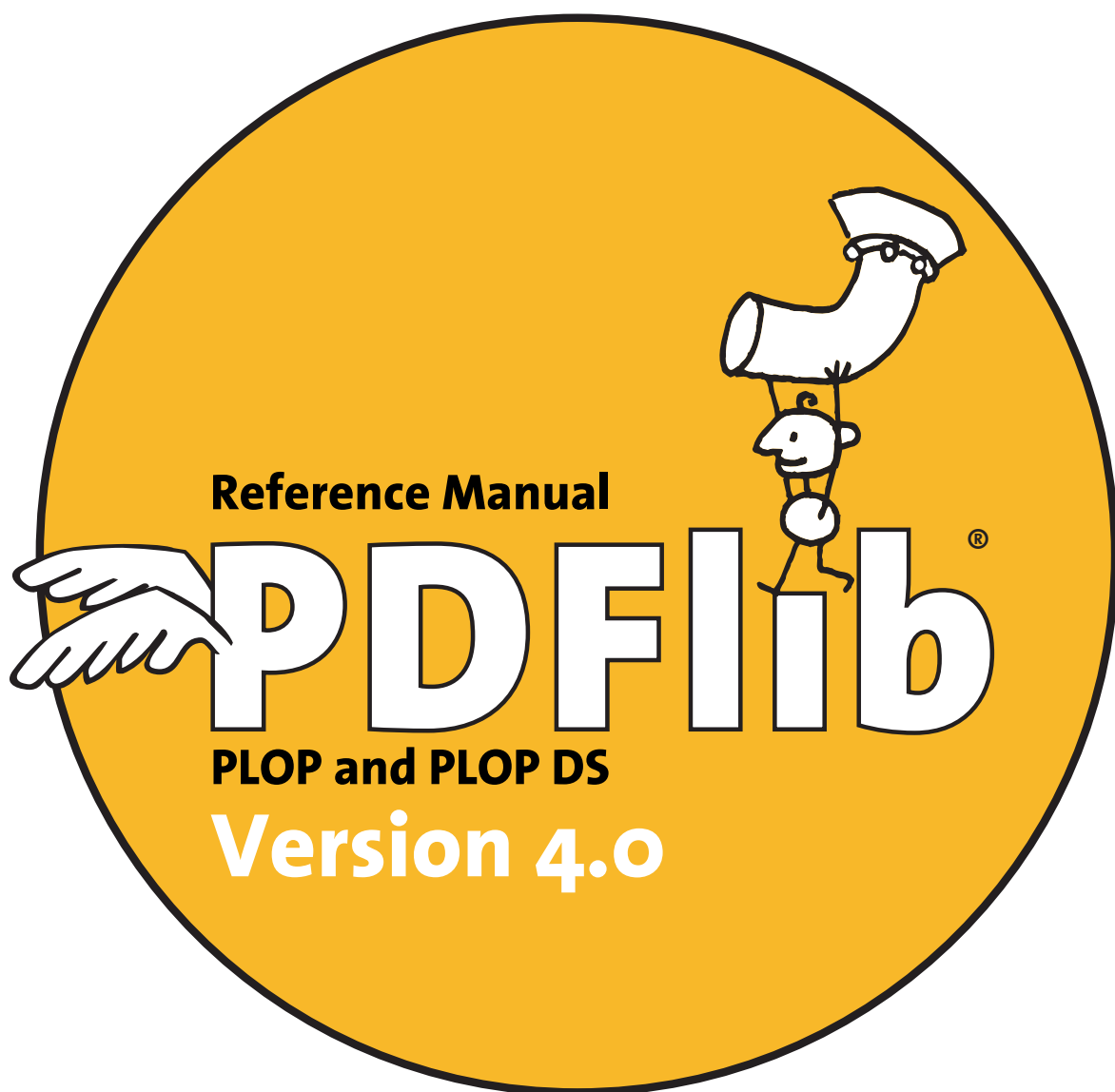


**PDFlib GmbH München, Germany**



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

Copyright © 2002-2009 PDFlib GmbH. All rights reserved.

PDFlib GmbH  
Franziska-Bilek-Weg 9, 80339 München, Germany  
[www.pdflib.com](http://www.pdflib.com)

phone +49 • 89 • 452 33 84-0  
fax +49 • 89 • 452 33 84-99

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

Licensing contact: [sales@pdflib.com](mailto:sales@pdflib.com)  
Customer support: [support@pdflib.com](mailto:support@pdflib.com) (please include your license number)

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

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

*Adobe, Acrobat, and PostScript are trademarks of Adobe Systems Inc. AIX, IBM, OS/390, WebSphere, iSeries, and zSeries are trademarks of International Business Machines Corporation. ActiveX, Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation. Apple, Macintosh and TrueType are trademarks of Apple Computer, Inc. Unicode and the Unicode logo are trademarks of Unicode, Inc. Unix is a trademark of The Open Group. Java and Solaris are a trademark of Sun Microsystems, Inc. Other company product and service names may be trademarks or service marks of others.*

*PDFlib PLOP and PLOP DS contain parts of the following third-party software:  
Zlib compression library, Copyright © 1995-2002 Jean-loup Gailly and Mark Adler  
Cryptographic software written by Eric Young, Copyright © 1995-1998 Eric Young ([ey@cryptsoft.com](mailto:ey@cryptsoft.com))  
Cryptographic software, Copyright © 1998-2002 The OpenSSL Project ([www.openssl.org](http://www.openssl.org))  
Independent JPEG Group's JPEG software, Copyright © 1991-1998, Thomas G. Lane  
Expat XML parser, Copyright © 1998, 1999, 2000 Thai Open Source Software Center Ltd*

*PDFlib PLOP and PLOP DS contain the RSA Security, Inc. MD5 message digest algorithm.*



# Contents

## o First Steps with PLOP/PLOP DS 5

- o.1 Installing the Software 5
- o.2 Applying the PLOP/PLOP DS License Key 6

## 1 PLOP and PLOP DS Features 9

- 1.1 Overview 9
- 1.2 Roadmap to Documentation and Samples 11
- 1.3 Encryption, Decryption, and Permissions 12
- 1.4 Web-Optimized (Linearized) PDF 13
- 1.5 Optimization (Size Reduction) 14
- 1.6 Repair Mode for damaged PDF 15
- 1.7 Query Document Information with pCOS 16
- 1.8 Inserting and Extracting Document Info Entries 17
- 1.9 Inserting and Extracting XMP Metadata 18
- 1.10 Digital Signatures with PLOP DS 19
- 1.11 PLOP Processing Details 20

## 2 PLOP and PLOP DS Command-line Tool 23

- 2.1 PLOP and PLOP DS Command-line Options 23
- 2.2 PLOP and PLOP DS Command-line Examples 26

## 3 PLOP and PLOP DS Library Language Bindings 27

- 3.1 C Binding 27
- 3.2 C++ Binding 29
- 3.3 COM Binding 30
- 3.4 Java Binding 31
- 3.5 .NET Binding 33
- 3.6 Perl Binding 34
- 3.7 PHP Binding 35
- 3.8 RPG Binding 37

## 4 PDF Security 39

- 4.1 Overview of PDF Security 39
- 4.2 Strength of PDF Encryption 41
- 4.3 PDF Security Features in PLOP 43
- 4.4 Securing PDF Documents with PLOP 46

<b>5</b>	<b>Digital Signatures with PLOP DS</b>	<b>49</b>
5.1	Basic Digital Signature Concepts	49
5.2	Obtaining and Managing Digital IDs	50
5.3	Signing PDF Documents with PLOP DS	52
5.4	Validating Digital Signatures with Acrobat	56
<b>6</b>	<b>The pCOS Interface</b>	<b>61</b>
6.1	Simple pCOS Examples	61
6.2	Handling Basic PDF Data Types	63
6.3	Composite Data Structures and IDs	65
6.4	Path Syntax	66
6.5	Pseudo Objects	68
6.6	Encrypted PDF Documents	75
<b>7</b>	<b>PLOP and PLOP DS Library API Reference</b>	<b>77</b>
7.1	Option Lists	77
7.2	General Functions	79
7.3	Document Input and Output Functions	81
7.4	Exception Handling	89
7.5	Option Handling	91
7.6	pCOS Functions	92
7.7	Unicode Conversion Functions	95
<b>A</b>	<b>Combining PDFlib with PLOP/PLOP DS</b>	<b>97</b>
<b>B</b>	<b>PLOP Library Quick Reference</b>	<b>98</b>
<b>C</b>	<b>Revision History</b>	<b>99</b>
	<b>Index</b>	<b>101</b>

# o First Steps with PLOP/PLOP DS

## o.1 Installing the Software

PLOP and PLOP DS are delivered as a combined installer package for Windows systems, and as a combined compressed archive for all other supported operating systems. The installer and the archive contain the PLOP/PLOP DS command-line tool and the PLOP/PLOP DS library, plus documentation and examples. After installing or unpacking the package the following steps are recommended:

- ▶ An introduction of PLOP and PLOP DS features is available in Chapter 1, »PLOP and PLOP DS Features«, page 9.
- ▶ Users of the PLOP/PLOP DS command-line tool can use the executable right away. The available options are discussed in Section 2.1, »PLOP and PLOP DS Command-line Options«, page 23, and are also displayed when you execute the PLOP command-line tool without any options.
- ▶ Users of the PLOP/PLOP DS library/component should read one of the sections in Chapter 3, »PLOP and PLOP DS Library Language Bindings«, page 27, corresponding to their environment of choice, and review the installed examples. On Windows the PLOP and PLOP DS programming examples are accessible via the Start menu.

If you obtained a commercial PLOP or PLOP DS license you must apply your license key according to the next page.

**Restrictions of the evaluation version.** The PLOP/PLOP DS command-line tool and library can be used as fully functional evaluation versions even without a commercial license. Unless a valid license key is applied, PLOP will include the text *unlicensed* in the output document's metadata and will insert an extra front page at the beginning of the document.

In some situations insertion of the front page may result in PDF output which no longer conforms to PDF/X or PDF/A even if the input conforms to one of these standards. The non-conformance is specific to the front page, and is not an issue once a valid license key is applied. In order to facilitate testing the front page will be suppressed if one or both of the following conditions are true:

- ▶ Encryption with the fixed strings *demo* or *DEMO* (options *userpassword* and *masterpassword*).
- ▶ Applying a digital signature with a digital ID where the subject name (also called *common name*, or *CN*) contains *demo* or *DEMO*; suitable digital IDs for testing are contained in all PLOP packages.

pCOS functions are restricted to small documents (less than 10 pages and less than 1 MB) in evaluation mode.

Unlicensed versions of PLOP or PLOP DS must not be used for production purposes, but only for evaluating the product. Using the software for production purposes requires a valid license.

## o.2 Applying the PLOP/PLOP DS License Key

Using PLOP/PLOP DS for production purposes requires a valid license key. Once you purchased a license you must apply your license key in order to get rid of the extra front page and enable the use of arbitrary passwords. There are several methods for applying the license key; choose one of the methods detailed below.

If the *frontpage* option for *PLOP\_set\_option()* is *false*, an exception will be thrown instead of creating the front page when no valid license key could be found.

*Note* PLOP/PLOP DS license keys are platform-dependent, and can only be used on the platform for which they have been purchased. While a PLOP DS license key activates all features of PLOP, a PLOP license key does not activate the signature features which are only available in PLOP DS.

**Enter the license key in the Windows installer.** Windows users can enter the license key when they install PLOP/PLOP DS using the supplied installer. This is the recommended method on Windows. If you do not have write access to the registry or cannot use the installer refer to one of the alternate methods below instead.

**Enter the license key in a license file.** Set an environment (shell) variable which points to a license file before PLOP/PLOP DS functions are called. If you are using the PLOP/PLOP DS library you can alternatively set the path to the license file by setting the *licensefile* parameter with the *PLOP\_set\_option()* function. The license file must be a text file with the following structure (lines beginning with a '#' characters contain comments, and will be ignored):

```
# Licensing information for PDFlib GmbH products
PDFlib license file 1.0
PLOP 4.0 ...your license key...
```

The details of setting environment variables vary across systems, but a typical statement for a Unix shell looks as follows:

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

On IBM eServer iSeries the license file can be specified as follows:

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

This command can be specified in the startup file *QSTRUP* and will work for all PDFlib GmbH products.

**Set the license key in an option for the PLOP/PLOP DS command-line tool.** If you use the PLOP/PLOP DS command-line tool you can supply an option which contains the name of a license file or the license key itself:

```
plop --ploptopt "license=...your license key..." ...more options...
plop --ploptopt "licensefile=/path/to/your/license/file" ...more options...
```

If the path name contains space characters you must enclose the path with braces:

```
plop --ploptopt "licensefile={/path/to/license file}" ...more options...
```

**Set the license key with a PLOP/PLOP DS library call.** If you use the PLOP/PLOP DS library, add a line to your script or program which sets the license key at runtime:

- In COM/VBScript:

```
oPLOP.set_option "license=...your license key..."
```

- In C:

```
PLOP_set_option(plop, "license=...your license key...");
```

- In C++, Java and C#:

```
plop.set_option("license=...your license key...");
```

- In Perl and PHP with the functional API:

```
PLOP_set_option($plop, "license=...your license key...");
```

- In PHP with the object-oriented API:

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

- In RPG:

```
d license      s              50  
c              eval      license='license=... your license key ...'+x'00'  
c              callp      PLOP_set_option(plop:license:0)
```

The *license* option must be set only once, immediately after instantiating the PLOP object, i.e., after calling *PLOP\_new()* (in C, Perl, RPG) or creating a PLOP object (in C++, COM, .NET, Java, or PHP).





# 1 PLOP and PLOP DS Features

## 1.1 Overview

PLOP is available in two flavors: the PLOP base product and the extended version PLOP DS.

**PLOP features.** PLOP supports the following kinds of PDF processing:

- ▶ Protection: encrypt a PDF document with a user or master password (or both); remove PDF encryption if you know the document's master password; add or remove permission settings (e.g., printing or text extraction not allowed) if you know the document's master password.
- ▶ Linearize PDF documents for enhanced viewer experience when retrieving PDF files from a Web server (see below).
- ▶ Optimize the size of PDF documents by reducing redundant objects.
- ▶ Repair damaged PDF documents.
- ▶ Use the integrated pCOS interface to query information about the document's security status (encrypted with user or master password), permission settings, document metadata, and many other properties.
- ▶ Insert and retrieve predefined or custom document information entries.
- ▶ Insert and retrieve XMP metadata.

PLOP is PDF/A-aware: if the input document conforms to the PDF/A standard, the output will either conform to PDF/A as well (for most operations), or the operation will be rejected if it would result in non-conforming PDF/A output (e.g. encryption). PLOP is also PDF/X-aware in a similar way.

**PLOP DS features.** PLOP DS offers all features of PLOP, plus the ability to apply digital signatures to PDF documents. The signatures can be validated in Adobe Acrobat and Adobe Reader.

Signatures can be created from digital IDs in the PKCS#12 and PFX certificate formats. On Windows digital IDs from the Windows certificate store can be used. On Windows and some other platforms cryptographic tokens with PKCS#11 support can be used (e.g. a smartcard or USB stick).

**Advantages.** PLOP/PLOP DS offer the following advantages:

- ▶ All PLOP and PLOP DS operations are PDF/X- and PDF/A-aware: if the input conforms to one of these standards, the output is guaranteed to conform to the same standard if possible. If this is not possible (e.g. encryption was requested for PDF/A input) the operation will either be rejected or the standard identification removed.
- ▶ PLOP is a standalone tool which does not require any third-party software for reading, encrypting, signing, or writing PDF.
- ▶ PLOP can technically and legally be deployed on a server, is fully thread-safe, and has been checked for memory leaks. PLOP has been engineered for heavy server usage, and can be used in Web server environments, for high-volume batch processing, etc.
- ▶ PLOP is available on many platforms and for several programming environments.
- ▶ For added flexibility, PLOP is available both as a command-line tool and a programming library (component) for various development languages.

**PLOP/PLOP DS command-line tool or library?** PLOP/PLOP DS is available both as a programming library (component) for various development languages, and as a command-line tool for batch operations. Both offer the same feature set, but are suitable for different deployment tasks. Here are some guidelines for choosing among the library and the command-line tool:

- ▶ The command-line PLOP/PLOP DS tool is suited for batch processing PDF documents. It doesn't require any programming, but offers powerful command-line options which can be used to integrate it into complex workflows. The PLOP/PLOP DS command-line tool can also be called from environments which do not support the use of the library.
- ▶ The PLOP/PLOP DS programming library integrates well into a variety of common development environments, such as Active Server Pages (ASP), Visual Basic, Java (including servlets), PHP, RPG, and plain C or C++ application development.

The PLOP/PLOP DS license covers both the command-line tool and the library.

## 1.2 Roadmap to Documentation and Samples

**Mini samples for the PLOP language bindings.** The PLOP distribution contains a number of simple programming examples for all supported language bindings. These demonstrate basic PLOP library programming tasks:

- ▶ The *encrypt* sample encrypts an unencrypted PDF document with user and master password.
- ▶ The *decrypt* sample decrypts an encrypted PDF document using its master password.
- ▶ The *noprint* sample sets the *noprint* and *nocopy* access permissions, and encrypts the file with a master password.
- ▶ The *dumper* sample uses the pCOS interface to collect general properties, information about the encryption status of a file as well as document information and XMP metadata.
- ▶ The *insertxmp* sample reads XMP metadata from a file, and inserts the XMP in a PDF document. Sample XMP files are supplied for testing.
- ▶ The *linearize* sample applies linearization to an existing PDF document, and changes a document info entry.

Optimization is implicitly demonstrated by all samples since the optimization process is enabled by default.

The following mini samples are for use with PLOP DS:

- ▶ The *sign* sample shows how to apply a digital signature to an existing PDF document.
- ▶ The *hellosign* shows how to dynamically create a document with PDFlib and pass it to PLOP (in memory), which then applies a digital signature to it. Note that this example requires the PDFlib product which is not included in the PLOP package. Free evaluation packages for PDFlib are available from our Web site, however.

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

**Sample calls of the PLOP command-line tool.** The PLOP command-line tool supports various options which are documented in Section 2.1, »PLOP and PLOP DS Command-line Options«, page 23. The remaining sections in Chapter 1, »PLOP and PLOP DS Features«, page 9, as well as Section 2.2, »PLOP and PLOP DS Command-line Examples«, page 26 and other chapters contain sample calls of the PLOP command-line tool.

**pCOS Cookbook.** The *pCOS Cookbook* is a collection of code fragments for the pCOS interface which is integrated in PLOP. It is available at the following URL: [www.pdfliib.com/pcos-cookbook](http://www.pdfliib.com/pcos-cookbook).

Details of the pCOS interface are documented in Chapter 6, »The pCOS Interface«, page 61.

## 1.3 Encryption, Decryption, and Permissions

Encrypting and decrypting PDF documents as well as permission restrictions are covered in detail in Chapter 4, »PDF Security«, page 39. In the current section we will only provide a quick summary and some initial examples.

**Querying security settings.** With the pCOS programming interface, which is integrated in the PLOP library, you can query various security settings of a PDF document. The required function calls and parameters can be seen in the *dumper* mini sample, which is included in all PLOP packages. The corresponding option for the PLOP command-line tool is `--info` (see Section 1.7, »Query Document Information with pCOS«, page 16, for an example).

**Encrypting documents with PLOP.** You can encrypt documents by specifying the *user-password* or *masterpassword* option (or both) for `PLOP_create_file()`. Note that a user password always requires a master password, but not vice versa. Sample code for encrypting PDF documents can be seen in the *encrypt* mini sample, which is included in all PLOP packages. The equivalent options for the PLOP command-line tool are `--user` and `--master`.

Example: encrypt a file with user password *demo* and master password *DEMO*:

```
plop --user demo --master DEMO --outfile encrypted.pdf input.pdf
plop -u demo -m DEMO -o encrypted.pdf input.pdf
```

**Specify permission restrictions with PLOP.** You can specify the permission restrictions in the *permissions* option for `PLOP_create_file()` which supports various keywords (see Table 4.3, page 44). Sample code for specifying permission restrictions of PDF documents can be seen in the *noprint* mini sample, which is included in all PLOP packages. The equivalent option for the PLOP command-line tool is `--permissions`. Note that permission restrictions always require a master password.

Example: encrypt a document with the master password *DEMO*, and disallow printing the document and copying contents:

```
plop --master DEMO --permissions "noprint nocopy" --outfile encrypted.pdf input.pdf
plop -m DEMO --permissions "noprint nocopy" -o encrypted.pdf input.pdf
```

**Decrypting documents with PLOP.** You can decrypt documents by specifying the appropriate user or master password in the *password* option for `PLOP_create_file()`. Full sample code for decrypting PDF documents can be seen in the *decrypt* mini sample, which is included in all PLOP packages. The equivalent option for the PLOP command-line tool is `--password`.

Example: decrypt a single file with the master password *DEMO*. All access restrictions which may have been applied to the input document will be removed (since the output is unencrypted):

```
plop --password DEMO --outfile decrypted.pdf encrypted.pdf
plop -p DEMO -o decrypted.pdf encrypted.pdf
```

More encryption and decryption examples can be found in Section 4.4, »Securing PDF Documents with PLOP«, page 46.

## 1.4 Web-Optimized (Linearized) PDF

PLOP can apply a process called linearization to PDF documents. The resulting property is called *Fast Web View* in Acrobat. Linearization reorganizes the objects within a PDF file and adds supplemental information which can be used for faster access.

While non-linearized PDFs must be fully transferred to the client, a Web server can transfer linearized PDF documents one page at a time using a process called byte-serving. It allows Acrobat (running as a browser plugin) to retrieve individual parts of a PDF document separately. The result is that the first page of the document will be presented to the user without having to wait for the full document to download from the server. This provides enhanced user experience.

Note that the Web server streams PDF data to the browser, not PLOP. Instead, PLOP prepares the PDF files for byteserving. All of the following requirements must be met in order to take advantage of byteserving PDFs:

- ▶ The PDF document must be linearized, which can be achieved with PLOP. Linearization can be applied along with encryption or decryption in a single run. In Acrobat you can check whether a file is linearized by looking at its document properties («Fast Web View: yes«).
- ▶ The Web server must support byteserving. The underlying byterange protocol is part of HTTP 1.1 and therefore implemented in all current Web servers.
- ▶ The user must use Acrobat as a Browser plugin, and have page-at-a-time download enabled in Acrobat (Acrobat 6/7/8/9: *Edit, Preferences, [General...], Internet, Allow fast web view*). Note that this is enabled by default.

The larger a PDF file (measured in pages or MB), the more it will benefit from linearization when delivered over the Web.

Linearization and encryption/decryption can be applied in combination. However, in order to linearize a protected file you must provide the proper master password (see Table 4.2).

*Note Linearizing a PDF document generally slightly increases its file size due to the additional linearization information. This increase may or may not be compensated by the applied optimization techniques (see Section 1.5, »Optimization (Size Reduction)«, page 14).*

**Linearizing PDF documents with PLOP.** You can enable the linearization step with the *linearize* option for `PLOP_create_file()`. Sample code for linearizing PDF documents can be seen in the *linearize* mini sample, which is included in all PLOP packages.

The equivalent option for the PLOP command-line tool is `--webopt`. Example: linearize all PDF documents in a directory (assuming these do not require any password), and copy the resulting files to the target directory *output*. Verbosity level 2 prints the names of all input and output files as they are processed:

```
plop --verbose 2 --webopt --targetdir output *.pdf
plop -v 2 -w -t output *.pdf
```

## 1.5 Optimization (Size Reduction)

While processing PDF documents PLOP can apply file optimization in addition to other operations:

- ▶ PLOP detects multiple instances of identical data, and removes all instances but one. This is mostly relevant for fonts and images, but may affect other data types as well, e.g. ICC profiles or even complete pages with identical content. An embedded font or image will be removed if another font or image contains the exact same data; all references to the removed data will be replaced with references to the remaining instance of the font or image. For example, if a document has been assembled from several PDFs containing parts of a document and all of these parts contain the same embedded font, the resulting combined PDF may carry excess font data. PLOP will reduce the redundant font data, and keep only one instance of the font.
- ▶ Unused objects will be removed from the PDF file in a process known as *garbage collection*. In some cases (when the *Save* menu item in Acrobat has been used, as opposed to *Save As...*) Acrobat will append changes to a file while retaining the previous state of the document. PLOP removes all objects related to older versions of the document.
- ▶ The output will be written using compact syntax. For example, unnecessary white-space will be removed, certain inefficient constructs (indirect integer objects) will be replaced with more efficient equivalents, and hexadecimal strings (e.g. color palettes for indexed color spaces) will be replaced with more compact binary representations.

PLOP will never apply any optimization steps which could result in loss of information (e.g. unembedding fonts, downsampling images). All relevant information for viewing or printing the document in the exact same quality of the input will be retained in the output.

**Optimizing PDF documents with PLOP.** Since optimization in PLOP is enabled by default, there is no need to supply any option to activate it. However, for extreme performance requirements you can disable the optimization step with the *optimize=none* option for *PLOP\_create\_file()*. The equivalent option for the PLOP command-line tool is *-fast*.

Example: optimize a document with the PLOP command-line tool:

```
plop --outfile optimized.pdf input.pdf
plop -o optimized.pdf input.pdf
```

## 1.6 Repair Mode for damaged PDF

PLOP implements a repair mode for damaged PDF so that even certain kinds of damaged documents can be processed. However, in rare cases a damaged PDF document may be rejected if PLOP is unable to repair it.

**Repairing PDF documents with PLOP.** The repair mode is activated automatically when PLOP encounters damaged input. However, using the *repair=force* option of *PLOP\_open\_document()* you can enforce the repair mode even if no problems occurred when opening the document. The equivalent option for the PLOP command-line tool is *--inputopt repair=force*. You can disable the repair mode with *repair=none*.

Example: force reconstruction of a document with the PLOP command-line tool:

```
plop --inputopt repair=force --outfile repaired.pdf damaged.pdf  
plop --inputopt repair=force -o repaired.pdf damaged.pdf
```

## 1.7 Query Document Information with pCOS

The pCOS interface is covered in detail in Chapter 6, »The pCOS Interface«, page 61. In the current section we will only provide a quick summary and some initial examples.

With the pCOS programming interface, which is integrated in the PLOP library, you can query various properties of a PDF document. Sample code for querying document information with pCOS can be seen in the *dumper* mini sample, which is included in all PLOP packages. The corresponding option for the PLOP command-line tool is *--info*.

Example: display security and other information about a PDF document:

```
plop --info *.pdf
plop -i *.pdf
```

This program call will result in output similar to the following:

```
File name: PLOP-manual.pdf
PDF version: 1.4
Encryption: No encryption
  Master pw: false
  User pw: false
  nocopy: false (copying is allowed)
  nomodify: false (adding form fields and other changes is allowed)
  noannots: false (adding or changing comments or form fields is allowed)
  noassemble: false (insert/delete/rotate pages, creating bookmarks is allowed)
  noforms: false (filling form fields is allowed)
  noaccessible: false (extracting text or graphics for accessibility is allowed)
  nohiresprint: false (high-resolution printing is allowed)
plainmetadata: true (metadata is not encrypted)
  Linearized: true
PDF/X status: none
PDF/A status: none
  Tagged PDF: false
  Signatures: 0
Reader-enabled: false

No. of pages: 90
No. of fonts: 8
  embedded Type 1 CFF font TheSans-Plain
  embedded Type 1 CFF font TheSansExtraBold-Plain
  ...more fonts...

CreationDate: 'D:20070616003116Z'
Subject: 'PDFlib PLOP: PDF Linearization, Optimization, Protection'
Author: 'PDFlib GmbH'
Creator: 'FrameMaker 7.0'
Producer: 'Acrobat Distiller 8.1.0 (Windows)'
ModDate: 'D:20070616021141Z'
  Title: 'PDFlib PLOP and PLOP DS Manual'

XMP meta data: is present
```



## 1.8 Inserting and Extracting Document Info Entries

PDF supports two kinds of document metadata which contain general information about a document – document info entries and XMP metadata.

Document info entries are keys with associated strings that hold some unstructured information. The predefined info keys *Subject*, *Title*, *Author*, and *Keywords* are commonly used, but arbitrary custom keys can be defined for specific purposes. Document information entries are considered the old and simple kind of PDF metadata.

With PLOP you can add new document information entries or replace the values of existing info entries. Both predefined or custom entries can be set. If the input document contains XMP document metadata, all predefined info entries will automatically be synchronized to the XMP metadata in order to keep the metadata consistent.

**Inserting document info entries with PLOP.** You can set document info entries with the *docinfo* option for *PLOP\_create\_file()*. Sample code for setting document info entries can be seen in the *linearize* mini sample (in addition to linearization this sample demonstrates how to set document info), which is included in all PLOP packages.

Example: specify the predefined document info entry *Subject* and the custom info entry *Department*; note the braces around *Product Manual* to protect the space character:

```
docinfo={Department Techdoc Subject {Product Manual}}
```

This option can be supplied to the PLOP command-line tool via the *--outputopt* option as follows:

```
plop --outputopt "docinfo={Department Techdoc Subject {Product Manual}}" --outfile output.pdf input.pdf  
plop --outputopt "docinfo={Department Techdoc Subject {Product Manual}}" -o output.pdf input.pdf
```

**Extracting document info entries with PLOP.** With the pCOS programming interface, which is integrated in the PLOP library, you can extract document information entries (keys and values) from a PDF document. The required function calls and parameters can be seen in the *dumper* mini sample, which is included in all PLOP packages.

The corresponding option for the PLOP command-line tool is *--info* (see Section 1.7, »Query Document Information with pCOS«, page 16, for an example).

## 1.9 Inserting and Extracting XMP Metadata

XMP (*Extensible Metadata Platform*<sup>1</sup>) is an XML framework with many predefined properties. However, as the name implies, XMP can be extended to satisfy specific requirements using custom extension schemas. XMP is much more powerful than document information entries, and is for example required in the PDF/A standard. Many industry groups have published standards based on XMP for various vertical applications, e.g. digital imaging or prepress data exchange.

You can find more detailed information on XMP as well as links to other resources at [www.pdflib.com/developer/xmp-metadata](http://www.pdflib.com/developer/xmp-metadata).

With PLOP you can insert XMP metadata in PDF documents, or extract XMP from PDF. Inserted XMP will be validated to make sure that valid output can be created. If the input document conforms to the PDF/A-1 standard, the user-supplied XMP must conform to the XMP rules set forth in PDF/A. Again, these rules (including XMP extension schema validation) will be checked by PLOP to make sure that PDF/A-1 input plus user-supplied XMP will result in conforming PDF/A output.

XMP insertion with PLOP can be used in the following and many other situations (the names of sample XMP files in the PLOP distribution are provided in parenthesis):

- ▶ Add XMP metadata to PDF/A-1 documents, including support for XMP extension schemas as defined in the PDF/A-1 standard (*machine\_pdfa1.xmp*).
- ▶ Add XMP metadata describing the scan process for digitized legacy documents (*engineering.xmp*).
- ▶ Add XMP metadata according to the Ghent Workgroup (GWG) Ad Ticket scheme, (*gwg\_ad\_ticket.xmp*). For more details see [www.gwg.org/Jobtickets.phtml](http://www.gwg.org/Jobtickets.phtml).
- ▶ Add company-specific XMP metadata (*acme.xmp*).

**Inserting XMP metadata with PLOP.** In order to insert metadata you must create a file which contains valid XMP metadata in UTF-8 format. You can insert XMP with the *metadata* option for *PLOP\_create\_file()*, which supports several suboptions. Sample code for inserting XMP in PDF documents is available in the *insertxmp* mini sample, which is included in all PLOP packages.

Example: insert XMP metadata from a file called *gwg.xmp*, where the XMP is validated against the XMP 2004 standard:

```
plop --outputopt "metadata={filename=gwg_ad_ticket.xmp validate=xmp2004}" --outfile output.pdf input.pdf  
plop --outputopt "metadata={filename=gwg_ad_ticket.xmp validate=xmp2004}" -o output.pdf input.pdf
```

**Extracting XMP metadata with PLOP.** With the pCOS programming interface, which is integrated in the PLOP library, you can extract XMP metadata from a PDF document. The required function calls and parameters can be seen in the *dumper* mini sample, which is included in all PLOP packages. Note that the sample code in the *dumper* sample does not actually print the XMP metadata, but simply reports the size of the XMP found in the document.

The PLOP command-line tool can not be used for extracting XMP metadata. We offer a powerful pCOS command-line tool for extracting information from PDF.

1. See [www.adobe.com/products/xmp](http://www.adobe.com/products/xmp)

## 1.10 Digital Signatures with PLOP DS

The ability to digitally sign PDF documents is only available in PLOP DS, but not in the PLOP base product.


Digital signatures for PDF documents are covered in detail in Chapter 5, »Digital Signatures with PLOP DS«, page 49. In the current section we will only provide a quick summary and some initial examples.

**Querying signature properties.** With the pCOS programming interface, which is integrated in the PLOP library, you can query signature settings of a PDF document. The required function calls and parameters are available in the pCOS Cookbook topic *interactive\_elements/signatures*. The corresponding option for the PLOP command-line tool is `--info` (see Section 1.7, »Query Document Information with pCOS«, page 16).



**Signing documents with PLOP DS.** Applying a signature requires a digital ID, which may be available as a file, in the Windows certificate store, or on a cryptographic token (e.g. a smartcard or USB stick). While the former requires a password for accessing the digital ID, the Windows certificate store is usually protected by the Windows login and does not require any password. Cryptographic tokens are often protected by a PIN.

You can apply a digital signature with the `sign` option for `PLOP_create_file()`, which supports several suboptions. Sample code for signing PDF documents is available in the `sign` mini sample, which is included in all PLOP packages. The equivalent option for the PLOP command-line tool is `--signopt`. The `hellosign` mini sample shows how to dynamically create PDF documents with PDFlib and then apply a signature with PLOP DS.



Examples: Create an invisible signature for a PDF document using a digital ID from the file `demo1024.p12`. The password for the digital ID is contained in the file `pw.txt`:

```
plop --signopt "digitalid={filename=demo1024.p12} passwordfile=pw.txt"   
--outfile signed.pdf input.pdf  
plop -S "digitalid={filename=demo1024.p12} passwordfile=pw.txt" -o signed.pdf input.pdf
```

(Windows only) Create an invisible signature for a PDF document using a certificate from the Windows Certificate Store (from the default store *My*). This assumes that the digital ID is protected by your Windows login so that no password must be supplied:

```
plop --signopt "engine=miscapi digitalid={certstore={store=My subject={DEMO PLOP User 1024}}}"   
--outfile signed.pdf input.pdf  
plop -S "engine=miscapi digitalid={certstore={store=My subject={DEMO PLOP User 1024}}}"   
-o signed.pdf input.pdf
```

(Only platforms with PKCS#11 support, e.g. Windows) Create an invisible signature for a PDF document using a digital ID from a cryptographic token. The PKCS#11 interface for the token is implemented in the library `cryptoki.dll` which must be provided by the smartcard supplier. The password for the digital ID is contained in the file `pw.txt`:

```
plop --signopt "engine=pkcs#11 digitalid={filename=cryptoki.dll} passwordfile=pw.txt"   
--outfile signed.pdf input.pdf  
plop -S "engine=pkcs#11 digitalid={filename=cryptoki.dll} passwordfile=pw.txt"   
-o signed.pdf input.pdf
```

More signature examples can be found in Chapter 5.3, »Signing PDF Documents with PLOP DS«, page 52.

## 1.11 PLOP Processing Details

**Acceptable input documents.** All valid PDF files up to PDF 1.7 will be accepted and processed by PLOP, subject to the availability of the password for the desired operation. PDF documents created with Acrobat 9 (technically: PDF 1.7 Adobe Extension Level 3) can also be processed with the exception of AES-256 encryption. PLOP will attempt to repair various kinds of damaged PDF documents.

See below for restrictions which apply to certain combinations of input documents and requested operations.

**PDF version.** The PDF version number of the generated output document will never be less than the PDF version number of the input document, but it may be forced to a higher number as detailed below. The PDF output version can be specified with the *compatibility* option of `PLOP_create_file()` or the `--outpuopt` option of the PLOP command-line tool. If this option has not been specified, PLOP will use the PDF version of the input document, modified according to the following rules:

- ▶ Encryption, i.e. any of the options *userpassword*, *masterpassword*, and *permissions*, pushes the version to PDF 1.4.
- ▶ Digitally signing (option *sign*) the document pushes the version to PDF 1.3.
- ▶ Inserting XMP metadata (option *metadata*) pushes the version to PDF 1.4 unless the input conforms to PDF/X-1:2001, PDF/X-1a:2001, or PDF/X-3:2002. In these cases the version number will be left unchanged. The PDF/X exception is designed to simplify applications of the Ghent Workgroup's Ad Ticket XMP application.
- ▶ The *plainmetadata* keyword for the *permissions* option pushes the version to PDF 1.5.

**Sacrificing certain properties of the input PDF.** Conflicts can arise between several PDF document properties and certain PLOP operations. For example, PDF/A documents are not allowed to use encryption. What should PLOP do when encryption is requested for PDF/A input? By default it will throw an exception and refuse the operation. However, you can use the option *sacrifice* for `PLOP_create_file()` or the `--outpuopt` option of the PLOP command-line tool to give the requested operation priority over the input property. In the example above, the PDF/A conformance entry will be removed from the document in order to allow encryption.

There are several combinations of input document properties and requested operations. In all of these combinations you can use the *sacrifice* option to allow an operation by sacrificing a particular document property (see Table 7.3, page 83, for details):

- ▶ PDF/A: PLOP applies digital signatures in a PDF/A-compliant manner: input documents which conform to the PDF/A-1a or PDF/A-1b standard are guaranteed to be PDF/A-compliant after signing. However, applying encryption, i.e. any of the options *userpassword*, *masterpassword*, and *permissions*, is not allowed for PDF/A documents since PDF/A prohibits any encryption. You can sacrifice PDF/A compliance with the *sacrifice={pdfa}* option, though.
- ▶ PDF/X: PDF/X does not allow encryption, or visible signature fields on the page. In these situations PLOP will raise an exception, and you can sacrifice PDF/X compliance with the *sacrifice={pdfx}* option.
- ▶ Existing signatures (including certification signatures) in the input document will not be kept. In order to avoid destroying existing signatures, PLOP will refuse to sign documents which already contain one or more signatures. You can sacrifice existing

signatures with the option *sacrifice={signature}* to *PLOP\_create\_file()* or the *--outputopt* option of the PLOP command-line tool.

- ▶ PLOP cannot apply signatures if the document contains form fields without Appearances (e.g. form fields created with PDFlib 6 or 7), and will therefore throw an exception for this kind of input. The reason is that Acrobat will have to rebuild the missing appearance streams for form fields, which would instantly invalidate the signature. You can sacrifice all existing form fields in this situation with the option *sacrifice={fields}* to *PLOP\_create\_file()* or in the *--outputopt* option of the PLOP command-line tool.
- ▶ If an unencrypted document contains encrypted file attachments for which the password is not available, processing will stop by default. You can sacrifice all encrypted file attachments in this situation with the option *sacrifice={encryptedattachments}* to *PLOP\_create\_file()* or in the *--outputopt* option of the PLOP command-line tool. All encrypted file attachments for which the password is not available will be removed with this option.

**Properties of the input document which are generally lost.** The following properties of the input document will be lost after applying any PLOP operation:

- ▶ If the input document is linearized, the linearization will be lost by default. In order to linearize the output, supply the *linearize* option to *PLOP\_create\_file()* or the *--linearize* option to the PLOP command-line tool.
- ▶ Reader-enabled documents: processing Reader-enabled PDF documents with PLOP will result in output which is not Reader-enabled. Since Reader-enabled documents can only be created with Adobe software there is no workaround for this.

**Temporary disk space requirements.** PLOP reads an input PDF document and writes an output PDF. The output document will require roughly the same amount of disk space as the input document (unless PLOP's optimizing step removes redundant information). In many cases no additional disk space will be required. However, PLOP/PLOP DS require additional temporary disk space for its operation if linearization or digital signature are enabled.

Temporary files will be created in the current directory by default, but this can be changed with the *tempdirname* option of *PLOP\_create\_file()*. The disk space for temporary data roughly equals the size of the input file. If linearization is requested in combination with in-core PDF generation (i.e., no output file name supplied), PLOP requires temporary disk space with roughly two times the size of the input.

**Performance.** Since PLOP's operating performance depends on the PDF document's size, complexity, and internal structure there are no fixed speed ratings. However, in general you can expect throughput of up to several hundred pages per second, or several MB of PDF input data per second on a current system. Linearization and optimization will slightly increase processing times compared to simply encrypting, decrypting, or signing a document.



# 2 PLOP and PLOP DS Command-line Tool

## 2.1 PLOP and PLOP DS Command-line Options

The combined command-line tool for PLOP and PLOP DS allows you to encrypt, decrypt, optimize, repair, and sign one or more PDF documents without the need for any programming. In addition, it can be used to query the status of PDF documents. The PLOP program can be controlled via a number of command-line options. It is called as follows for one or more input PDF files (items in square brackets are optional):

```
plop --help
plop [ <general options> ] --info [ --outfile <filename> ] <filename> ...
plop [ <general options> ] <transform options> --outfile <filename> <filename>
plop [ <general options> ] <transform options> --targetdir <pathname> <filename>...
```

The PLOP command-line tool is built on top of the PLOP library. By default, PLOP will repair input documents which are found to be damaged, and will optimize the output for smallest file size. You can supply library options using the *--inputopt*, *--outputopt*, and *--plopopt* options according to the option tables in Chapter 7, »PLOP and PLOP DS Library API Reference«, page 77. Table 2.1 lists all PLOP command-line options.

Table 2.1 PLOP command-line options

option	parameters	function
--		End the list of options; this is useful in case file names start with a - character.
@filename <sup>1</sup>		Specify a response file with options; for a syntax description see »Response files«, page 25. Response files will only be recognized before the -- option and before the first filename, and can not be used to replace the parameter for another option.
--compatibility, -c	<version>	Set the document's PDF version to one of the strings 1.4, 1.5, 1.6, or 1.7 for Acrobat 5, 6, 7, or 8. If the output is to be encrypted, the strongest possible encryption algorithm supported by the selected PDF version will be used (use 1.6 to force AES encryption). The selected PDF version may be increased automatically by other options according to the following rules: <ul style="list-style-type: none"><li>► Digitally signing requires at least PDF 1.3.</li><li>► The permission settings noforms, noaccessible, noassemble or nohires-print require at least PDF 1.4.</li><li>► The plainmetadata setting requires at least PDF 1.5.</li></ul> Default: the PDF version of the input document, or a higher version as mandated by the rules above.
--fast, -f		Disable optimization step for faster processing.
--help, -? (or no option)		Display help with a summary of available options.
--info, -i		Display status information for the input file; no PDF output will be produced.
--inmemory		Load the input file(s) into memory and process it from there. This can result in a significant performance gain on some systems.
--inputopt	<option list>	Additional option list for PLOP_open_document() (see Table 7.2, page 81)

Table 2.1 PLOP command-line options

option	parameters	function
<b>--master<sup>2,3</sup>, -m</b>	<password>	Output master password; missing option means no password.
<b>--noreplace, -n</b>		If the output file already exists, it will not be overwritten and an exception will be thrown. Default: existing output files will be overwritten.
<b>--outfile, -o</b>	<filename>	(Requires exactly one input document except with --info; one of --outfile and --targetdir must be supplied) Output file name; input and output file name must be different.
<b>--outputopt</b>	<option list>	Additional option list for <b>PLOP_create_file()</b> (see Table 7.3, page 83)
<b>--password<sup>2</sup>, -p</b>	<password>	User or master password for input document(s). This password will be used for all input documents. Input documents which require different passwords must be processed in separate program calls.
<b>--permissions<sup>2,3</sup></b>	<permissions>	(Requires --master) The access permission list for the output document. It contains any number of the noprint, nomodify, nocopy, noannots, noassemble, noforms, noaccessible, nohiresprint, and plainmetadata keywords (see Table 4.3, page 44). In addition, the following keyword can be used (default: no permission restrictions): <b>keep</b> Keep the permission settings of the input document. This setting can be amended by additional keywords in order to modify the permission settings of the input PDF, e.g. keep noprint.
<b>--ploptopt</b>	<option list>	Additional option list for <b>PLOP_set_option()</b> (see Table 7.5, page 91). This can be used to pass the license or licensefile options.
<b>--resize, -R</b>	<blocksize>	(MVS only) The record size of the output file. Default: o (unblocked)
<b>--tempfilename, -T</b>	<filename>	(MVS only) Full file name for a temporary file for PLOP's internal processing. If empty, PLOP will generate a unique temporary file name. The user is responsible for deleting the temporary file when PLOP finished. Default: empty
<b>--tempdirname</b>	<dirname>	Name of a directory where temporary files needed for PLOP's internal processing will be created. If empty, PLOP will generate temporary files in the current directory. Default: empty
<b>--searchpath, -s<sup>1</sup></b>	<path>	Name of a directory where files will be searched. The path must not start with a minus character »-« (prepend ./ if required). Default: current directory
<b>--signopt, -S</b>	<option list>	(Only available in PLOP DS) Option list for the sign option of <b>PLOP_create_file()</b> for digitally signing documents (see Table 7.4, page 86).
<b>--targetdir, -t</b>	<dirname>	(One of --outfile and --targetdir must be supplied) Output directory name; the directory must already exist.
<b>--user, -u<sup>2,3</sup></b>	<password>	Output user password; missing option means no password.
<b>--verbose, -v</b>	0, 1, 2, 3	Verbosity level (default: 1): <b>0</b> no output <b>1</b> only errors <b>2</b> errors and file names <b>3</b> detailed reporting
<b>--webopt, -w</b>		Linearize the PDF output for Web delivery. Linearization can be combined with other processing options, or used in a stand-alone manner. Default: no linearization

1. This option can be supplied more than once.  
2. This option will be used for all input files.  
3. This option triggers output encryption; if any of these is supplied PLOP will encrypt the output.



**Constructing PLOP command lines.** The following rules must be obeyed for constructing PLOP command lines:

- ▶ Input files will be searched in all directories specified as *searchpath*.
- ▶ Short forms are available for some options, and can be mixed with long options.
- ▶ Long options can be abbreviated provided the abbreviation is unique (e.g. *--plop* instead of *--plopt*).
- ▶ If an option is supplied more than once only the last instance will be taken into account. However, this rule does not hold for options which are marked as repeatable in Table 2.1.
- ▶ Depending on the encryption status of the input file, a user or master password may be required for processing. This must be supplied with the *--password* option. PLOP will check whether this password is sufficient for the requested action (see Table 4.2), and will generate an error if it isn't.

PLOP checks the full command line before processing any file. If an option syntax error is encountered in the options anywhere on the command line, no files will be processed at all.

If a file cannot be processed (e.g. because the required password is missing), an error message will be created, and PLOP will continue processing the remaining files.

**File names.** File names which contain blank characters require some special handling when used with command-line tools like PLOP. In order to process a file name with blank characters you should enclose the complete file name with double quote " characters. Wildcards can be used according to standard practice. For example, *\*.pdf* denotes all files in a given directory which have a *.pdf* file name suffix. Note that on some systems case is significant, while on others it isn't (i.e., *\*.pdf* may be different from *\*.PDF*). Also note that on Windows systems wildcards do not work for file names containing blank characters.

**Response files.** In addition to options supplied directly on the command-line, options can also be supplied in a response file. The contents of a response file will be inserted in the command-line at the location where the *@filename* option was found.

A response file is a simple text file with options and parameters. It must adhere to the following syntax rules:

- ▶ Option values must be separated with whitespace, i.e. space, linefeed, return, or tab.
- ▶ Values which contain whitespace must be enclosed with double quotation marks: "
- ▶ Double quotation marks at the beginning and end of a value will be omitted.
- ▶ A double quotation mark must be masked with a backslash to use it literally: \"
- ▶ A backslash character must be masked with another backslash to use it literally: \\

Response files can be nested, i.e. the *@filename* syntax can be used in another response file.

**Exit codes.** The PLOP command-line tool returns with an exit code which can be used to check whether or not the requested operations could be successfully carried out:

- ▶ Exit code 0: all command-line options and input files could be successfully and fully processed.
- ▶ Exit code 1: one or more file processing errors occurred, but processing continued.
- ▶ Exit code 2: some error was found in the command-line options. Processing stopped at the particular bad option, and no documents have been processed.

## 2.2 PLOP and PLOP DS Command-line Examples

The following examples demonstrate some useful combinations of PLOP command-line options. All samples are shown in two variations; the first uses the long format of all options, while the second uses the equivalent short option format. More examples are available in the following sections:

- ▶ Chapter 1, »PLOP and PLOP DS Features«, page 9 (various sections)
- ▶ Section 4.4, »Securing PDF Documents with PLOP«, page 46
- ▶ Section 5.3, »Signing PDF Documents with PLOP DS«, page 52.

Display security and other information about all PDF files in the current directory:

```
plop --info *.pdf
plop -i *.pdf
```

Linearize all PDF documents in a directory (assuming these do not require any password), and copy the resulting files to the target directory *output*. Since optimization is enabled by default, linearizing a file will at the same time optimize its size. Verbosity level 2 prints the names of all input and output files as they are processed:

```
plop --verbose 2 --webopt --targetdir output *.pdf
plop -v 2 -w -t output *.pdf
```

Encrypt all files in the current directory with the same user password *demo* and master password *DEMO*, and place the resulting files in the target directory *output*:

```
plop --targetdir output --user demo --master DEMO *.pdf
plop -t output -u demo -m DEMO *.pdf
```

Create an invisible signature for a PDF document, using a digital ID from the file *demo1024.p12*. The password for the digital ID is contained in the file *pw.txt*:

```
plop --signopt "digitalid={filename=demo1024.p12} passwordfile=pw.txt" ←
    --outfile signed.pdf input.pdf
plop -S "digitalid={filename=demo1024.p12} passwordfile=pw.txt" -o signed.pdf input.pdf
```

# 3 PLOP and PLOP DS Library Language Bindings

In this chapter we will discuss language-specific aspects of the PLOP/PLOP DS library.

## 3.1 C Binding

**Linking a Program against the PLOP/PLOP DS Library.** In order to use the library within your C program, you must include the PLOP header file *ploplib.h* in your source code, and must link your executable against the library:

- ▶ On Unix systems this is achieved by using a command similar to the following (assuming the library *libplop.a* and the header file *ploplib.h* are available in the current directory):

```
cc encrypt.c -o encrypt -I. -L. -lplop -lm
```

- ▶ On Windows systems PLOP/PLOP DS is delivered as a DLL. The DLL should be placed in the Windows system directory. Microsoft Visual C++ project files for the C samples are contained in the PLOP distribution. You must link your executable against the following libraries:

```
libplop.lib, advapi32.lib
```

**Exception Handling in C.** Since the C language doesn't provide any means for structured exception handling (like *try/catch* in C++ or Java) the PLOP library implements a private scheme based on functions and macros. This scheme makes the advantages of structured exception handling (in particular, not having to clutter one's code with conditionals to check for error situations) available to traditional C programs. PLOP exception handling is easy to use once a few basic rules are obeyed:

- ▶ The bulk of calls to the PLOP library should be bracketed with calls to *PLOP\_TRY()* and *PLOP\_CATCH()*.
- ▶ *PLOP\_delete()* must never be called within a *PLOP\_TRY()* block.
- ▶ If any of the API functions called in the *PLOP\_TRY()* block triggers an exception, program execution within the current *PLOP\_TRY()* block will stop, and will instead continue at the first statement in the subsequent *PLOP\_CATCH()* block. This block may contain code to deal with the error situation, can also query detailed information about the error from PLOP, and can call *PLOP\_delete()* to clean up the PLOP library. However, no library calls with the current input or output documents must be issued.
- ▶ *PLOP\_TRY()* / *PLOP\_CATCH()* blocks may be nested, and PLOP exceptions can be re-thrown to the enclosing block (see below).

*Note It is possible to write PLOP library client programs without any TRY/CATCH clauses. In this case a default exception handler will be called when an exception occurs. However, this is not recommended practise. Production software should always use PLOP's exception handling.*

**Volatile variables.** Special care must be taken regarding variables that are used in both the *PLOP\_TRY()* and the *PLOP\_CATCH()* blocks. Since the compiler doesn't know about the

control transfer from one block to the other, it might produce inappropriate code (e.g., register variable optimizations) in this situation. Fortunately, there is a simple rule to avoid these problems:

*Note Variables used in both the `PLOP_TRY()` and `PLOP_CATCH()` blocks should be declared volatile.*

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

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

```
PLOP_TRY(plop)                /* outer try block */
{
    /* ... */

    PLOP_TRY(plop)             /* inner try block */
    {
        /* ... */
    }
    PLOP_CATCH(plop)           /* inner catch block */
    {
        /* error cleanup */
        PLOP_RETHROW(plop);
    }
    /* ... */
}
PLOP_CATCH(plop)              /* outer catch block */
{
    /* more error cleanup */
    PLOP_delete(plop);
}
```

The `PLOP_RETHROW()` invocation in the inner error handler will transfer program execution to the first statement of the outer `PLOP_CATCH()` block immediately.

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

```
PLOP_TRY(plop)
{
    /* ... */

    if (error_condition)
    {
        PLOP_EXIT_TRY(plop);
        return -1;
    }
}
PLOP_CATCH(plop)
{
    /* error cleanup */
    PLOP_RETHROW(plop);
}
```

## 3.2 C++ Binding

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

Using the C++ object wrapper replaces the functional approach with API functions and *PLOP\_* prefixes in all PLOP function names with a more object-oriented approach: a *PLOP* object offers methods, and the method names no longer have the *PLOP\_* prefix.

The PLOP C++ binding will package Unicode text in standard C++ strings in UTF-16 format. Clients must be prepared to process such strings appropriately.

## 3.3 COM Binding

**Installing the PLOP Edition for COM.** Install PLOP/PLOP DS with the supplied Windows Installer. The installer will make appropriate registry entries, and register the PLOP component with Windows so that it can be used from any COM-compatible program.

**Exception Handling in COM.** The PLOP/PLOP DS component implements standard COM exception behavior, and will throw a COM exception with an explanatory message. PLOP users can use standard programming means to catch the exception and react on it.

**Using the PLOP COM Edition with .NET.** As an alternative to PLOP.NET (see Section 3.5, »NET Binding«, page 33) the COM edition of PLOP can also be used with .NET. First, you must create a .NET assembly from the PLOP COM edition using the *tlbimp.exe* utility:

```
tlbimp plop_com.dll /namespace:plop_com /out:Interop.plop_com.dll
```

You can use this assembly within your .NET application. If you add a reference to *plop\_com.dll* from within Visual Studio .NET an assembly will be created automatically.

The following code fragment shows how to use the PLOP COM edition with VB.NET:

```
Imports plop_com
...
Dim p As plop_com.IPDF
...
p = New PLOP()
...
buf = p.get_buffer()
```

The following code fragment shows how to use the PLOP COM edition with C#:

```
using plop_com;
...
static plop_com.IPDF p;
...
p = New PLOP();
...
buf = (byte[])p.get_buffer();
```

The rest of your code works as with the .NET version of PLOP. Please note that in C# you have to cast the result of *get\_buffer()* since there is no automatic conversion from the VARIANT data type returned by the COM object here.

## 3.4 Java Binding

**Installing the PLOP Edition for Java.** PLOP/PLOP DS has been implemented as a native C library which attaches to Java via the JNI (Java Native Interface). Obviously, for developing Java applications you will need the JDK which includes support for the JNI. For the PLOP binding to work, the Java VM must have access to the PLOP Java wrapper library and the PLOP Java package.

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

```
com.pdflib.plop
```

This package is available in the *plop.jar* file and contains a single class called *plop*. Last-minute comments on using PLOP in various Java development environments may be found in the *readme.txt* file.

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

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

You can check the value of this property as follows:

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

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

- ▶ Unix: The library *libplop\_java.so* must be placed in one of the default locations for shared libraries, or in an appropriately configured directory.
- ▶ Mac OS X: The library *libplop\_java.jnilib* must be placed in one of the default locations for shared libraries, or in an appropriately configured directory.
- ▶ Windows: The library *plop\_java.dll* must be placed in the Windows system directory, or a directory which is listed in the *PATH* environment variable.

**PLOP servlets and Java application servers.** PLOP/PLOP DS is perfectly suited for server-side Java applications, especially servlets. When using PLOP with a specific servlet engine the following configuration issues must be observed:

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

Examples for using PLOP within servlets are contained in the PLOP distribution.

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

**Exception Handling in Java.** All PLOP/PLOP DS methods will throw an exception of type *PLOPlibException* in case of an error. PLOP users can use standard Java language features to catch the exception and react on it:

```
try {
    plop plop;
    /* ... PLOP statements ... */

} catch (PLOPlibException e) {
    System.err.println("encrypt: PLOP Exception occurred:");
    System.err.println(e.get_apiname() + ": " + e.getMessage());
} finally {
    /* delete the PLOP object */
    if (plop != null) plop.delete();
}
```



## 3.5 .NET Binding

The .NET edition of PLOP supports all relevant .NET concepts. In technical terms, the PLOP.NET edition is a C++ class (with a managed wrapper for the unmanaged PLOP core library) which runs under control of the .NET framework. It is packaged as a static assembly with a strong name. The PLOP assembly (*PLOPlib\_dotnet.dll*) contains the actual library plus meta information.

*Note PLOP.NET requires the .NET Framework 2.0 or above.*

**Installing the PLOP Edition for .NET.** Install PLOP with the supplied Windows MSI installer. The PLOP.NET MSI installer will install the PLOP assembly plus auxiliary data files, documentation and samples on the machine interactively. The installer will also register PLOP so that it can easily be referenced on the .NET tab in the *Add Reference* dialog box of Visual Studio .NET.

**Installing PLOP.NET for ASP.NET.** In order to use PLOP.NET in your ASP.NET scripts you must make the PLOP.NET assembly available to ASP. This can be achieved by placing *PLOPlib\_dotnet.dll* in the *bin* subdirectory of your IIS installation (if it doesn't exist you must manually create it), or the *bin* directory of your Web application, e.g.

```
C:\inetpub\wwwroot\bin\PLOPlib_dotnet.dll or  
C:\inetpub\wwwroot\WebApplicationX\bin\PLOPlib_dotnet.dll
```

**Special considerations for ASP.NET.** When using external files ASP's *MapPath* facility must be used in order to map path names on the local disk to paths which can be used within ASP.NET scripts. Take a look at the ASP.NET samples supplied with PLOP, and the ASP.NET documentation if you are not familiar with *MapPath*. Don't use absolute path names in ASP.NET scripts since these may not work without *MapPath*.

The directory containing your ASP.NET scripts must have execute permission, and also write permission unless the in-core method for generating PDF is used (the supplied ASP samples use in-core PDF generation).

**Trust levels in ASP.NET 2.0 and above.** ASP.NET 2.0 introduced some restrictions regarding the allowed operations in various trust levels for Web applications. Since PLOP.NET contains unmanaged code, it requires *Full Trust* level. PLOP.NET applications cannot be deployed in ASP.NET 2.0 applications with any other trust level, including High or Medium Trust.

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

In order to convey exception-related information PLOP defines its own exception class *PLOPlib\_dotnet.PLOPlibException* with the members *get\_errnum*, *get\_errmsg*, and *get\_apiname*.

## 3.6 Perl Binding

**Installing the PLOP Edition for Perl.** PLOP/PLOP DS is implemented as a C library which can dynamically be attached to Perl. This requires Perl to be built with support for loading extensions at runtime. The name of the PLOP Perl extension is *ploplib\_pl*.

**Exception Handling in Perl.** When a PLOP exception occurs, a Perl exception is thrown. It can be caught and acted upon using an *eval* sequence:

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

## 3.7 PHP Binding

**Installing the PLOP Edition for PHP.** PLOP/PLOP DS is implemented as a C library which can dynamically be attached to PHP. PLOP supports several versions of PHP. Depending on the version of PHP you use you must choose the appropriate PLOP library from the unpacked PLOP archive.

Detailed information about the various flavors and options for using PLOP with PHP, including the question of whether or not to use a loadable PLOP module for PHP, can be found in the *PDFlib-in-PHP-HowTo* document which can be found on the PDFlib Web site. Although it is mainly targeted at using PDFlib with PHP the discussion applies equally to using PLOP with PHP.

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

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

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

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

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

This will display a long info page about your current PHP configuration. On this page check the section titled *plop*. If this section contains the phrase

```
PDFlib PLOP (PDF Linearization, Optimization, Protection) => enabled
```

(plus the PLOP version number) you successfully installed PLOP for PHP.

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

```
dl("libplop_php.so");          # for Unix
dl("libplop_php.dylib");       # for Mac OS X
dl("libplop_php.dll");         # for Windows
```

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

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

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

```
try {
...some PLOP instructions...
```

```
} catch (PLOplibException $e) {  
    print "PLOP exception occurred:\n";  
    print "[" . $e->get_errnum() . "]" " " . $e->get_apiname() . ": "  
        $e->get_errmsg() . "\n";  
}  
catch (Exception $e) {  
    print $e;  
}
```

## 3.8 RPG Binding

**Installing the PLOP Edition for RPG.** PLOP/PLOP DS provides a */copy* module that defines all prototypes and some useful constants needed to compile ILE-RPG programs with embedded PLOP functions.

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

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

```
d/copy QRPGLSRC,PLOPLIB
```

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

```
d/copy plopsrclib/QRPGLSRC,PLOPLIB
```

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

```
CRTBNDDIR BNDDIR(PLOPLIB/PLOPLIB) TEXT('PLOPlib Binding Directory')
```

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

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

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

```
CRTBNDRPG PGM(PLOPLIB/ENCRYPT) SRCFILE(PLOPLIB/QRPGLSRC) SRCMBR(*PGM) DFTACTGRP(*NO) BNDDIR(PLOPLIB/PLOPLIB)
```

**Exception Handling in RPG.** PLOP clients written in ILE-RPG can use a limited form of PLOP's try/catch mechanism as follows:

```
c          eval      rtn=plop_try(plop)
c          if        PLOP_create_file(plop:out_filename:0:optlist)=--1
c          or plop_catch(plop)=1
c      *
c          callp     PLOP_delete(plop)
c          eval      error='Couldn't open output file '+
c                  %trim(out_filename)
c          exsr      exit
c      endif
```



# 4 PDF Security

## 4.1 Overview of PDF Security

PDF encryption features have been introduced with Acrobat 2 (PDF 1.1), based on the RC4 encryption algorithm and 40-bit keys. They have been considerably improved in Acrobat 5 with 128-bit keys, and again in Acrobat 7 with the introduction of AES encryption (Advanced Encryption Standard) with 128-bit keys. The majority of applications use Acrobat's standard security features (symmetric encryption), as opposed to self-sign (public-key) encryption or third-party PDF encryption tools. In this manual we will always refer to standard security which works in the full Acrobat product and the free Adobe Reader as well as in third-party PDF viewers. Standard security offers the following protection features:

- ▶ The user password (also referred to as open password) is required to open the file for on-screen viewing.
- ▶ The master password (also referred to as owner or permissions password) is required to change any security settings, i.e. permissions, user or master password. Files with user and master passwords can be opened for viewing by supplying either one.
- ▶ Permission settings restrict certain actions for the PDF document, such as printing or extracting text (see below).

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

**Permission settings.** Acrobat supports the following permission settings which can be granted or denied individually (a few settings depend on others, though):

- ▶ *Printing:* If printing is not allowed, the print button in Acrobat will be disabled. Acrobat supports a distinction between high-resolution and low-resolution printing. Low-resolution printing generates a bitmapped image of the page which is suitable only for personal use, but prevents high-quality reproduction and re-distilling. Note that bitmap printing not only results in low output quality, but will also considerably slow down the printing process.
- ▶ *General Editing:* If this is disabled, any document modification is prohibited. Content extraction and printing are allowed.
- ▶ *Content Copying and Extraction:* If this is disabled, selecting document contents and copying it to the clipboard for repurposing the contents is prohibited. The accessibility interface also is disabled. If you need to use Catalog and Search with such documents you must select the *Certified Plugins Only* preference in Acrobat.
- ▶ *Authoring Comments and Form Fields:* If this is disabled, adding, modifying, or deleting comments and form fields is prohibited. Form field filling is allowed.
- ▶ *Form Field Fill-in or Signing:* If this is enabled, users can sign and fill in forms, but not create form fields.
- ▶ *Content Accessibility Enabled:* Allow accessibility software (such as a screenreader) to use the document contents.
- ▶ *Document Assembly:* If this is disabled, inserting, deleting or rotating pages, or creating bookmarks and thumbnails is prohibited.

**Plaintext metadata.** PDF 1.5 (Acrobat 6) introduced a new feature called plaintext metadata. With this feature encrypted documents can contain unencrypted metadata to aid search engines in retrieving document metadata even from encrypted documents.

In order to display or modify a document's security settings in Acrobat, click *File, Properties..., Security, Show Details...* or *Change Settings...*, respectively. Figure 4.1 shows the security settings dialog in Acrobat.

**Encrypted file attachments.** In PDF 1.6 (Acrobat 7) and above file attachments can be encrypted even in otherwise unprotected documents.

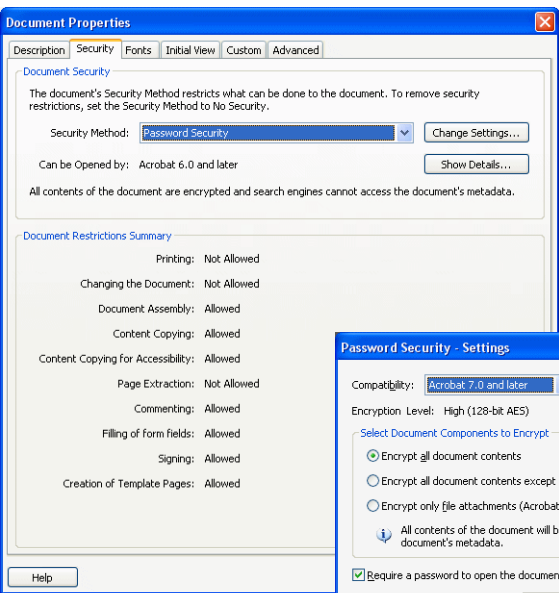
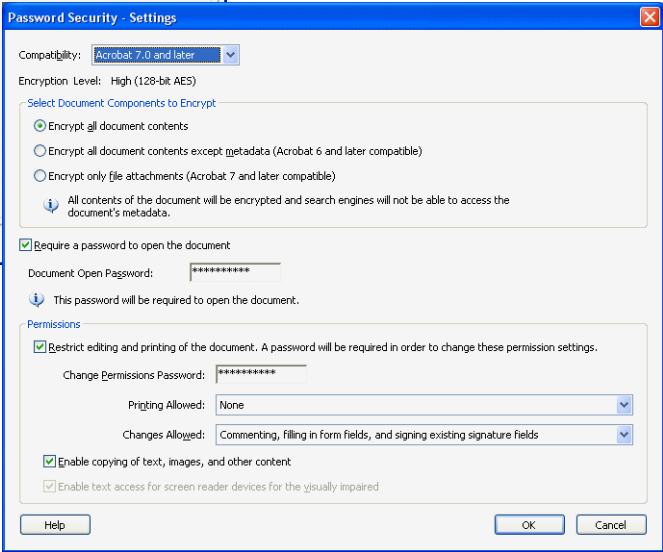


Fig. 4.1  
Viewing (left) and setting (below)  
standard security settings in Acrobat





## 4.2 Strength of PDF Encryption

**Key lengths.** PDF encryption doesn't use the user or master passwords directly for scrambling the document contents, but calculates an encryption key from the password and other parameters including the permission settings. While the passwords may be up to 32 characters in length, the length of the encryption key used for actually encrypting the document is independent from the length of the password: for PDF versions up to and including 1.3 (Acrobat 4) the key length is 40 bits, while for PDF 1.4 (Acrobat 5) and above the key length is 128 bits. Encryption with 40-bit keys can be cracked, and should therefore not be used.

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

While PDF encryption internally works with 40-bit or 128-bit encryption keys, on the user level passwords of up to 32 characters are used. The internal key which is used to encrypt the PDF document is derived from the user-supplied password by applying some complicated calculations. If the password is weak, the resulting protection will be weak as well, regardless of the key length. Even 128-bit keys and AES encryption are not secure if short passwords are used. We will discuss this in more detail below.

**Enforcing permission settings.** Setting some access restriction, such as *printing prohibited* will disable the respective menu item in Acrobat (for example, the *File, Print...* menu item and corresponding toolbar icon will be disabled). However, this not necessarily holds true for third-party PDF viewers. It is up to the developer of PDF tools whether or not access permissions in a document will actually be honored. Indeed, several PDF tools are known to ignore permission settings altogether. Commercially available PDF cracking tools can be used to immediately disable any access restrictions. This has nothing to do with cracking the encryption; there is simply no way that a PDF file without any password can make sure it won't be printed while it still remains viewable. PDF document authors should always keep this in mind. This is actually documented in Adobe's own PDF reference:

*»There is nothing inherent in PDF encryption that enforces the document permissions specified in the encryption dictionary. It is up to the implementors of PDF consumer applications to respect the intent of the document creator by restricting user access to an encrypted PDF file according to the permissions contained in the file.«*

**Cracking protected PDF.** There are commercial tools available for cracking protected PDFs using a brute-force or dictionary attack against either the encryption key or one of the user or master passwords. A brute-force attack simply tries all possible password or key combinations, while a dictionary attack checks all entries in a list of well-known words, such as an English dictionary or a list of names. In our tests we found that short passwords can easily and quickly be cracked. Table 4.1 details the cracking times for finding an unknown master password and various password lengths (in characters)

with a single computer. Note that distributed attacks with a large number of computers are much faster. For comparison we tried both »easy« passwords consisting only of upper- and lowercase characters, as well as better passwords which make use of the set of all printable characters. Table 4.1 demonstrates that short passwords are very insecure. Remember that permission restrictions can be disabled immediately if no user password has been set.

Table 4.1 Time required to crack a protected PDF file (AES encryption with 128-bit keys) with a brute-force attack on the password for various key and password lengths on a PC with Intel Core 2 with 2 GHz

password quality	user password length				
	no user password (only permission restrictions)	up to 4 chars Avoid!	up to 5 chars Avoid!	up to 6 chars Avoid!	> 6 chars
only upper- and lowercase Avoid!	immediately	4 min.	5 hours	10 days	years
all printable characters	immediately	1 hour	years	years	years

**Security Recommendations.** The facts presented in the preceding sections will help you in selecting secure settings for protecting your PDF documents. For maximum security observe the following suggestions:

- ▶ Passwords should be longer than six characters, and should contain non-alphabetic characters. Passwords should not resemble your spouse’s or pet’s name, birthday etc. Cracking documents with short or weak passwords requires only minutes or hours.
- ▶ Do not rely on access permissions for documents without any user password, since the permission settings cannot be reliably enforced.

## 4.3 PDF Security Features in PLOP

PLOP applies or removes Acrobat standard security features to or from PDF files. PLOP can apply user and master passwords, and set access permissions to prevent printing the document with Acrobat, extracting text, modifying the document, etc. PLOP uses the RC4 and AES encryption algorithms (depending on the selected PDF version). In order to decrypt a document the appropriate master password is required.

**Encryption strength.** When creating encrypted output the strongest possible encryption algorithm supported by the target PDF version will be used. You can control the encryption strength by specifying the PDF version number in the *compatibility* option for `PLOP_create_file()`:

- ▶ PDF 1.4 and 1.5: RC4 encryption with 128-bit keys will be used.
- ▶ PDF 1.6 and above: AES encryption with 128-bit keys will be used.

It is widely known that 40-bit keys are not secure. PLOP therefore works with 128-bit encryption keys only, and never applies 40-bit keys for encryption. 40-bit-encrypted documents are acceptable as input, however.

*Note PLOP does not currently support AES-256 encryption per PDF 1.7 Adobe Extension Level 3/ Acrobat 9.*

**Required passwords for various operations.** In order to strictly obey the author's intentions as reflected by a PDF document's security settings, not all operations on encrypted documents may be allowed. PLOP acts according to the following rules:

- ▶ Querying the encryption status is always possible, regardless of any password.
- ▶ Querying document info fields of encrypted documents works with either user or master password. However, if the document doesn't require a user password, document info fields can always be queried.
- ▶ Changing or removing the user password, master password, or permission settings requires the master password.
- ▶ Linearizing, optimizing, repairing, or signing an encrypted document (see Section 1.4, »Web-Optimized (Linearized) PDF«, page 13) requires the master password.

Table 4.2 summarizes the requirements for all operations.

Table 4.2 Required passwords for various operations on encrypted documents

known passwords	query encryption status	query document info	change passwords or permissions	linearize, optimize, repair, or sign
none	yes	only if no user password is set	no	no
user	yes	yes	no	no
master	yes	yes	yes	yes

**Setting Passwords with PLOP.** In the PLOP library API and the PLOP command-line options we refer to the original PDF document as the *input* document, and the encrypted or decrypted result as the *output* document (although both may end up with the same file name). If the input document is protected, PLOP will require either the user or master password depending on the desired operation (see Table 4.2). If the input document could successfully be opened (either because it was unprotected, or because the proper

password was supplied) any combination of user password, master password, and permission settings can be applied to the output document. However, PLOP interacts with the client-supplied passwords for the output document in the following ways:

- ▶ If a user password or permission settings, but no master password has been supplied, a regular user would easily be able to change the security settings, thereby defeating any protection. For this reason PLOP considers this situation as an error.
- ▶ If the user and master password are the same, a distinction between user and owner of the file would no longer be possible, again defeating effective protection. PLOP considers this situation as an error.
- ▶ For both user and master passwords up to 32 characters can be used.

**Setting Permissions with PLOP.** PLOP can be used to query, set or remove any of the permission settings detailed in Table 4.3. Unless specified otherwise, all actions will be allowed by default. Specifying access restrictions will disable the respective feature in Acrobat. Access restrictions can be applied without setting any user password, but at least a master password is required. Table 4.3 lists all access restriction keywords used in PLOP.

Table 4.3 Access restriction keywords

keyword	explanation
<b>noprint</b>	Acrobat will prevent printing the file.
<b>nomodify</b>	Acrobat will prevent users from adding form fields or making any other changes.
<b>nocopy</b>	Acrobat will prevent copying and extracting text or graphics, and will disable accessibility
<b>noannots</b>	Acrobat will prevent adding or changing comments or form fields.
<b>noforms<sup>1</sup></b>	Acrobat will prevent form field filling, even if noannots hasn't been specified. Setting this restriction implies noannots automatically.
<b>noaccessible<sup>1</sup></b>	Acrobat will prevent extracting text or graphics for accessibility purposes (e.g. a screenreader)
<b>noassemble<sup>1</sup></b>	Acrobat will prevent inserting, deleting, or rotating pages and creating bookmarks and thumbnails, even if nomodify hasn't been specified. Setting this restriction implies nomodify automatically.
<b>nohiresprint<sup>1</sup></b>	Acrobat will prevent high-resolution printing. If noprint hasn't been specified printing is restricted to the »print as image« feature which prints a low-resolution rendition of the page.
<b>plain-metadata<sup>2</sup></b>	Keep document metadata unencrypted even for encrypted documents (see below)

1. Pushes the PDF output version number to PDF 1.4 (requires Acrobat 5 or above)

2. Pushes the PDF output version number to PDF 1.5 (requires Acrobat 6 or above)

**Non-ASCII characters in passwords.** Attention must be paid when characters outside the range 0x20-0x7E are used in passwords, i.e. characters which are not in the traditional ASCII character set. As an example, let's take a look at the use of the character Ä within a password. On the Mac this character has code 0x80, while on Windows it is encoded as 0xC4. Since users expect the file to be opened when using the password Ä on either platform, Acrobat converts the supplied password to an internal encoding (called PDFDocEncoding) before applying the password. Characters which are not available in this encoding will be mapped to the *space* character. PDFDocEncoding contains all characters of the Mac and Windows platforms, but requires several characters to be converted. In the example above, when the user encrypts the file with password Ä on the Mac,

PLOP would be unable to decrypt the file if the code for Å would be used directly. PLOP therefore applies the same password conversion as Acrobat in order to make sure that files encrypted with Mac or Windows versions of Acrobat can successfully be decrypted. Upon decryption PLOP will automatically detect the required conversion:

- ▶ WinAnsi to PDFDocEncoding conversion if the document was encrypted with Acrobat on Windows.
- ▶ MacRoman to PDFDocEncoding conversion if the document was encrypted with Acrobat on the Mac;
- ▶ No conversion if the document was encrypted with some other software.

When encrypting files, PLOP will act like Acrobat on Windows and interpret the supplied passwords in WinAnsi encoding, i.e., it will apply a WinAnsi to PDFDocEncoding conversion to the supplied user and master passwords; on EBCDIC platforms it will apply EBCDIC to WinAnsi conversion prior to that.

**What you can't do with PLOP.** It is important to realize that there are certain operations on encrypted documents which are technically feasible, but which are nevertheless unsupported in PLOP because they would violate the document author's intentions:

- ▶ PLOP is not a cracker tool – it cannot be used to gain access to protected documents without knowing the (appropriate user or master) password.
- ▶ PLOP does not allow you to change permission settings without having the master password.
- ▶ PLOP does not allow you to change the user or master password without knowing the master password.
- ▶ PLOP does not read any document information fields from encrypted documents without having either the user or master password.
- ▶ PLOP supports Acrobat standard security, but not public key encryption (e.g. Acrobat self-sign security), or any third-party encryption or digital rights management systems for PDF (such as FileOpen).
- ▶ PLOP is not a Digital Rights Management (DRM) system: you can't tie a document to individual computers, users, or CPUs.

## 4.4 Securing PDF Documents with PLOP

You can encrypt documents by specifying the *userpassword* or *masterpassword* option (or both) for `PLOP_create_file()`. Note that a user password always requires a master password, but not vice versa. Full sample code for securing PDF documents and removing security with the PLOP library can be seen in the *encrypt* and *decrypt* programming samples, which are included in all PLOP packages. The equivalent options for the PLOP command-line tool are `--user` and `--master`. Permission restrictions can be specified with the *permissions* option for `PLOP_create_file()`; the equivalent option for the PLOP command-line tool is `--permissions`.

### Encryption examples.

Encrypt a file with user password *demo* and master password *DEMO*:

```
plop --user demo --master DEMO --outfile encrypted.pdf input.pdf
plop -u demo -m DEMO -o encrypted.pdf input.pdf
```

Encrypt all files in the current directory with the same user password *demo* and master password *DEMO*, and place the resulting files in the target directory *output*:

```
plop --targetdir output --user demo --master DEMO *.pdf
plop -t output -u demo -m DEMO *.pdf
```

Passwords which contain space characters must be enclosed in braces as in the following example: encrypt a document with the master password *two words*:

```
plop --master {two words} --outfile encrypted.pdf input.pdf
plop -m {two words} -o encrypted.pdf input.pdf
```

### Decryption examples.

Decrypt a single file with the master password *DEMO*. All access restrictions which may have been applied to the input document will be removed (since the output is unencrypted):

```
plop --password DEMO --outfile decrypted.pdf encrypted.pdf
plop -p DEMO -o decrypted.pdf encrypted.pdf
```

### Re-encrypt with stronger crypto.

PLOP can be used to apply stronger encryption to documents which are encrypted with short keys or weak passwords. You must supply the old and the new password. Selecting PDF 1.6 output compatibility activates strong AES encryption. The following example assumes that the input is encrypted with the master password *old*, and the output will be AES-encrypted with the master password *DEMO*. The new password can even be the same as the old password. Of course you should only use really strong passwords (see Section 4.2, »Strength of PDF Encryption«, page 41), not short ones as in this example:

```
plop -c 1.6 -p old -m DEMO -o strong.pdf weak.pdf
plop --compatibility 1.6 --password old --master DEMO --outputfile strong.pdf weak.pdf
```

### Permission settings.

Apply the master password *DEMO* and the permission settings *noprint*, *nocopy*, and *noannots* to all files in a directory, and copy the resulting files to the target directory *output*. AES encryption will be used (forced by PDF version 1.6), regardless of the encryp-

tion used in the input documents. Verbosity level 2 prints the names of all input and output files as they are processed:

```
plop --verbose 2 --compatibility 1.6 --master DEMO ↵  
    --permissions "noprint nocopy noannots" --targetdir output *.pdf  
plop -v 2 -c 1.6 -m DEMO --permissions "noprint nocopy noannots" -t output *.pdf
```

Remove all permission restrictions from a file, and copy the result to a different output file with the same master password. This requires the master password for the input document:

```
plop --password DEMO --master DEMO --outfile unrestricted.pdf protected.pdf  
plop -p DEMO -m DEMO -o unrestricted.pdf protected.pdf
```

Re-encrypt a document (e.g. to replace weak encryption with strong AES encryption or weak passwords with better ones), and clone the permission settings of the input document. Copy the result to a different output file. This requires the master password for the input document:

```
plop --password DEMO --master LONGPASSWORD --permissions keep ↵  
    --outfile unrestricted.pdf protected.pdf  
plop -p DEMO -m LONGPASSWORD --permissions keep -o unrestricted.pdf protected.pdf
```





# 5 Digital Signatures with PLOP DS

*Note The ability to digitally sign PDF documents is only available in PLOP DS, but not in the PLOP base product.*

## 5.1 Basic Digital Signature Concepts

Explaining the details of digital signatures is beyond the scope of this manual. However, we will list the most important concepts which play a role when digitally signing PDF documents with PLOP DS.

Digital signatures are based on Public Key Cryptography, also called asymmetric encryption. It works with a private key which is only available to the person who signs a document, and a public key which is available to everyone so that they can validate the signatures.

Public keys are generally distributed in a so-called certificate file, which contains the signer's public key and his name and contact details. In order to avoid forged certificates this information package is again signed by a trusted third party which issues a certificate to a person or other entity, such as an enterprise or a server. Such trusted third parties are called Certificate Authority (CA) or Trust Center (TC). The CA's own certificate is called the root certificate. It is usually published on the CA's web site for everyone to download it. It is required to validate certificates issued by the CA via its fingerprint (see below).

Certificates are generally stored in the X.509 format. It is important to distinguish certificates from a package containing both the certificate and the corresponding private key, which is called a digital ID. While certificates can freely be distributed to everyone, digital IDs must be carefully protected. Accessing the private key in a digital ID (in order to apply a digital signature) usually requires a password or passphrase. Common storage formats for digital IDs are PKCS#12 and PFX. Note that certificates and digital IDs are not always clearly distinguished; some people will talk about *signing a document with a certificate* when they actually mean *signing with a digital ID*.

Every certificate has an associated hash value (also called fingerprint) which can be used to double-check whether the certificate is genuine. In order to check a CA certificate manually you must read its fingerprint using suitable software, and compare it to the CA's fingerprint obtained via some other (trustworthy) means. Certificates are valid for a certain period of time. They are no longer valid as soon as their expiration date has passed, or if they have explicitly been revoked by the CA. Revoking a certificate may be necessary because the certificate holder has left the associated organization or the private key has been compromised.

Public Key Infrastructure (PKI) is a software environment which covers all relevant tasks for distributing and checking the validity of certificates. Certificate checking may involve online checks (using a protocol called Online Certificate Status Protocol, OCSP) or revocation lists. If neither CA nor PKI are available, self-signed certificates can be used. These require direct exchange of the certificate's fingerprint via a trusted transport, and are therefore generally only feasible for small user groups.

## 5.2 Obtaining and Managing Digital IDs

**Crypto engines for creating digital signatures.** PLOP DS supports various crypto engines. A crypto engine is a piece of software which implements various cryptographic functions that are required to generate digital signatures. The choice of a crypto engine affects the format and storage location of digital IDs, integration with other software and the operating system. PLOP DS supports the following crypto engines:

- ▶ The *builtin* engine is available on all platforms. It implements the required cryptographic functions directly in the PLOP DS kernel, without any external dependencies. This engine is active by default, but can also be selected explicitly with the suboption *engine=builtin* for the *sign* option of *PLOP\_create\_file()*.
- ▶ The *mscapi* engine refers to the Microsoft Cryptographic API (available only on Windows), which is an integrated part of the operating system. It allows PLOP DS to interoperate with the cryptographic infrastructure provided by Windows as well as third-party software or hardware which is attached via a CAPI driver. The *mscapi* engine can be selected with the suboption *engine=mscapi* for the *sign* option of *PLOP\_create\_file()*.
- ▶ The *pkcs#11* engine refers to a software interface called PKCS#11 which provides unified access to cryptographic tokens, where token stands for a smartcard, USB stick or other cryptographic device. Tokens offer higher security than software certificates, and are often protected with a PIN. This engine is not available on all platforms. The *PKCS#11* engine can be selected with the suboption *engine=pkcs#11* for the *sign* option of *PLOP\_create\_file()*.
- ▶ PLOP DS users can hook up an *external crypto engine (CE)*. This can be used to implement specific requirements regarding certificates or signature generation. A description of the CE interface and the corresponding binaries are available on request. The binaries in the standard PLOP DS distribution do not support the CE interface.

**Supported formats for digital IDs.** PLOP DS requires a digital ID for signing PDF documents. A digital ID contains the signer's digital certificate plus the corresponding private key, and is usually protected by a password or similar means. PLOP DS supports the following kinds of digital IDs:

- ▶ On all platforms with *engine=builtin*: digital ID files in PKCS#12 format (usually *.p12*) or PFX format (usually *.pfx*)
- ▶ On Windows with *engine=mscapi*: digital IDs in the Windows certificate store.
- ▶ On all platforms with PKCS#11 support with *engine=pkcs#11*: digital IDs stored on a smartcard or other cryptographic token (device) attached to the computer.

**Sources of digital IDs.** There are various sources where you can obtain a digital ID. Many IDs are intended for signing e-mail; these e-mail IDs can also be used in PLOP DS for signing PDF documents. Your choice of source for a digital ID depends on the number of required IDs (e.g. one per employee or only one corporate ID) and the desired degree of control:

- ▶ Obtain a digital ID from one of the public certificate authorities which issue IDs freely or for a fee.
- ▶ Build your own private certificate authority so that you can create digital IDs yourself. There are various software packages available for building a CA. Examples include the free OpenSSL software (see [www.openssl.org](http://www.openssl.org)), the *keytool* application which

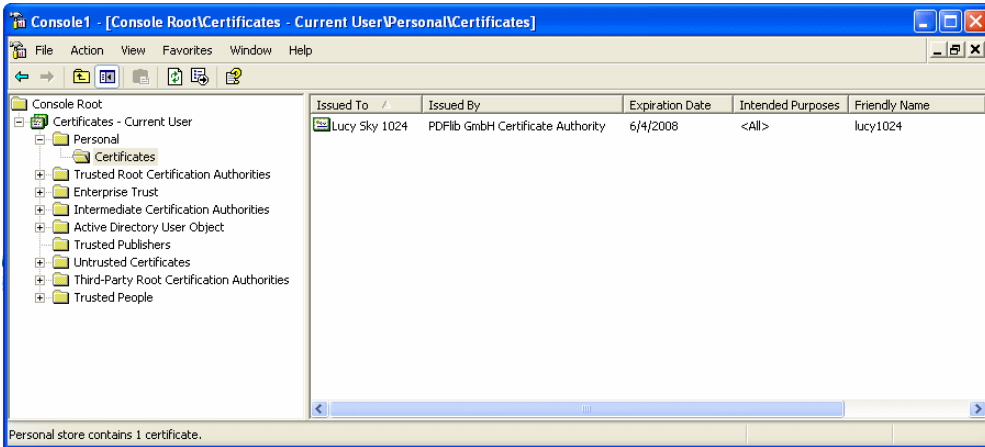


Fig. 5.1  
Managing the Windows certificate store with the management console MMC.

is part of Java, and the Certificate Services which are part of the Microsoft Windows Server operating system.

- ▶ Create a digital ID from a self-signed certificate. You can create self-signed certificates in Acrobat as follows: *Advanced, Security Settings, Digital IDs, Add ID, create a self-signed digital ID for use with Acrobat (Acrobat 8)* or *A new digital ID I want to create now (Acrobat 9)*. In the next step you can specify a PKCS#12 disk file or the Windows certificate store as target. Both methods will work in PLOP DS.

**Managing the Windows certificate store.** Windows 2000 and above can hold an arbitrary number of certificates which are organized in several certificate stores (and physically stored in the registry). To install a new certificate in the PFX or PKCS#12 format simply double-click on the certificate file and follow the certificate wizard.

You can view and organize certificates in Windows using the Microsoft Management Console (MMC) as follows:

- ▶ Click on *Start, Run...*, type *mmc*, and click OK. This will start the Management Console.
- ▶ In the *File* menu click *Add/Remove Snap-in...* and *Add...*
- ▶ In *Available Standalone Snap-ins* select *Certificates* and click *Add*.
- ▶ In the next dialog select *My user account* and *Finish*. Alternatively, use *Service account* or *Computer account* if this is the store where you keep your certificates.
- ▶ Click on *Close* and *OK*.

Now you can browse the installed certificates. Your own certificates will be available in the *Personal* category, which can be addressed in PLOP DS with the following option list (supplied to the *--signopt* command-line option or the *sign* option of *PLOP\_create\_file()*:

```
engine=mscapi digitalid={certstore={store=My subject={Demo PLOP User}}}
```

You can view certificate details by double-clicking on a certificate in MMC. In order to export a certificate in PFX format right-click on a certificate in the list and click *All Tasks, Export...*

Using the Management Console you can also import a certificate: right-click on a certificate store (e.g. *Personal*) and select *All Tasks, Import...*

## 5.3 Signing PDF Documents with PLOP DS

Signatures are implemented as form fields in PDF. PDF signatures always relate to the whole document (as opposed to particular pages), and are available in two flavors:

- ▶ Invisible signatures do not occupy any space on the page. They can be viewed in Acrobat by bringing up the *Signatures* tab (*View, Navigation Panels, Signatures...*).
- ▶ Visible signatures use a rectangular form field which is located somewhere on a page in the document. You can specify the page number, field name and field coordinates.

Additional properties can be specified for both types of signatures, e.g. location, reason for signing, and contact information.

In order to apply a digital signature with PLOP DS you need a digital ID (see Section 5.2, »Obtaining and Managing Digital IDs«, page 50). If you work with a digital ID file or token you will need the corresponding password. If you work with a personal (account-specific) digital ID in the Windows certificate store the ID will be protected by your Windows login.

**Applying signatures with PLOP DS.** The examples below show how to digitally sign PDF documents with the PLOP DS command-line tool and library. The option list supplied to the `--signopt` can be supplied to the PLOP DS API function `PLOP_create_file()` (option `sign`) in order to create a signature from within your own program. Full programming examples for all supported language bindings are contained in the PLOP DS package. The examples assume digital ID files `demo1024.p12` and `demo1024.pfx` with the password `demo`, and digital IDs in the Windows certificate store for the hypothetical user name `DEMO PLOP user 1024`. Sample digital ID files are also included in the distribution packages.

Create an invisible signature for a PDF document, using a digital ID from the file `demo1024.p12`. The password for the digital ID is contained in the file `pw.txt`:

```
plop --signopt "digitalid={filename=demo1024.p12} passwordfile=pw.txt" ←  
--outfile signed.pdf input.pdf  
plop -S "digitalid={filename=demo1024.p12} passwordfile=pw.txt" -o signed.pdf input.pdf
```

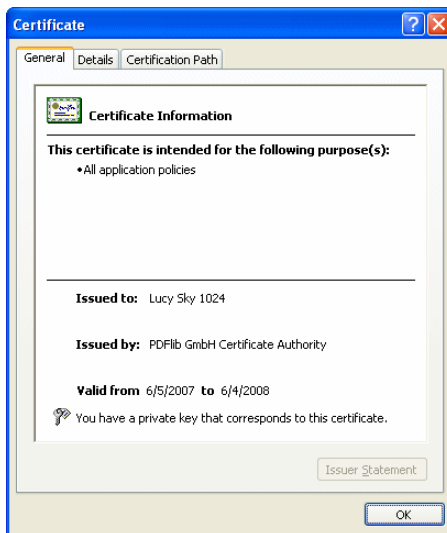


Fig. 5.2  
Certificate properties in Windows

Create a visible signature field in the lower left part of page 1. The password *demo* is directly supplied, which is not recommended for multi-user systems since the command-line with the password may be visible to other users (e.g. via the *ps* command on Unix systems):

```
plop --signopt "appearance={fieldname=Signature1 rect={10 10 200 100} } ␣  
    digitalid={filename=demo1024.p12} password={demo}" --outfile signed.pdf input.pdf  
plop -S "appearance={fieldname=Signature1 rect={10 10 200 100} } ␣  
    digitalid={filename=demo1024.p12} password={demo}" -o signed.pdf input.pdf
```

*Note* You must double-click and install the digital ID in the file *demo1024.pfx* in the Windows certificate store in order to run the Windows examples below.

(Windows only) Create an invisible signature for a PDF document, using a certificate from the Windows Certificate Store (from the default store *My*). This assumes that the digital ID is protected by your Windows login so that no password must be supplied:

```
plop --signopt "engine=miscapi digitalid={certstore={store=My subject={Demo PLOP User 1024}}}" ␣  
    --outfile signed.pdf input.pdf  
plop -S "engine=miscapi digitalid={certstore={store=My subject={Demo PLOP User 1024}}}" ␣  
    -o signed.pdf input.pdf
```

(Windows only) Create an invisible signature for a PDF document, using a certificate in the file *demo1024.pfx*:

```
plop --signopt "engine=miscapi digitalid={filename=demo1024.pfx} passwordfile=pw.txt" ␣  
    --outfile signed.pdf input.pdf  
plop --S "engine=miscapi digitalid={filename=demo1024.pfx} passwordfile=pw.txt" ␣  
    -o signed.pdf input.pdf
```

Create an invisible signature and encrypt the document with the master password *SECRET* for PDF encryption and password *demo* for accessing the digital ID:

```
plop --master SECRET --signopt "digitalid={filename=demo1024.p12} password={demo}" ␣  
    --outfile signed.pdf input.pdf  
plop --m SECRET --S "digitalid={filename=demo1024.p12} password={demo}" ␣  
    -o signed.pdf input.pdf
```

**Signing with smartcards and other cryptographic tokens.** Using the PKCS#11 engine in PLOP DS you can use certificates on a smartcard or other cryptographic token. This requires a DLL or shared library which implements a token-specific protocol. The PKCS#11 DLL must be provided by the token vendor as part of the corresponding driver kit. It must be installed on the system and must be available to PLOP DS. On Windows this means the DLL must either be copied to the Windows system directory, a directory which is included in the PATH environment variable, or the current directory. Note that a PKCS#11 DLLs may depend on other DLLs. In this case all required DLLs supplied by the token vendor must be made available to PLOP DS.

In the following examples we will refer to the vendor-specific PKCS#11 DLL as *cryptoki.dll*. The name of the actual DLL may be different.

Create an invisible signature for a PDF document, using a digital ID from a token addressed via PKCS#11. The PIN for the token is contained in the file *pw.txt*:

```
plop --signopt "engine=pkcs#11 digitalid={filename=cryptoki.dll} passwordfile=pw.txt" ␣  
    --outfile signed.pdf input.pdf
```

```
plop -S "engine=pkcs#11 digitalid={filename=cryptoki.dll} passwordfile=pw.txt" ←  
-o signed.pdf input.pdf
```

Create an invisible signature for a PDF document, using a digital ID from a token addressed via PKCS#11. No PIN is supplied in this command; instead, the PIN for the token must be typed into the token's integrated keyboard:

```
plop --signopt "engine=pkcs#11 digitalid={filename=cryptoki.dll}" ←  
--outfile signed.pdf input.pdf  
plop -S "engine=pkcs#11 digitalid={filename=cryptoki.dll}" -o signed.pdf input.pdf
```

Create a visible signature field in the lower left part of page 1. The PIN *1234* for the token is directly supplied, which is not recommended for multi-user systems since the command-line with the password may be visible to other users:

```
plop --signopt "appearance={fieldname=Signature1 rect={10 10 200 100} } ←  
engine=pkcs#11 digitalid={filename=cryptoki.dll} password={1234}" ←  
--outfile signed.pdf input.pdf  
plop -S "appearance={fieldname=Signature1 rect={10 10 200 100} } ←  
engine=pkcs#11 digitalid={filename=cryptoki.dll} password={1234}" ←  
-o signed.pdf input.pdf
```

**Unlocking digital IDs.** Digital IDs are generally protected with a password, passphrase, or PIN since they contain the confidential private key for creating the digital signature. In order to unlock a digital ID for use with PLOP DS you must provide proper authentication. If you supply the wrong password PLOP DS will throw an exception. The details of unlocking the digital ID depend on the selected crypto engine:

- ▶ With *engine=builtin*: You must supply the corresponding password with the *password* suboption of the *sign* option. If you are using the PLOP DS command-line tool it is strongly recommended to supply the password indirectly in an auxiliary file with the *passwordfile* suboption. If you supply the password directly instead of in a password file other users could possibly read it since the command-line may be visible to other users on a multi-user system.
- ▶ With *engine=mscapi*: Depending on your certificate settings the digital IDs in the Windows certificate store may be protected by your Windows login, and no additional password is required.
- ▶ With *engine=pkcs#11*: If the cryptographic token allows passwords/PINs to be submitted by software you must supply the *password* option as with *engine=builtin* (see above). If the token requires direct PIN or password entry (e.g. a smartcard reader with attached keyboard) you can omit the *password* option (or supply an empty string) and must manually type the PIN into the token's keyboard. Details of password/PIN handling may vary among cryptographic tokens.

**Cryptographic details of PLOP DS signatures.** PLOP DS creates signatures with the following characteristics:

The message digest (hash) algorithm for creating the signature is always SHA-1. PLOP DS never uses the MD5 algorithm which is not considered secure enough.

The encryption algorithm and key length for generating the signature are determined by the digital ID (they are specified when creating the public/private key pair for the ID). PLOP DS supports the following algorithms and key lengths:

- ▶ RSA with up to 4096 bit key length (requires Acrobat 6 or above for validation);
- ▶ DSA with up to 4096 bit key length (requires Acrobat 7 or above for validation).

**PLOP DS signature restrictions.** PLOP DS does not currently support the following signature-related features:

- ▶ Certified PDF: this is a special kind of signature where an author certifies the validity of a document which can later be modified by others.
- ▶ Incremental update and multiple signatures per document;
- ▶ Signature appearance: you cannot specify the text or images which are displayed in a visible signature field.
- ▶ Applying a digital signature can not be combined with linearization.






## 5.4 Validating Digital Signatures with Acrobat

PLOP DS creates standard PDF signatures according to Adobe's documented specification. Digital signatures created with PLOP DS do not require any third-party software for validation, but can be validated with Acrobat Standard/Professional 6 or above and Adobe Reader 6 or above. If the digital ID contains DSA keys instead of the more common RSA keys Acrobat 7 or above is required for validating the signature. Third-party validation software for PDF signatures which supports standard Acrobat signatures will also be able to validate signatures created with PLOP DS.

A signature is valid if all of the following conditions are true:

- ▶ The document has not been modified since the signature was applied.
- ▶ The certificate which has been used for signing is valid. A certificate is invalid if it is expired (according to the expiration date in the certificate) or has been revoked (this requires online testing or checking against a revocation list).
- ▶ The certificate belongs to a known person or entity, or has been issued by a well-known certificate authority (see below).

Proceed as follows to validate PDF signatures with Acrobat: open the *Signatures* tab (*View, Navigation Panels, Signatures...*) and right-click the signature, and select *Validate Signature*. In the resulting dialog box Acrobat will display a validity status icon for the signature along with additional information. Depending on the validation state, Acrobat 6/7/8 uses different icons for digital signatures (Acrobat 9 icons are slightly different):

- ▶ The signature icon  indicates the presence of a blank signature. Note that PLOP DS does not create blank signatures.
- ▶ A check mark  indicates that the signature is valid.
- ▶ A red X  indicates that the signature is invalid.
- ▶ A question mark  indicates that the signer's certificate is not contained in the list of trusted identities.
- ▶ A triangle  indicates that the document has been modified after the signature was applied. Since PLOP DS does not modify the document after signing it, any modifications must have been applied with other software (after signing the document).

Acrobat 6/7/8 will also display the signature status in the field area of a visible signature. However, Acrobat 9 no longer displays signature status icons in individual fields, but only in the Signatures panel (near the top of the Window) and the *Signatures* tab (at the left side of the window). Since this behavior may be confusing in mixed environments you can restore the previous behavior (i.e. display the validity status icon in each field) by setting the following registry key:

```
HKEY_CURRENT_USER\Software\Adobe\Adobe Acrobat\9.0\Security\cPubSec\iDisplayValidIcon
```

to the value 0 (digit zero). This will restore Acrobat 8 behavior in Acrobat 9. Acrobat security documentation provided by Adobe contains more details on registry settings for Acrobat.



**Establishing trust for signer certificates.** There are several ways how Acrobat can establish that the certificate used for signing a document should be considered trustworthy:

- ▶ Certificates issued by the Certificate Authority (CA) whose certificate is built into Acrobat are always accepted as trustworthy. However, the built-in CA does not issue software certificates which could be used in PLOP DS.
- ▶ Certificates issued by one of the CAs which are built into the Windows certificate store. This is the recommended method for end users outside of your enterprise; it is described in more detail below.
- ▶ The certificate is available in the Windows certificate store. This requires the user to manually add the signer's certificate to the Windows certificate store, which is generally only feasible in enterprise environments.
- ▶ You manually configure Acrobat to accept the individual signer's certificate, or to accept all certificates signed by a certain CA. This is recommended for enterprise environments. The required steps are described below.

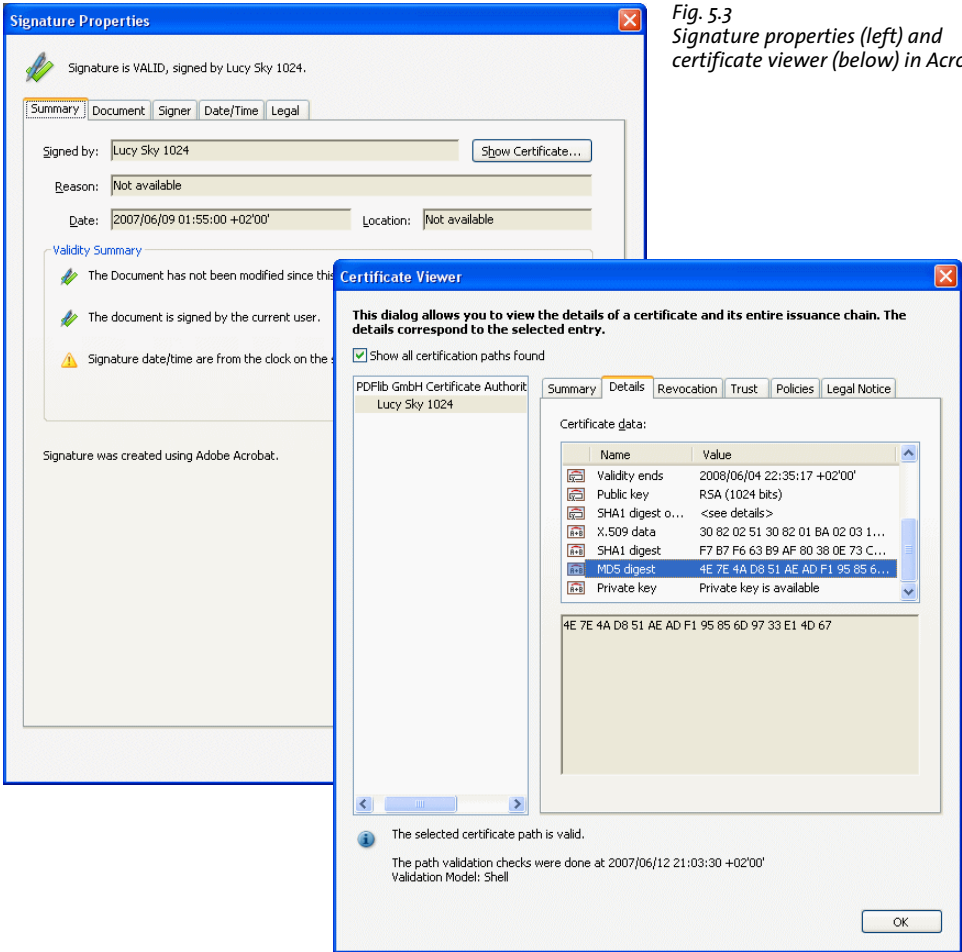


Fig. 5-3  
Signature properties (left) and  
certificate viewer (below) in Acrobat

**Allowing Acrobat to access the Windows certificate store.** If Acrobat is configured to access the Windows certificate store it can validate all signatures created with a CA which is contained in the certificate store. This could be one of the CAs which are already built into Windows (recommended for users outside of controlled enterprise environments) or a custom CA (suitable if you run your own enterprise CA and can configure it in the certificate store of all users). In the management console you can locate the certificates for these well-known CAs in the *Trusted Root Certification Authorities* store or in *Third-Party Root Certification Authorities*.

In order to take advantage of the CAs built into Windows proceed as follows: Obtain a digital ID from one of the commercial CAs which are built into Windows. You can review the list of built-in CAs as follows: start the Management Console with the Signature snap-in (see »Managing the Windows certificate store«, page 51), and navigate to *Trusted Root Certification Authorities/Certificates*. Now you can see a list with dozens of commercial CAs. Select a certificate from the list, and double-click on its name. In the *Certificates* dialog go to *Details*, scroll to the Subject entry, and double-click on it. This should display enough information for contacting the CA, e.g. an e-mail address. Obtain a digital ID from one of these CAs and use it for signing PDF documents with PLOP DS.

In order to make a custom CA known to the Windows store obtain its certificate, double-click on it, and import it into the Windows certificate store.

In both cases above you must make sure that Acrobat will accept all certificates issued by the selected CA (or any CA in the Windows certificate store). In Acrobat 7/8/9 proceed as follows (see Figure 5.4): *Edit, Preferences, [General...], Security, Advanced Preferences..., Windows Integration* (Acrobat 6: *Edit, Preferences, [General...], Digital Signatures, Advanced Preferences...*). In the resulting dialog, below the text *Trust ALL root certificates in the Windows Certificate Store for the following operations*: check the box labelled *Validating Signatures*.

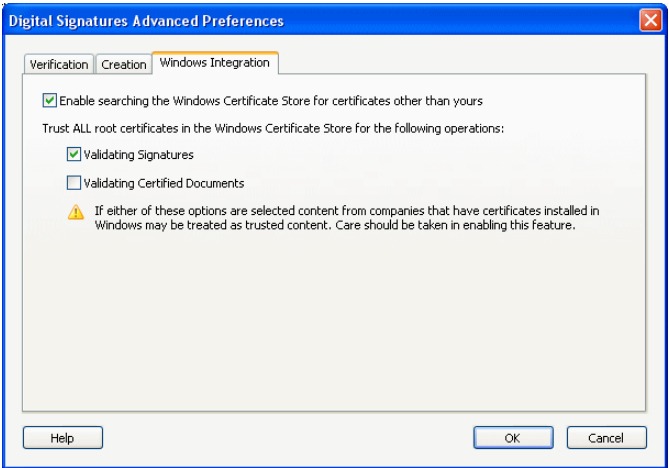


Fig. 5.4  
Configure Acrobat to access the  
Windows Certificate Store

**Accepting an individual certificate.** You can add individual certificates to the list of trusted identities using any of the following methods (refer to the Acrobat help for details):

- ▶ Add the certificate to the Windows certificate store: double-click on the certificate file; this will bring up a wizard for installing the certificate. In addition, you must configure Acrobat to access the Windows certificate store (see above).
- ▶ Import a certificate from a disk file or a directory server: In Acrobat 8/9 this can be achieved with the following steps: *Advanced, Manage Trusted Identities..., Add Contacts, Browse...* Select the certificate, click *Edit Trust...*, go to the *Trust* tab and in the *Trust this certificate for:* checkbox select *Signatures and as a Trusted Root*.
- ▶ Import a certificate from a signed PDF: In Acrobat 8/9 this can be achieved with the following steps: open a signed PDF, open the *Signatures* tab (*View, Navigation Panels, Signatures*), right-click on the signature and select *[Show Signature] Properties*. This will bring up the *Signature Properties* dialog. In the *Summary* tab click on *Show Certificate...*, go to the *Details* tab and check the certificate's fingerprint (MD5 or SHA-1 digest). If the fingerprint matches the fingerprint provided to you by the signer separately by trusted means, go to the *Trust* tab, click on *Add to Trusted Identities...*, and finally check *Signatures and as a trusted root* (Acrobat 8) or *Use this certificate as a trusted root* (Acrobat 9).

**Accepting all certificates issued by a CA.** This method is recommended if you work with documents signed by many different individuals which all use certificates issued by the same CA (typically an enterprise CA). Obtain the CA's certificate and import it into Acrobat as detailed above for individual certificates, or into Windows if you allow Acrobat to access the Windows certificate store. As a result, Acrobat will accept all signatures created with certificates issued by this CA.



# 6 The pCOS Interface

The pCOS (*PDFlib Comprehensive Object Syntax*) interface provides a simple and elegant facility for retrieving arbitrary information from all sections of a PDF document which do not describe page contents, such as page dimensions, metadata, interactive elements, etc. pCOS users are assumed to have some basic knowledge of internal PDF structures and dictionary keys, but do not have to deal with PDF syntax and parsing details.

We strongly recommend that pCOS users obtain a copy of the *PDF Reference*, which is available as follows:

Adobe Systems Incorporated: PDF Reference, Sixth Edition: Version 1.7. Downloadable PDF from [www.adobe.com/devnet/pdf/pdf\\_reference.html](http://www.adobe.com/devnet/pdf/pdf_reference.html)

## 6.1 Simple pCOS Examples

*Cookbook* A collection of pCOS coding fragments for solving specific problems can be found in the pCOS Cookbook.

Assuming a valid PDF document handle is available, the pCOS functions *PLOP\_pcos\_get\_number()*, *PLOP\_pcos\_get\_string()*, and *PLOP\_pcos\_get\_stream()* can be used to retrieve information from a PDF using the pCOS path syntax. Table 6.1 lists some common pCOS paths and their meaning.

Table 6.1 pCOS paths for commonly used PDF objects

pCOS path	type	explanation
length:pages	number	number of pages in the document
/Info/Title	string	document info field Title
/Root/Metadata	stream	XMP stream with the document's metadata
fonts[...]/name	string	name of a font; the number of entries can be retrieved with length:fonts
fonts[...]/vertical	boolean	check a font for vertical writing mode
fonts[...]/embedded	boolean	embedding status of a font
pages[...]/width	number	width of the visible area of the page

**Number of pages.** The total number of pages in a document can be queried as follows:

```
pagecount = p.pcos_get_number(doc, "length:pages");
```

**Document info fields.** Document information fields can be retrieved with the following code sequence:

```
objtype = p.pcos_get_string(doc, "type:/Info/Title");
if (objtype.equals("string"))
{
    /* Document info key found */
    title = p.pcos_get_string(doc, "/Info/Title");
}
```

**Page size.** Although the *MediaBox*, *CropBox*, and *Rotate* entries of a page can directly be obtained via pCOS, they must be evaluated in combination in order to find the actual size of a page. Determining the page size is much easier with the *width* and *height* keys of the *pages* pseudo object. The following code retrieves the width and height of page 3 (note that indices for the *pages* pseudo object start at 0):

```
pagenum = 2
width = p.pcos_get_number(doc, "pages[" + pagenum + "]/width");
height = p.pcos_get_number(doc, "pages[" + pagenum + "]/height");
```

**Listing all fonts in a document.** The following sequence creates a list of all fonts in a document along with their embedding status:

```
fontcount = p.pcos_get_number(doc, "length:fonts");

for (i=0; i < fontcount; i++)
{
    fontname = p.pcos_get_string(doc, "fonts[" + i + "]/name");
    embedded = p.pcos_get_number(doc, "fonts[" + i + "]/embedded");
}
```

**Encryption status.** You can query the *pcosmode* pseudo object to determine the pCOS mode for the document:

```
if (p.pcos_get_number(doc, "pcosmode") == 2)
{
    /* full pCOS mode */
}
```

**XMP meta data.** A stream containing XMP meta data can be retrieved with the following code sequence:

```
objtype = p.pcos_get_string(doc, "type:/Root/Metadata");
if (objtype.equals("stream"))
{
    /* XMP meta data found */
    metadata = p.pcos_get_stream(doc, "", "/Root/Metadata");
}
```

## 6.2 Handling Basic PDF Data Types

pCOS offers the three functions `PLOP_pcos_get_number()`, `PLOP_pcos_get_string()`, and `PLOP_pcos_get_stream()`. These can be used to retrieve all basic data types which may appear in PDF documents.

**Numbers.** Objects of type *integer* and *real* can be queried with `PLOP_pcos_get_number()`. pCOS doesn't make any distinction between integer and floating point numbers.

**Names and strings.** Objects of type *name* and *string* can be queried with `PLOP_pcos_get_string()`. Name objects in PDF may contain non-ASCII characters and the # syntax (decoration) to include certain special characters. pCOS deals with PDF names as follows:

- ▶ Name objects will be undecorated (i.e. the # syntax will be resolved) before they are returned.
- ▶ Name objects will be returned as Unicode strings in most language bindings. However, in the C and C++ language bindings they will be returned as UTF-8 without BOM.

Since the majority of strings in PDF are text strings `PLOP_pcos_get_string()` will treat them as such. However, in rare situations strings in PDF are used to carry binary information. In this case strings should be retrieved with the function `PLOP_pcos_get_stream()` which preserves binary strings and does not modify the contents in any way.

**Booleans.** Objects of type *boolean* can be queried with `PLOP_pcos_get_number()` and will be returned as 1 (true) or 0 (false). `PLOP_pcos_get_string()` can also be used to query boolean objects; in this case they will be returned as one of the strings *true* and *false*.

**Streams.** Objects of type *stream* can be queried with `PLOP_pcos_get_stream()`. Depending on the pCOS data type (*stream* or *fstream*) the contents will be compressed or uncompressed. Using the *keepfilter* option of `PLOP_pcos_get_stream()` the client can retrieve compressed data even for type *stream*.

Stream data in PDF may be preprocessed with one or more filters. The list of filters present at the stream can be queried from the stream dictionary; for images this information is much easier accessible in the image's *filterinfo* dictionary. If a stream's filter chain contains only supported filters its type will be *stream*. When retrieving the contents of a *stream* object, `PLOP_pcos_get_stream()` will remove all filters and return the resulting unfiltered data.

*Note* pCOS does not support the following stream filters: JBIG2 and JPX.

If there is at least one unsupported filter in a stream's filter chain, the object type will be reported as *fstream* (filtered stream). When retrieving the contents of an *fstream* object, `PLOP_pcos_get_stream()` will remove the supported filters at the beginning of a filter chain, but will keep the remaining unsupported filters and return the stream data with the remaining unsupported filters still applied. The list of applied filters can be queried from the stream dictionary, and the filtered stream contents can be retrieved with `PLOP_pcos_get_stream()`. Note that the names of supported filters will not be removed when querying the names of the stream's filters, so the client should ignore the names of supported filters.

Streams in PDF generally contain binary data. However, in rare cases (text streams) they may contain textual data instead (e.g. JavaScript streams). In order to trigger the appropriate text conversion, use the the *convert=unicode* option in *PLOP\_pcos\_get\_stream()*.



## 6.3 Composite Data Structures and IDs

Objects with one of the basic data types can be arranged in two kinds of composite data structures: arrays and dictionaries. pCOS does not offer specific functions for retrieving composite objects. Instead, the objects which are contained in a dictionary or array can be addressed and retrieved individually.

**Arrays.** Arrays are one-dimensional collections of any number of objects, where each object may have arbitrary type.

The contents of an array can be enumerated by querying the number *N* of elements it contains (using the *length* prefix in front of the array's path, see Table 6.2), and then iterating over all elements from index 0 to *N*-1.

**Dictionaries.** Dictionaries (also called associative arrays) contain an arbitrary number of object pairs. The first object in each pair has the type *name* and is called the key. The second object is called the value, and may have an arbitrary type except *null*.

The contents of a dictionary can be enumerated by querying the number *N* of elements it contains (using the *length* prefix in front of the dictionary's path, see Table 6.2), and then iterating over all elements from index 0 to *N*-1. Enumerating dictionaries will provide all dictionary keys in the order in which they are stored in the PDF using the *.key* suffix at the end of the dictionary's path. Similarly, the corresponding values can be enumerated with the *.val* suffix. Inherited values (see below) and pseudo objects will not be visible when enumerating dictionary keys, and will not be included in the *length* count.

Some page-related dictionary entries in PDF can be inherited across a tree-like data structure, which makes it difficult to retrieve them. For example the *MediaBox* for a page is not guaranteed to be contained in the page dictionary, but may be inherited from an arbitrarily complex page tree. pCOS eliminates this problem by transparently inserting all inherited keys and values into the final dictionary. In other words, pCOS users can assume that all inheritable entries are available directly in a dictionary, and don't have to search all relevant parent entries in the tree. This merging of inherited entries is only available when accessing the pages tree via the *pages[ ]* pseudo object; accessing the */Pages* tree, the *objects[ ]* pseudo object, or enumerating the keys via *pages[ ][ ]* will return the actual entries which are present in the respective dictionary, without any inheritance applied.

**pCOS IDs for dictionaries and arrays.** Unlike PDF object IDs, pCOS IDs are guaranteed to provide a unique identifier for an element addressed via a pCOS path (since arrays and dictionaries can be nested an object can have the same PDF object ID as its parent array or dictionary). pCOS IDs can be retrieved with the *pcosid* prefix in front of the dictionary's or array's path (see Table 6.2).

The pCOS ID can therefore be used as a shortcut for repeatedly accessing elements without the need for explicit path addressing. For example, this will improve performance when looping over all elements of a large array. Use the *objects[ ]* pseudo object to retrieve the contents of an element identified by a particular ID.

## 6.4 Path Syntax

The backbone of the pCOS interface is a simple path syntax for addressing and retrieving any object contained in a PDF document. In addition to the object data itself pCOS can retrieve information about an object, e.g. its type or length. Depending on the object's type (which itself can be queried) one of the functions *PLOP\_pcos\_get\_number()*, *PLOP\_pcos\_get\_string()*, and *PLOP\_pcos\_get\_stream()* can be used to obtain the value of an object. The general syntax for pCOS paths is as follows:

```
[<prefix>:][pseudoname[<index>]]/<name>[<index>]/<name>[<index>] ... [.key|.val]
```

The meaning of the various path components is as follows:

- ▶ The optional *prefix* can attain the values listed in Table 6.2.
- ▶ The optional *pseudo object name* may contain one of the values described in Section 6.5, »Pseudo Objects«, page 68.
- ▶ The *name* components are dictionary keys found in the document. Multiple names are separated with a / character. An empty path, i.e. a single / denotes the document's Trailer dictionary. Each name must be a dictionary key present in the preceding dictionary. Full paths describe the chain of dictionary keys from the initial dictionary (which may be the Trailer or a pseudo object) to the target object.
- ▶ Paths or path components specifying an array or dictionary can have a numerical index which must be specified in decimal format between brackets. Nested arrays or dictionaries can be addressed with multiple index entries. The first entry in an array or dictionary has index 0.
- ▶ Paths or path components specifying a dictionary can have an index qualifier plus one of the suffixes *.key* or *.val*. This can be used to retrieve a particular dictionary key or the corresponding value of the indexed dictionary entry, respectively. If a path for a dictionary has an index qualifier it must be followed by one of these suffixes.

**Encoding for pCOS paths.** In most cases pCOS paths will contain only plain ASCII characters. However, in a few cases (e.g. PDFlib Block names) non-ASCII characters may be required. pCOS paths must be encoded according to the following rules:

- ▶ When a path component contains any of the characters /, [, ], or #, these must be expressed by a number sign # followed by a two-digit hexadecimal number.
- ▶ In Unicode-aware language bindings the path consists of a regular Unicode string which may contain ASCII and non-ASCII characters.
- ▶ In non-Unicode-aware language bindings the path must be supplied in UTF-8. The string may or may not contain a BOM, but this doesn't make any difference. A BOM may be placed at the start of the path, or at the start of individual path components (i.e. after a slash character).

On EBCDIC systems the path must generally be supplied in *ebcdic* encoding. Characters outside the ASCII character set must be supplied as EBCDIC-UTF-8 (with or without BOM).

**Path prefixes.** Prefixes can be used to query various attributes of an object (as opposed to its actual value). Table 6.2 lists all supported prefixes.

The *length* prefix and content enumeration via indices are only applicable to plain PDF objects and pseudo objects of type *array*, but not any other pseudo objects. The *pcosid* prefix cannot be applied to pseudo objects. The *type* prefix is supported for all pseudo objects.

Table 6.2 pCOS path prefixes

prefix	explanation
length	(Number) Length of an object, which depends on the object's type:
	<b>array</b> Number of elements in the array
	<b>dict</b> Number of key/value pairs in the dictionary
	<b>stream</b> Number of key/value pairs in the stream dict (not the stream length; use the Length key to determine the length of stream data in bytes)
	<b>fstream</b> Same as stream
	<b>other</b> 0
pcosid	(Number) Unique pCOS ID for an object of type dictionary or array. If the path describes an object which doesn't exist in the PDF the result will be -1. This can be used to check for the existence of an object, and at the same time obtaining an ID if it exists.
type	(String or number) Type of the object as number or string:
	<b>0, null</b> Null object or object not present (use to check existence of an object)
	<b>1, boolean</b> Boolean object
	<b>2, number</b> Integer or real number
	<b>3, name</b> Name object
	<b>4, string</b> String object
	<b>5, array</b> Array object
	<b>6, dict</b> Dictionary object (but not stream)
	<b>7, stream</b> Stream object which uses only supported filters
	<b>8, fstream</b> Stream object which uses one or more unsupported filters
	Enums for these types are available for the convenience of C and C++ developers.

# 6.5 Pseudo Objects

Pseudo objects extend the set of pCOS paths by introducing some useful elements which can be used as an abbreviation for information which is present in the PDF, but cannot easily be accessed by reading a single value. The following sections list all supported pseudo objects. Pseudo objects of type *dict* can not be enumerated.

**Universal pseudo objects.** Universal pseudo objects are always available, regardless of encryption and passwords. This assumes that a valid document handle is available, which may require setting the option *requiredmode* suitably when opening the document. Table 6.3 lists all universal pseudo objects.

Table 6.3 Universal pseudo objects

object name	explanation
<b>encrypt</b>	(Dict) Dictionary with keys describing the encryption status of the document:
<b>length</b>	(Number) Length of the encryption key in bits
<b>algorithm</b>	(Number)
<b>description</b>	(String) Encryption algorithm number or description:
	-1 Unknown encryption
	0 No encryption
	1 40-bit RC4 (Acrobat 2-4)
	2 128-bit RC4 (Acrobat 5)
	3 128-bit RC4 (Acrobat 6)
	4 128-bit AES (Acrobat 7)
	5 Public key on top of 128-bit RC4 (Acrobat 5) (unsupported)
	6 Public key on top of 128-bit AES (Acrobat 7) (unsupported)
	7 Adobe Policy Server (Acrobat 7) (unsupported)
	8 Adobe Digital Editions (EBX) (unsupported)
<b>master</b>	(Boolean) True if the PDF requires a master password to change security settings (permissions, user or master password), false otherwise
<b>user</b>	(Boolean) True if the PDF requires a user password for opening, false otherwise
<b>noaccessible, noannots, noassemble, nocopy, noforms, nohiresprint, nomodify, noprint</b>	(Boolean) True if the respective access protection is set, false otherwise
<b>plainmetadata</b>	(Boolean) True if the PDF contains unencrypted meta data, false otherwise
<b>extension-level</b>	(String) Adobe Extension Level based on ISO 32000, or 0 if no extension level is present. Acrobat 9 creates documents with extension level 3.
<b>filename</b>	(String) Name of the PDF file.
<b>filesize</b>	(Number) Size of the PDF file in bytes
<b>fullpdf-version</b>	(Number) Numerical value for the PDF version number. The numbers increase monotonically for each PDF/Acrobat version. The value 100 * BaseVersion + ExtensionLevel will be returned, e.g.
	150 PDF 1.5 (Acrobat 6)
	160 PDF 1.6 (Acrobat 7)
	170 PDF 1.7 (Acrobat 8)
	173 PDF 1.7 Adobe Extension Level 3 (Acrobat 9)
<b>linearized</b>	(Boolean) True if the PDF document is linearized, false otherwise
<b>major</b>	(Number) Major, minor, or revision number of the library, respectively.
<b>minor</b>	
<b>revision</b>	

Table 6.3 Universal pseudo objects

object name	explanation
pcosinterface	(Number) Interface number of the underlying pCOS implementation. This specification describes interface number 3. The following table details which product versions implement various pCOS interface numbers:
	1 TET 2.0, 2.1
	2 pCOS 1.0
	3 PDFlib+PDI 7, PPS 7, TET 2.2, pCOS 2.0, PLOP 3.0, TET 2.3
	4 PLOP 4.0, TET 3.0
pcosmode	(Number/string) pCOS mode as number or string:
pcos-modename	0 minimum
	1 restricted
	2 full
pdfversion	(Number) PDF version number multiplied by 10, e.g. 16 for PDF 1.6
pdfversion-string	(String) Full PDF version string in the form expected by various API functions for setting the PDF output compatibility, e.g. 1.5, 1.6, 1.7, 1.7ext3
version	(String) Full library version string in the format <major>.<minor>.<revision>, possibly suffixed with additional qualifiers such as beta, rc, etc.

**Pseudo objects for PDF objects, pages, and interactive elements.** Table 6.4 lists pseudo objects which can be used for retrieving object or page information, or serve as short-cuts for various interactive elements.

Table 6.4 Pseudo objects for PDF objects, pages, and interactive elements

object name	explanation
<b>articles</b>	(Array of dicts) Array containing the article thread dictionaries for the document. The array will have length 0 if the document does not contain any article threads. In addition to the standard PDF keys pCOS supports the following pseudo key for dictionaries in the articles array: <b>beads</b> (Array of dicts) Bead directory with the standard PDF keys, plus the following: <b>destpage</b> (Number) Number of the target page (first page is 1)
<b>bookmarks</b>	(Array of dicts) Array containing the bookmark (outlines) dictionaries for the document. In addition to the standard PDF keys pCOS supports the following pseudo keys for dictionaries in the bookmarks array: <b>level</b> (Number) Indentation level in the bookmark hierarchy <b>destpage</b> (Number) Number of the target page (first page is 1) if the bookmark points to a page in the same document, -1 otherwise.
<b>fields</b>	(Array of dicts) Array containing the form fields dictionaries for the document. In addition to the standard PDF keys in the field dictionary and the entries in the associated Widget annotation dictionary pCOS supports the following pseudo keys for dictionaries in the fields array: <b>level</b> (Number) Level in the field hierarchy (determined by « as separator) <b>fullname</b> (String) Complete name of the form field. The same naming conventions as in Acrobat 7 will be applied.
<b>names</b>	(Dict) A dictionary where each entry provides simple access to a name tree. The following name trees are supported: AP, AlternatePresentations, Dests, EmbeddedFiles, IDS, JavaScript, Pages, Renditions, Templates, URLs. Each name tree can be accessed by using the name as a key to retrieve the corresponding value, e.g.: names/Dests[0].key retrieves the name of a destination names/Dests[0].val retrieves the corresponding destination dictionary In addition to standard PDF dictionary entries the following pseudo keys for dictionaries in the Dests names tree are supported: <b>destpage</b> (number) Number of the target page (first page is 1) if the destination points to a page in the same document, -1 otherwise. In order to retrieve other name tree entries these must be queried directly via /Root/Names/Dests etc. since they are not present in the name tree pseudo objects.
<b>objects</b>	(Array) Address an element for which a pCOS ID has been retrieved earlier using the pcoid prefix. The ID must be supplied as array index in decimal form; as a result, the PDF object with the supplied ID will be addressed. The length prefix cannot be used with this array.

Table 6.4 Pseudo objects for PDF objects, pages, and interactive elements

object name	explanation
<b>pages</b>	<p>(Array of dicts) Each array element addresses a page of the document. Indexing it with the decimal representation of the page number minus one addresses that page (the first page has index 0). Using the length prefix the number of pages in the document can be determined. A page object addressed this way will incorporate all attributes which are inherited via the /Pages tree. The entries /MediaBox and /Rotate are guaranteed to be present. In addition to standard PDF dictionary entries the following pseudo entries are available for each page:</p> <p><b>colorspaces, extgstates, fonts, images, patterns, properties, shadings, templates</b> (Arrays of dicts) Page resources according to Table 6.5.</p> <p><b>annots</b> (Array of dicts) In addition to the standard PDF keys in the Annots array pCOS supports the following pseudo key for dictionaries in the annots array:</p> <p><b>destpage</b> (Number; only for Subtype=Link and if a Dest entry is present) Number of the target page (first page is 1)</p> <p><b>blocks</b> (Array of dicts) Shorthand for pages[ ]/PieceInfo/PDFlib/Private/Blocks[ ], i.e. the page's block dictionary. In addition to the existing PDF keys pCOS supports the following pseudo key for dictionaries in the blocks array:</p> <p><b>rect</b> (Rectangle) Similar to Rect, except that it takes into account any relevant CropBox/MediaBox and Rotate entries and normalizes coordinate ordering.</p> <p><b>height</b> (Number) Height of the page. The MediaBox or the CropBox (if present) will be used to determine the height. Rotate entries will also be applied.</p> <p><b>isempty</b> (Boolean) True if the page is empty, and false if the page is not empty</p> <p><b>label</b> (String) The page label of the page (including any prefix which may be present). Labels will be displayed as in Acrobat. If no label is present (or the PageLabel dictionary is malformed), the string will contain the decimal page number. Roman numbers will be created in Acrobat's style (e.g. VI), not in classical style which is different (e.g. XLV). If /Root/PageLabels doesn't exist, the document doesn't contain any page labels.</p> <p><b>width</b> (Number) Width of the page (same rules as for height)</p> <p>The following entries will be inherited: CropBox, MediaBox, Resources, Rotate.</p>
<b>pdfa</b>	(String) PDF/A conformance level of the document (e.g. PDF/A-1a:2005) or none
<b>pdfx</b>	(String) PDF/X conformance level of the document (e.g. PDF/X-1a:2001) or none
<b>tagged</b>	(Boolean) True if the PDF document is tagged, false otherwise

**Pseudo objects for simplified resource handling.** Resources are a key concept for managing various kinds of data which are required for completely describing the contents of a page. The resource concept in PDF is very powerful and efficient, but complicates access with various technical concepts, such as recursion and resource inheritance. pCOS greatly simplifies resource retrieval and supplies several groups of pseudo objects which can be used to directly query resources. Some of these pseudo resource dictionaries contain entries in addition to the standard PDF keys in order to further simplify resource information retrieval. pCOS pseudo resources reflect resources from the user's point of view, and differ from native PDF resources:

- ▶ Some entries may have been added (e.g. inline images, simple color spaces) or deleted (e.g. listed fonts which are not used on any page).
- ▶ In addition to the original PDF dictionary keys resource dictionaries may contain some user-friendly keys for auxiliary information (e.g. embedding status of a font, number of components of a color space).

pCOS supports two groups of pseudo objects for resource retrieval. Global resource arrays contain all resources of a given type in a PDF document, while page-based resources contain only the resources used by a particular page. The corresponding pseudo arrays are available for all resource types listed in Table 6.5:

- ▶ A list of all resources in the document is available in the global resource array (e.g. *images[ ]*). Retrieving the *length* of one of the global resource pseudo arrays results in a resource scan (see below) for all pages
- ▶ A list of resources on each page is available in the page-based resource array (e.g. *pages[ ]/images[ ]*). Accessing the *length* of one of a page's resource pseudo arrays results in a resource scan for that page (to collect all resources which are actually used on the page, and to merge images on that page).

A *resource scan* is a full scan of the page including Unicode mapping and image merging, but excluding Wordfinder operation. Applications which require a full resource listing and all page contents are recommended to process all pages before querying resource pseudo objects in order to avoid the resource scans in addition to the regular page scans.



Table 6.5 Pseudo objects for resources; each resource category P creates two resource arrays P[ ] and pages[ ]/P[ ].

object name	explanation
<b>colorspaces</b>	(Array of dicts) Array containing dictionaries for all color spaces on the page or in the document. In addition to the standard PDF keys in color space and ICC profile stream dictionaries the following pseudo keys are supported: <b>alternateid</b> (Integer; only for name=Separation and DeviceN) Index of the underlying alternate color space in the colorspaces[ ] pseudo object. <b>baseid</b> (Integer; only for name=Indexed) Index of the underlying base color space in the colorspaces[ ] pseudo object. <b>colorantname</b> (Name; only for name=Separation) Name of the colorant. Non-ASCII CJK color names will be converted to Unicode. <b>colorantnames</b> (Array of names; only for name=DeviceN) Names of the colorants <b>components</b> (Integer) Number of components of the color space <b>name</b> (String) Name of the color space: CalGray, CalRGB, DeviceCMYK, DeviceGray, DeviceN, DeviceRGB, ICCBased, Indexed, Lab, Separation <b>csarray</b> (Array; not for name=DeviceGray/RGB/CMYK) Array describing the underlying native color space, i.e. the original color space object in the PDF. Color space resources will include all color spaces which are referenced from any type of object, including the color spaces which do not require native PDF resources (i.e. DeviceGray, DeviceRGB, and DeviceCMYK).
<b>extgstates</b>	(Array of dicts) Array containing the dictionaries for all extended graphics states (ExtGStates) on the page or in the document
<b>fonts</b>	(Array of dicts) Array containing dictionaries for all fonts on the page or in the document. In addition to the standard PDF keys in font dictionaries, the following pseudo keys are supported: <b>name</b> (String) PDF name of the font without any subset prefix. Non-ASCII CJK font names will be converted to Unicode. <b>embedded</b> (Boolean) Embedding status of the font <b>type</b> (String) Font type: (unknown), Composite, Multiple Master, OpenType, TrueType, TrueType (CID), Type 1, Type 1 (CID), Type 1 CFF, Type 1 CFF (CID), Type 3 <b>vertical</b> (Boolean) true for fonts with vertical writing mode, false otherwise

Table 6.5 Pseudo objects for resources; each resource category *P* creates two resource arrays *P*[ ] and pages[ ]/*P*[ ].

object name	explanation
<b>images</b>	<p>(Array of dicts) Array containing dictionaries for all images on the page or in the document. The TET product will add merged (artificial) images to the images[ ] array.</p> <p>In addition to the standard PDF keys the following pseudo keys are supported:</p> <p><b>bpc</b> (Integer) The number of bits per component. This entry is usually the same as BitsPerComponent, but unlike this it is guaranteed to be available. For JPEG2000 images it may be -1 since the number of bits per component may not be available in the PDF structures.</p> <p><b>colorspaceid</b> (Integer) Index of the image's color space in the colorspaces[ ] pseudo object. This can be used to retrieve detailed color space properties. For JPEG2000 images the id may be -1 since the color space may not be encoded in the PDF structures.</p> <p><b>filterinfo</b> (Dict) Describes the remaining filter for streams with unsupported filters or when retrieving stream data with the keepfilter option set to true. If there is no such filter no filterinfo dictionary will be available. The dictionary contains the following entries:</p> <p><b>name</b> (Name) Name of the filter</p> <p><b>supported</b> (Boolean) True if the filter is supported</p> <p><b>decodeparms</b> (Dict) The DecodeParms dictionary if one is present for the filter</p> <p><b>maskid</b> (Integer) Index of the image's mask in the images[ ] pseudo object if the image has an explicit pixel-based mask, otherwise -1</p> <p><b>mergetype</b><sup>1</sup> The following types describe the status of the image:</p> <p><b>0</b> (normal) The image corresponds to an image in the PDF.</p> <p><b>1</b> (artificial) The image is the result of merging multiple consumed images (i.e. images with mergetype=2) into a single image. The resulting artificial image does not exist in the PDF data structures as an object.</p> <p><b>2</b> (consumed) The image should be ignored since it has been merged into a larger image. Although the image exists in the PDF, it usually should not be extracted because it is part of an artificial image (i.e. an image with mergetype=1).</p> <p><b>pdftype</b> (Integer) Type of the image. The following types describe a real image which corresponds to a native PDF image:</p> <p><b>0</b> Normal PDF image XObject</p> <p><b>1</b> Inline image</p> <p><b>2</b> (Only for artificial images) Mixed, i.e. result of merging normal and inline images</p>
<b>patterns</b>	(Array of dicts) Array containing dictionaries for all patterns on the page or in the document
<b>properties</b>	(Array of dicts) Array containing dictionaries for all properties on the page or in the document
<b>shadings</b>	<p>(Array of dicts) Array containing dictionaries for all shadings on the page or in the document. In addition to the standard PDF keys in shading dictionaries the following pseudo key is supported:</p> <p><b>colorspaceid</b> (Integer) Index of the underlying color space in the colorspaces[ ] pseudo object.</p>
<b>templates</b>	(Array of dicts) Array containing dictionaries for all templates (Form XObjects) on the page or in the document

1. This entry reflects information regarding all pages processed so far. It may change its value while processing other pages in the document. All pages in the document must have been processed if definite information (regarding the full document) is required.

# 6.6 Encrypted PDF Documents

pCOS supports encrypted and unencrypted PDF documents as input. However, full object retrieval for encrypted documents requires the appropriate master password to be supplied when opening the document. Depending on the availability of user and master password, encrypted documents can be processed in one of the pCOS modes described below.

**Full pCOS mode (mode 2).** Encrypted PDFs can be processed without any restriction provided the master password has been supplied upon opening the file. All objects will be returned unencrypted. Unencrypted documents will always be opened in full pCOS mode.

**Restricted pCOS mode (mode 1).** If the document has been opened without the appropriate master password and does not require a user password (or the user password has been supplied) pCOS operations are subject to the following restriction: The contents of objects with type *string*, *stream*, or *fstream* can not be retrieved with the following exceptions:

- ▶ The objects */Root/Metadata* and */Info/\** (document info keys) can be retrieved if *nocopy=false* or *plainmetadata=true*.
- ▶ The objects *bookmarks[...]/Title* and *pages[...]/annots/Contents* (bookmark and annotation contents) can be retrieved if *nocopy=false*, i.e. if text extraction is allowed for the main text on the pages.

**Minimum pCOS mode (mode 0).** Regardless of the encryption status and the availability of passwords, the universal pCOS pseudo objects listed in Table 6.3 are always available. For example, the *encrypt* pseudo object can be used to query a document's encryption status. Encrypted objects can not be retrieved in minimum pCOS mode.

Table 6.6 lists the resulting pCOS modes for various password combinations. Depending on the document's encryption status and the password supplied when opening the file, PDF object paths may be available in minimum, restricted, or full pCOS mode. Trying to retrieve a pCOS path which is inappropriate for the respective mode will raise an exception.

Table 6.6 Resulting pCOS modes for various password combinations

If you know...	...pCOS will run in...
none of the passwords	restricted pCOS mode if no user password is set, minimum pCOS mode otherwise
only the user password	restricted pCOS mode
the master password	full pCOS mode



# 7 PLOP and PLOP DS Library API Reference

## 7.1 Option Lists

Option lists are a powerful yet easy method to control PLOP operations. Instead of requiring a multitude of function parameters, many API methods support option lists, or optlists for short. These are strings which may contain an arbitrary number of options. Optlists support various data types and composite data like arrays. In most languages optlists can easily be constructed by concatenating the required keywords and values. C programmers may want to use the *sprintf()* function in order to construct optlists. An optlist is a string containing one or more pairs of the form

```
name value(s)
```

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

```
name=value
```

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

- ▶ Boolean: *true* or *false*; if the value of a boolean option is omitted, the value *true* is assumed. As a shorthand notation *none* can be used instead of *name false*.
- ▶ String: strings containing whitespace or '=' characters must be bracketed with { and }. An empty string can be constructed with {}. The characters {, }, and \ must be preceded by an additional \ character if they are supposed to be part of the string.
- ▶ Text strings are a special kind of string for certain options. While most options of type string accept only ASCII values, text strings may also carry Unicode values beyond ASCII. In Unicode-aware language bindings you can simply supply arbitrary Unicode values for such options. In non-Unicode-aware language bindings the user must prepend a UTF-8 BOM to text strings if the string is to be interpreted as UTF-8 (or EBCDIC UTF-8 on iSeries and zSeries). If no UTF-8 BOM is present, text strings will be interpreted in *auto* encoding, i.e. the current code page on Windows, the current job's encoding on iSeries, *ebcdic* on zSeries, and *iso8859-1* on Unix and Mac OS X.
- ▶ Keyword: one of a predefined list of fixed keywords
- ▶ Float and integer: decimal floating point or integer numbers; point and comma can be used as decimal separators.
- ▶ Handle: several internal object handles, e.g., document or page handles. Technically these are integer values.

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

```
password={secret string}  
linearize=true
```

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

```
permissions={ noprint nocopy }
```

*Note The backslash \ character requires special handling in many programming languages*

# 7.2 General Functions

<i>Perl PHP</i>	<i>resource PLOP_new()</i>
<i>C</i>	<i>PLOP *PLOP_new(void)</i>
	Create a new PLOP context.
<i>Returns</i>	A handle to the new context, or NULL if not enough memory is available. The context must be supplied to all other API functions.
<i>Bindings</i>	Not available in object-oriented language bindings where it will be called automatically when a new PLOP object is created.
<i>Java</i>	<i>void delete()</i>
<i>C#</i>	<i>void Dispose()</i>
<i>Perl PHP</i>	<i>PLOP_delete(resource plop)</i>
<i>C</i>	<i>void PLOP_delete(PLOP *plop)</i>
	Delete a PLOP context and release all its internal resources.
<i>Details</i>	All open documents in the context are closed automatically. It is good programming practice, however, to close documents explicitly with <i>PLOP_close_document()</i> when they are no longer needed.
<i>Bindings</i>	In C this function must not be called within a <i>PLOP_TRY()/PLOP_CATCH()</i> clause.  In Java this method will be called by the finalizer method of PLOP. However, it is strongly recommended to explicitly call <i>delete()</i> for proper cleanup. The same holds true when an exception occurred.  In PHP and COM this function will be called automatically when the PLOP object is destroyed.  In .NET <i>Dispose()</i> should be called at the end of processing to clean up unmanaged resources.
<i>C++</i>	<i>void create_pvf(string filename, const void *data, size_t size, string optlist)</i>
<i>C# Java</i>	<i>void create_pvf(String filename, byte[] data, String optlist)</i>
<i>Perl PHP</i>	<i>PLOP_create_pvf(resource plop, string filename, string data, string optlist)</i>
<i>VB</i>	<i>Sub create_pvf(filename As String, data, optlist As String)</i>
<i>C</i>	<i>void PLOP_create_pvf(PLOP *plop, const char *filename, int len, const void *data, size_t size, const char *optlist)</i>
	Create a named virtual read-only file from data provided in memory. This may be useful for repeatedly used digital IDs or XMP metadata.
	<b>filename</b> (Name string) The name of the virtual file. This is an arbitrary string which can later be used to refer to the virtual file in other PLOP calls.
	<b>len</b> (C language binding only) Length of <i>filename</i> (in bytes) for UTF-16 strings. If <i>len=0</i> a null-terminated string must be provided.

**data** A reference to the data for the virtual file. In COM this is a variant of byte containing the data comprising the virtual file. In C and C++ this is a pointer to a memory location. In Java this is a byte array. In Perl and PHP this is a string.

**size** (C and C++ only) The length in bytes of the memory block containing the data.

**optlist** An option list according to Table 7.1. The following option can be used: *copy*.

**Details** The virtual file name can be supplied to any API function which uses input files. Some of these functions may set a lock on the virtual file until the data is no longer needed. Virtual files will be kept in memory until they are deleted explicitly with *PLOP\_delete\_pvf()*, or automatically in *PLOP\_delete()*.

Each PLOP object will maintain its own set of PVF files. Virtual files cannot be shared among different PLOP objects. Multiple threads working with separate PLOP objects do not need to synchronize PVF use. If *filename* refers to an existing virtual file an exception will be thrown. This function does not check whether *filename* is already in use for a regular disk file.

Unless the *copy* option has been supplied, the caller must not modify or free (delete) the supplied data before a corresponding successful call to *PLOP\_delete\_pvf()*. Not obeying to this rule will most likely result in a crash.

Table 7.1 Options for *PLOP\_create\_pvf()*

option	description
<i>copy</i>	(Boolean) PLOP will immediately create an internal copy of the supplied data. In this case the caller may dispose of the supplied data immediately after this call. The copy option will automatically be set to true in the COM, .NET, and Java bindings (default for other bindings: false). In other language bindings the data will not be copied unless the copy option is supplied.

C++ *int delete\_pvf(string filename)*

C# Java *int delete\_pvf(String filename)*

Perl PHP *int PLOP\_delete\_pvf(resource plop, string filename)*

VB *Function delete\_pvf(filename As String) As Long*

C *int PLOP\_delete\_pvf(PLOP \*plop, const char \*filename, int len)*

Delete a named virtual file and free its data structures (but not the contents).

**filename** (Name string) The name of the virtual file as supplied to *PLOP\_create\_pvf()*.

**len** (C language binding only) Length of *filename* (in bytes) for UTF-16 strings. If *len=0* a null-terminated string must be provided.

**Returns** -1 if the corresponding virtual file exists but is locked, and 1 otherwise.

**Details** If the file isn't locked, PLOP will immediately delete the data structures associated with *filename*. If *filename* does not refer to a valid virtual file this function will silently do nothing. After successfully calling this function *filename* may be reused. All virtual files will automatically be deleted in *PLOP\_delete()*.

The detailed semantics depend on whether or not the *copy* option has been supplied to the corresponding call to *PLOP\_create\_pvf()*: If the *copy* option has been supplied, both the administrative data structures for the file and the actual file contents (data) will be freed; otherwise, the contents will not be freed, since the client is supposed to do so.



# 7.3 Document Input and Output Functions

*Note* PLOP currently does not support processing of multiple documents with a single PLOP object simultaneously. After opening a document with one of the `PLOP_open_document*()` functions you must close it before opening another document.

C++	<code>int open_document(string filename, string optlist)</code>
C# Java	<code>int open_document(String filename, String optlist)</code>
Perl PHP	<code>int PLOP_open_document(resource plop, string filename, string optlist)</code>
VB	<code>Function open_document(filename As String, optlist As String) As Long</code>
C	<code>int PLOP_open_document(PLOP *plop, const char *filename, int len, const char *optlist)</code>

Open a PDF document (which may be protected) for processing.

**filename** (Name string, but Unicode file names are only supported on Windows) The full path name of the PDF file to be opened. On Windows it is OK to use UNC paths or mapped network drives as long as you have the necessary permissions (which may not be the case when running in ASP).

In non-Unicode language bindings file names with `len = 0` will be interpreted in the current system codepage unless they are preceded by a UTF-8 BOM, in which case they will be interpreted as UTF-8 or EBCDIC-UTF-8.

**len** (C language binding only) Length of `filename` (in bytes) for UTF-16 strings. If `len=0` a null-terminated string must be provided.

**optlist** An option list (see Section 7.1, »Option Lists«, page 77) according to Table 7.2.

**Returns** -1 (in PHP: 0) on error, and a document handle otherwise. After an error it is recommended to call `PLOP_get_errmsg()` to find out more details about the error.

**Details** If the document is encrypted its user or master password must be supplied in the `password` option unless the `requiredmode` option has been specified.

Table 7.2 Options for `PLOP_open_document*()`

option	description
<b>inmemory</b>	(Boolean; only for <code>PLOP_open_document()</code> ) If true, PLOP will load the complete file into memory and process it from there. This can result in a tremendous performance gain on some systems (especially MVS) at the expense of memory usage. If false, individual parts of the document will be read from disk as needed. Default: false
<b>password</b>	(String up to 32 characters; required for encrypted documents except with <code>requiredmode</code> ) The user or master password for the document. As detailed in Table 4.2, page 43, the document's user password, master password, or no password may be required depending on which operation is applied to the document. On EBCDIC platforms the password is expected in ebcidc encoding.
<b>repair</b>	(Keyword) Specifies how to treat damaged PDF input documents. Repairing a document takes more time than normal parsing, but may allow processing of certain damaged PDFs. Note that some documents may be damaged beyond repair (default: auto): <b>force</b> Unconditionally try to repair the document, regardless of whether or not it has problems. <b>auto</b> Repair the document only if problems are detected while opening the PDF. <b>none</b> No attempt will be made at repairing the document. If there are problems in the PDF the function call will fail.

Table 7.2 Options for `PLOP_open_document()`

option	description
<b>requiredmode</b>	(Keyword) The minimum pcas mode (minimum/restricted/full) which is acceptable when opening the document. The call will fail if the resulting pcas mode would be lower than the required mode. If the call succeeds it is guaranteed that the resulting pcas mode is at least the one specified in this option. However, it may be higher; e.g. <code>requiredmode=minimum</code> for an unencrypted document will result in full mode. Default: full

---

```

C++ int open_document_callback(void *opaque, size_t filesize,
    size_t (*readproc)(void *opaque, void *buffer, size_t size),
    int (*seekproc)(void *opaque, long offset), const char *optlist)
C   int PLOP_open_document_callback(PLOP *plop, void *opaque, size_t filesize,
    size_t (*readproc)(void *opaque, void *buffer, size_t size),
    int (*seekproc)(void *opaque, long offset), const char *optlist)

```

---

Open a PDF document (which may be protected) via a user-supplied function.

**opaque** Pointer to some opaque data structure which will be passed to `readproc`. PLOP does not use this pointer or the underlying data.

**filesize** The length of the document in bytes.

**readproc** A procedure which must be able to supply arbitrary chunks of `size` bytes of the document at memory location `buffer`. The procedure must return the number of bytes retrieved.

**seekproc** A procedure for seeking to position `offset` within the document. The procedure must return -1 in case of error, and 0 otherwise.

**optlist** An option list (see Section 7.1, »Option Lists«, page 77) according to Table 7.2.

**Returns** -1 (in PHP: 0) on error, and a document handle otherwise. After an error it is recommended to call `PLOP_get_errmsg()` to find out more details about the error.

**Bindings** Only available in the C and C++ language bindings.

---

```

C++ int open_document_mem(byte[ ] data, string optlist)
C# Java int open_document_mem(byte[ ] data, String optlist)
Perl PHP PLOP_open_document_mem(resource plop, string data, string optlist)
VB Function open_document_mem(data As Variant, optlist As String) As Long
C   int PLOP_open_document_mem(PLOP *plop, const char *data, long length, const char *optlist)

```

---

Deprecated; use `PLOP_create_pvf()` and `PLOP_open_document()`.

<b>C++</b>	<b><code>int create_file(string filename, string optlist)</code></b>
<b>C# Java</b>	<b><code>int create_file(String filename, String optlist)</code></b>
<b>Perl PHP</b>	<b><code>int PLOP_create_file(resource plop, string filename, string optlist)</code></b>
<b>VB</b>	<b><code>Function create_file(filename As String, optlist As String) As Long</code></b>
<b>C</b>	<b><code>int PLOP_create_file(PLOP *plop, const char *filename, int len, const char *optlist)</code></b>

Create a PDF output document (which may be protected) in memory or on disk file.

**filename** (Name string, but Unicode file names are only supported on Windows) The name of the generated output file, which should be different from the input file name supplied to `PLOP_open_document()`. If this is an empty string the output will be generated in memory, and can later be fetched with `PLOP_get_buffer()`. On MVS systems an empty string cannot be used in combination with the *linearize* option.

In non-Unicode language bindings file names with *len* = 0 will be interpreted in the current system codepage unless they are preceded by a UTF-8 BOM, in which case they will be interpreted as UTF-8 or EBCDIC-UTF-8.

**len** (C language binding only) Length of *filename* (in bytes) for UTF-16 strings. If *len*=0 a null-terminated string must be provided.

**optlist** An option list (see Section 7.1, »Option Lists«, page 77) according to Table 7.3.

**Returns** -1 (in PHP: 0) on error, and a document handle otherwise. After an error it is recommended to call `PLOP_get_errmsg()` to find out more details about the error.

**Details** Before calling this function one of `PLOP_open_document*()` functions must have been called. The document opened with the most recent call to one of these functions will be processed. See Section 4.3, »PDF Security Features in PLOP«, page 43, for conditions which will be enforced for the user and master passwords.

Table 7.3 Options for `PLOP_create_file()`

option	description
compatibility	<p>(Keyword) Specify the document's PDF version as one of the strings 1.4, 1.5, 1.6, or 1.7 for Acrobat 5, 6, 7, or 8. This will be used to select the appropriate encryption algorithm (if the output is encrypted). The strongest possible encryption algorithm supported by the selected PDF version will be used (use 1.6 to force AES encryption). The selected PDF version may be increased automatically by other options according to the following rules:</p> <ul style="list-style-type: none"> <li>▶ Digitally signing the document (option <code>sign</code>) pushes the version to PDF 1.3.</li> <li>▶ Encryption, i.e. any of the options <code>userpassword</code>, <code>masterpassword</code>, and <code>permissions</code>, pushes the version to PDF 1.4.</li> <li>▶ Inserting XMP metadata (option <code>metadata</code>) pushes the version to PDF 1.4 unless the input conforms to PDF/X-1:2001, PDF/X-1a:2001, or PDF/X-3:2002. In these cases the version number will be left unchanged.</li> <li>▶ The <code>plainmetadata</code> keyword for the <code>permissions</code> option pushes the version to PDF 1.5.</li> </ul> <p>Default: the PDF version of the input document, or a higher version as mandated by the rules above.</p>

Table 7.3 Options for `PLOP_create_file()`

option	description
<b>docinfo</b>	<p>(List of pairs of text strings) Set document info entries for the output document. If the document contains document XMP metadata, the supplied document info entries will be mirrored in the XMP. Each pair contains the name of an entry and its value. The following predefined and custom keys can be supplied (default: document info entries will be copied from the input document):</p> <p><b>Subject</b> Subject of the document</p> <p><b>Title</b> Title of the document</p> <p><b>Author</b> Author of the document</p> <p><b>Keywords</b> Keywords describing the contents of the document</p> <p><b>Trapped</b> Indicates whether trapping has been applied to the document. Allowed values are True, False, and Unknown. For PDF/X input Unknown is only allowed if sacrifice includes pdfx.</p> <p><b>any name other than Creator, CreationDate, Producer, ModDate, GTS_PDFXVersion, GTS_PDFXConformance, ISO_PDFEVersion</b> User-defined field name (must not contain any space character). PLOP supports an arbitrary number of fields. A custom field name should only be supplied once.</p>
<b>flush</b>	<p>(Keyword) Set the flushing strategy. This is only effective for in-memory generation (i.e., an empty filename) and affects the amount of data returned by <code>PLOP_get_buffer()</code>. When the linearize option is true, the flushing strategy must be none. (Default: none):</p> <p><b>none</b> The returned buffer is guaranteed to contain all data comprising the output document.</p> <p><b>content</b> <code>PLOP_get_buffer()</code> will stop every time a larger chunk of PDF content data (more specifically: a PDF stream object) has been processed, and the returned buffer will contain only parts of the output document.</p> <p><b>heavy</b> <code>PLOP_get_buffer()</code> will process smaller portions, and will be called more frequently.</p>
<b>linearize</b>	<p>(Boolean; can not be combined with sign or linearize) If true, the output document will be linearized. On MVS systems this option cannot be combined with in-memory generation (i.e. empty filename). Default: false</p>
<b>master-password</b>	<p>(String up to 32 characters) The master password for the document. If it is empty no master password will be applied. On EBCDIC platforms the password is expected in ebcdic encoding. Default: empty</p>
<b>metadata</b>	<p>(Option list; can not be combined with linearize) Supply XMP metadata for the document. PDF/A-1 identification entries are not allowed in the supplied XMP. The option list may contain the following options:</p> <p><b>filename</b> (Name string; required) The name of a file containing well-formed XMP metadata in UTF-8 format.</p> <p><b>validate</b> (Keyword) The XMP metadata will be validated according to the keyword. Default: pdfa1 if the input conforms to PDF/A-1 and the sacrifice option does not include pdfa1 (in this situation the validate option will be ignored); otherwise none. Supported keywords:</p> <p><b>none</b> No validation</p> <p><b>xmp2004</b> Validation according to the XMP 2004 specification</p> <p><b>pdfa1</b> Like xmp2004, plus testing for predefined properties and schemas, and extension schema validation according to PDF/A-1.</p>
<b>optimize</b>	<p>(Keyword) The optimization steps to be applied while processing the document (default: all):</p> <p><b>all</b> Apply all implemented optimizations.</p> <p><b>none</b> Don't apply any optimization; this will slightly speed up processing at the expense of file size.</p>
<b>permissions</b>	<p>(Keyword list; requires masterpassword) The access permission list for the output document. It contains any number of the noprint, nomodify, nocopy, noannots, noassemble, noforms, noaccessible, nohiresprint, and plainmetadata keywords (see Table 4.3, page 44). Default: empty</p>
<b>recordsize</b>	<p>(Integer; MVS only) The record size of the output file. Default: 0 (unblocked output)</p>

Table 7.3 Options for PLOP\_create\_file()

option	description
sacrifice	(List of keywords) This option can be used for controlling the behavior in case of conflicts between properties of the input PDF and the requested action. By default PLOP will not create any output if it detects a conflict, and throw an exception instead. However, you can sacrifice some property of the document in order to allow processing. The keywords below are supported; they will be ignored unless both the input and action triggers are true (default: empty list, i.e. an exception will be thrown in case of a conflict, and no output will be created): <b>encryptedattachments</b> (Input trigger: the document is not encrypted, but contains one or more encrypted file attachments; action trigger: the appropriate password for the encrypted file attachment has not been supplied with the password option). If this keyword is supplied, encrypted file attachments for which the password is not available will be removed. <b>fields</b> (Input trigger: the document contains form fields with NeedAppearances=true; action trigger: sign option). If this keyword is supplied, existing form fields (including existing signature fields which may be present) will be removed. <b>pdfa1</b> (Input trigger: the document conforms to PDF/A-1a:2005 or PDF/A-1b:2005; action triggers: any of the options userpassword, masterpassword, or permissions) If this keyword is supplied, PDF/A-1 input can be encrypted, but the PDF/A-1 conformance entries will be removed (i.e. the output will no longer be flagged as PDF/A-1). <b>pdfx</b> (Input trigger: the document conforms to PDF/X-1a, PDF/X-2, or PDF/X-3; action triggers: option sign with a signature rectangle on the page, or any of the options userpassword, masterpassword, or permissions) If this keyword is supplied, encrypting or adding a visible signature field inside the BleedBox (or the TrimBox/ArtBox if no BleedBox is present) will be allowed, but the PDF/X conformance entries will be removed (i.e. the output will no longer be flagged as PDF/X). <b>signatures</b> (Input trigger: the document contains one or more signatures; any action) If this keyword is supplied, existing signatures will be cleared (i.e. signature values, but not the corresponding form fields will be removed) in order to avoid creating output with invalid signatures.
sign	(Option list; can not be combined with linearize; only available in PLOP DS) Sign the created document according to the suboptions listed in Table 7.4.
tempfilename	(String; MVS only) Full file name for a temporary file needed for PLOP's internal processing. If empty, PLOP will generate a unique temp file name. The user is responsible for deleting the temporary file after PLOP_close_document(). If this option is supplied the filename parameter must not be empty. Default: empty
tempdirname	(String) Name of a directory where temporary files needed for PLOP's internal processing will be created. If empty, PLOP will generate temporary files in the current directory. This option will be ignored if the tempfilename option has been supplied. Default: empty
user-password	(String up to 32 characters; requires the masterpassword option) The user password for the document. If it is empty no user password will be applied. On EBCDIC platforms the password is expected in ebcdic encoding. Default: empty

Table 7.4 Suboptions for the sign option of `PLOP_create_file()` (only available in PLOP DS)

option	description
<b>appearance</b>	(Option list) Specifies the visual appearance of the form field which will hold the signature:
<b>fieldname</b>	(Text string; must not end in a period «.» character) Name of the signature field. If the document contains a signature field with this name, it will be used for the signature (and page and rect will be ignored), otherwise the field will be created. If a field by this name exists, but has a type other than Signature, an exception will be thrown. Default: Signature1
<b>page</b>	(Integer) Number of the page on which the (visible or invisible) signature field will be created. The first page has number 1. Default: 1
<b>rect</b>	(Rectangle) Coordinates of the lower left and upper right corners of the signature field in PDF coordinates (one unit is 1/72 inch). Default: {0 0 0 0} which creates an invisible signature
<b>contactinfo</b>	(Text string) Information provided by the signer to enable a recipient to contact the signer to verify the signature (e.g. a phone number)
<b>digitalid</b>	(Option list; required) Specifies the signer's digital ID (certificate and private key) with exactly one of the following suboptions:
<b>filename</b>	(String) Name of a digital ID file in PKCS#12 (only for engine=builtin or CE interface) or PFX format. PKCS#12 and PFX files can also be supplied as virtual files, i.e. memory data which was assigned a file name with <code>PLOP_create_pvf()</code> . For engine=pkcs#11 this option contains the name of a PKCS#11 DLL/shared library for the cryptographic token, e.g. smartcard.
<b>certstore</b>	(Option list; only for engine=mscapi) Options for locating an ID in Windows' certificate store:
<b>subject</b>	(String; required) Search an ID where the «subject» entry contains the supplied string. It will usually hold the «common name» (CN) entry of the digital ID.
<b>store</b>	(String) Name of the certificate store (common names: My, root, trust, CA). Default: My
<b>keyusage</b>	(Option list; only for engine=pkcs#11) Criteria for selecting the target ID if multiple IDs are present (e.g. on a smartcard). Each keyword corresponds to a bit in the KeyUsage certificate extension. The corresponding value for each keyword specifies whether the extension bit must be 1 (set), 0 (clear), or will be ignored (ignore) when selecting the ID. PLOP DS will use the ID which matches the specified criteria. If no matching ID was found an exception will be thrown. See RFC 3280 for a detailed description of the key usage extension. The following keywords are supported: clear, ignore, set. The default is ignore for all entries.
	<b>digitalsignature</b> (Keyword) One of the keywords clear/ignore/set to specify handling of the digitalsignature key usage extension (i.e. bit 0).
	<b>nonrepudiation</b> (Keyword) One of the keywords clear/ignore/set to specify handling of the nonrepudiation key usage extension (i.e. bit 1).
	For example, if a smartcard contains two IDs where the ID with the nonrepudiation flag must be used for signing, the following option list can be specified: digitalid={filename=cryptoki.dll keyusage={nonrepudiation=set}}
<b>engine</b>	(Keyword) Specifies the crypto engine to be used for digital signatures (default: builtin):
<b>builtin</b>	Use the built-in crypto engine; digital IDs must be fetched from a disk file (PFX or PKCS#12).
<b>mscapi</b>	(Only on Windows) Use Microsoft Crypto API as crypto engine; digital IDs can be fetched from the certificate store or from a disk file (PFX only).
<b>pkcs#11</b>	(Only on selected platforms) Use the PKCS#11 interface to fetch the certificate from a cryptographic token. The name of the corresponding PKCS#11 DLL/shared library for the token must be provided in the filename suboption of the digitalid option.
<b>location</b>	(Text string) Physical location of the signing

Table 7.4 Suboptions for the sign option of `PLOP_create_file()` (only available in PLOP DS)

option	description
password	(String which may be empty; for engine=builtin exactly one of password or passwordfile is required; other engines may use alternate methods) Specifies the password, pass phrase, or PIN for the digital ID. For engine=pkcs#11 this option must contain the PIN for the cryptographic token unless the PIN must be entered interactively on the token itself (e.g. a smartcard reader with keyboard). On EBCDIC platforms the password is expected in ebcdic encoding.
passwordfile	(String; for engine=builtin exactly one of password or passwordfile is required; other engines may use alternate methods) The first line of the file (excluding the line end character or characters) will be used as password, pass phrase, or PIN for the digital ID. On EBCDIC platforms the contents of the password file are expected in ebcdic encoding.
reason	(Text string) Reason for signing the document

---

```

C++ byte[ ] get_buffer()
C# Java byte[ ] get_buffer()
Perl PHP string PLOP_get_buffer(resource plop)
VB Function get_buffer() As Variant
C const char *PLOP_get_buffer(PLOP *plop, long *size)

```

---

Fetch full or partial buffer contents of the output document from memory.

**size** Only required in the C binding. A pointer to a memory location where the length of the returned buffer will be stored.

**Returns** A buffer containing output data. In COM this is a Variant array of unsigned bytes. JavaScript with COM does not allow to retrieve the length of the returned variant array (but it does work with other languages and COM). The client must consume the buffer contents before calling any other PLOP library function.

**Details** PDF output can only be fetched with this function if in-memory generation has been requested by supplying an empty file name to *PLOP\_create\_file()* (otherwise output will be written directly to a file). *PLOP\_get\_buffer()* must be called before calling *PLOP\_close\_document()*.

If the *flush* option of *PLOP\_create\_file()* has its default value of *none* the returned buffer is guaranteed to contain all data for the output document. With *flush=content* *PLOP\_get\_buffer()* will stop every time a larger chunk of PDF content data (more specifically: a PDF stream object) has been processed, and the returned buffer will contain only a fragment of the output document. With a *flush=heavy* this function will process smaller portions, and will be called more frequently.

Unless *flush=none*, *PLOP\_get\_buffer()* must be called in a loop until it returns an empty buffer. The client must concatenate the returned fragments in order to generate the full output document. If *flush=none* a single call to *PLOP\_get\_buffer()* is sufficient.

**See also** The behavior of this function is controlled by the *flush* option of *PLOP\_create\_file()*.

---

```

C++ void close_document(int doc)
C# Java close_document(int doc)
Perl PHP PLOP_close_document(resource plop, long doc)
VB Sub close_document(doc As Long)
C void PLOP_close_document(PLOP *plop, int doc)

```

---

Close the input and output documents.

**doc** A valid document handle obtained with *PLOP\_open\_document\*()*.

**Details** This function must be called for cleanup when processing is done, and before *PLOP\_delete()* is called.



# 7.4 Exception Handling

PLOP supplies auxiliary methods for handling library exceptions in the C language. Other PLOP language bindings use the native exception handling system of the respective language, such as *try/catch* clauses. The language wrappers will pack information about exception number, description, and API function name into the generated exception object. In the Java language binding these items can be retrieved selectively.

When a PLOP exception occurred, no other PLOP function except *PLOP\_delete()* may be called with the corresponding PLOP object.

The PLOP language bindings for Java and .NET define a separate *PLOplibException* object which offers several members to access detailed error information.

<b>C++</b>	<i>int get_errnum()</i>
<b>C# Java</b>	<i>int get_errnum()</i>
<b>Perl PHP</b>	<i>int PLOP_get_errnum(resource plop)</i>
<b>VB</b>	<i>Function get_errnum() As Long</i>
<b>C</b>	<i>int PLOP_get_errnum(PLOP *plop)</i>

Get the number of the last thrown exception, or the reason of a failed function call.

*Returns* The exception's error number.

*Bindings* In .NET this method is also available as *Errnum* in the *PLOplibException* object. In Java this method is also available as *get\_errnum()* in the *PLOplibException* object.

<b>C++</b>	<i>string get_errmsg()</i>
<b>C# Java</b>	<i>String get_errmsg()</i>
<b>Perl PHP</b>	<i>string PLOP_get_errmsg(resource plop)</i>
<b>VB</b>	<i>Function get_errmsg() As String</i>
<b>C</b>	<i>const char *PLOP_get_errmsg(PLOP *plop)</i>

Get the descriptive text of the last thrown exception, or the reason of a failed function call.

*Returns* A string describing the error, or an empty string if the last API call didn't cause any error.

*Bindings* In .NET this method is also available as *Errmsg* in the *PLOplibException* object. In Java this method is also available as *getMessage()* in the *PLOplibException* object.

<b>C++</b>	<i>string get_apiname()</i>
<b>C# Java</b>	<i>String get_apiname()</i>
<b>Perl PHP</b>	<i>string PLOP_get_apiname(resource plop)</i>
<b>VB</b>	<i>Function get_apiname() As String</i>
<b>C</b>	<i>const char *PLOP_get_apiname(PLOP *plop)</i>

Get the name of the API function which threw the last exception or failed.

*Returns* The name of a PLOP API function.

*Bindings* In .NET this method is also available as *Apiname* in the *PLOplibException* object.  
In Java this method is also available as *get\_apiname()* in the *PLOplibException* object.

---

**C** *PLOP\_TRY(PLOP \*plop)*

---

Set up an exception handling frame; must always be paired with *PLOP\_CATCH()*.

*Details* See »Exception Handling in C«, page 27.

---

**C** *PLOP\_CATCH(PLOP \*plop)*

---

Catch an exception; must always be paired with *PLOP\_TRY()*.

*Details* See »Exception Handling in C«, page 27.

---

**C** *PLOP\_EXIT\_TRY(PLOP \*plop)*

---

Inform the exception machinery that a *PLOP\_TRY()* will be left without entering the corresponding *PLOP\_CATCH()* clause.

*Details* See »Exception Handling in C«, page 27.

---

**C** *PLOP\_RETHROW(PLOP \*plop)*

---

Re-throw an exception to another handler.

*Details* See »Exception Handling in C«, page 27.

# 7.5 Option Handling

C++  
C# Java  
Perl PHP  
VB  
C

void set\_option(string optlist)  
void set\_option(String optlist)  
PLOP\_set\_option(resource plop, string optlist)  
Sub set\_option(optlist As String)  
void PLOP\_set\_option(PLOP \*plop, const char \*optlist)

Set one or more global options for PLOP.

**optlist** An option list specifying global options according to Table 7.5. If an option is provided more than once the last instance will override all previous ones. In order to supply multiple values for a single option (e.g. *searchpath*) supply all values in a list argument to this option.

**Details** Multiple calls to this function can be used to accumulate values for those options marked in Table 7.5. For unmarked options the new value will override the old one.

Table 7.5 Global options for PLOP\_set\_option()

option	description
license	(String) Set the license key. It must be set before the first call to PLOP_open_document*().
licensefile	(String) Set the name of a file containing the license key(s). The license file can be set only once before the first call to PLOP_open_document*(). Alternatively, the name of the license file can be supplied in an environment variable called PLOPLICENSEFILE or (on Windows) via the registry.
frontpage	(Boolean) If false, an exception will be thrown if no valid license key was found; if true, a front page will be created in evaluation mode according to Section 0.1, »Installing the Software«, page 5. This option must be set before the first call to PLOP_open_document*(). It doesn't have any effect if a valid license key was found. Default: true
searchpath <sup>1</sup>	(List of name strings) Relative or absolute path name(s) of a directory containing files to be read. The search path can be set multiply; the entries will be accumulated and used in least-recently-set order. An empty string deletes all existing search path entries. On Windows the searchpath can also be set via a registry entry. Default: empty

1. Option values can be accumulated with multiple calls.

## 7.6 pCOS Functions

The full pCOS syntax for retrieving object data from a PDF is supported; see Chapter 6, »The pCOS Interface«, page 61, for a detailed description.

---

```
C++ double pcos_get_number(int doc, string path)
C# Java double pcos_get_number(int doc, String path)
Perl PHP double PLOP_pcos_get_number(resource plop, long doc, string path)
VB Function pcos_get_number(doc as Long, path As String) As Double
C double PLOP_pcos_get_number(PLOP *plop, int doc, const char *path, ...)
```

---

Get the value of a pCOS path with type *number* or *boolean*.

**doc** A valid document handle obtained with *PLOP\_open\_document\*( )*.

**path** A full pCOS path for a numerical or boolean object.

**Additional parameters** (C language binding only) A variable number of additional parameters can be supplied if the *key* parameter contains corresponding placeholders (%s for strings or %d for integers; use %% for a single percent sign). Using these parameters will save you from explicitly formatting complex paths containing variable numerical or string values. The client is responsible for making sure that the number and type of the placeholders matches the supplied additional parameters.

**Returns** The numerical value of the object identified by the pCOS path. For Boolean values 1 will be returned if they are *true*, and 0 otherwise.

---

```
C++ string pcos_get_string(int doc, string path)
C# Java String pcos_get_string(int doc, String path)
Perl PHP string PLOP_pcos_get_string(resource plop, long doc, string path)
VB Function pcos_get_string(doc as Long, path As String) As String
C const char *PLOP_pcos_get_string(PLOP *plop, int doc, const char *path, ...)
```

---

Get the value of a pCOS path with type *name*, *string*, or *boolean*.

**doc** A valid document handle obtained with *PLOP\_open\_document\*( )*.

**path** A full pCOS path for a string, name, or boolean object.

**Additional parameters** (C language binding only) A variable number of additional parameters can be supplied if the *key* parameter contains corresponding placeholders (%s for strings or %d for integers; use %% for a single percent sign). Using these parameters will save you from explicitly formatting complex paths containing variable numerical or string values. The client is responsible for making sure that the number and type of the placeholders matches the supplied additional parameters.

**Returns** A string with the value of the object identified by the pCOS path. For Boolean values the strings *true* or *false* will be returned.

**Details** This function will raise an exception if pCOS does not run in full mode and the type of the object is *string* (see Section 6.6, »Encrypted PDF Documents«, page 75). As an exception, the objects */Info/\** (document info keys) can also be retrieved in restricted pCOS

mode if *nocopy=false* or *plainmetadata=true*, and *bookmarks[...]/Title* and *annots[...]/contents* can be retrieved in restricted pCOS mode if *nocopy=false*.

This function assumes that strings retrieved from the PDF document are text strings. String objects which contain binary data should be retrieved with *PLOP\_pcos\_get\_stream()* instead which does not modify the data in any way.

**Bindings** C and C++ language bindings: The string will be returned in UTF-8 format without BOM. C binding: The returned string can be used until the next call to this function.

---

C++	<code>const unsigned char *pcos_get_stream(int doc, int *length, string optlist, string path)</code>
C# Java	<code>byte[ ] pcos_get_stream(int doc, String optlist, String path)</code>
Perl PHP	<code>string PLOP_pcos_get_stream(resource plop, long doc, String optlist, string path)</code>
VB	<code>Function pcos_get_stream(doc as Long, optlist As String, path As String)</code>
C	<code>const unsigned char *PLOP_pcos_get_stream(PLOP *plop, int doc, int *length, const char *optlist, const char *path, ...)</code>

---

Get the contents of a pCOS path with type *stream*, *fstream*, or *string*.

**doc** A valid document handle obtained with *PLOP\_open\_document\*()*.

**length** (C and C++ language bindings only) A pointer to a variable which will receive the length of the returned stream data in bytes.

**optlist** An option list specifying stream retrieval options according to Table 7.6.

**path** A full pCOS path for a stream or string object.

**Additional parameters** (C language binding only) A variable number of additional parameters can be supplied if the *key* parameter contains corresponding placeholders (%s for strings or %d for integers; use %% for a single percent sign). Using these parameters will save you from explicitly formatting complex paths containing variable numerical or string values. The client is responsible for making sure that the number and type of the placeholders matches the supplied additional parameters.

**Returns** The unencrypted data contained in the stream or string. The returned data will be empty (in C and C++: NULL) if the stream or string is empty.

If the object has type *stream*, all filters will be removed from the stream contents (i.e. the actual raw data will be returned). If the object has type *fstream* or *string* the data will be delivered exactly as found in the PDF file, with the exception of ASCII85 and ASCII-Hex filters which will be removed.

**Details** This function will throw an exception if pCOS does not run in full mode (see Section 6.6, »Encrypted PDF Documents«, page 75). As an exception, the object */Root/Metadata* can also be retrieved in restricted pCOS mode if *nocopy=false* or *plainmetadata=true*. An exception will also be thrown if *path* does not point to an object of type *stream*, *fstream*, or *string*.

Despite its name this function can also be used to retrieve objects of type *string*. Unlike *PLOP\_pcos\_get\_string()*, which treats the object as a text string, this function will not modify the returned data in any way. Binary string data is rarely used in PDF, and cannot be reliably detected automatically. The user is therefore responsible for selecting the appropriate function for retrieving string objects as binary data or text.

**Bindings** COM: Most client programs will use the Variant type to hold the stream contents. JavaScript with COM does not allow to retrieve the length of the returned variant array (but it does work with other languages and COM).  
C and C++ language bindings: The returned data buffer can be used until the next call to this function.

*This function can be used to retrieve embedded font data from a PDF. Users are reminded of the fact that fonts are subject to the respective font vendor’s license agreement, and must not be reused without the explicit permission of the respective intellectual property owners. Please contact your font vendor to discuss the relevant license agreement.*

Table 7.6 Options for `PLOP_pcos_get_stream()`

option	description
<b>convert</b>	(Keyword; will be ignored for streams which are compressed with unsupported filters) Controls whether or not the string or stream contents will be converted (default: none) :
<b>none</b>	Treat the contents as binary data without any conversion.
<b>unicode</b>	Treat the contents as textual data (i.e. exactly as in <code>PLOP_pcos_get_string()</code> ), and normalize it to Unicode. In non-Unicode-aware language bindings this means the data will be converted to UTF-8 format without BOM. <i>This option is required for the data type »text stream« in PDF which is rarely used (e.g. it can be used for JavaScript, although the majority of JavaScripts is contained in string objects, not stream objects).</i>

# 7.7 Unicode Conversion Functions

These functions may be useful for Unicode string conversion. They are provided for the benefit of users working with language environments that are not Unicode-aware.

C++	<code>string utf16_to_utf8(string utf16string)</code>
Perl PHP	<code>string PLOP_utf16_to_utf8(resource p, string utf16string)</code>
C	<code>const char *PLOP_utf16_to_utf8(PLOP *p, const char *utf16string, int len, int *size)</code>
<hr/>	
Convert a string from UTF-16 format to UTF-8.	
<b>utf16string</b> The string to be converted. A Byte Order Mark (BOM) in the string will be interpreted. If it is missing the platform's native byte ordering is assumed.	
<b>len</b> (C language binding only) Length of <i>utf16string</i> in bytes.	
<b>size</b> (C language binding only) C-style pointer to a memory location where the length of the returned string (in bytes) will be stored. If the pointer is NULL it will be ignored.	
Returns	The converted UTF-8 string. The generated UTF-8 string will start with the UTF-8 BOM ( <code>\xEF\xBB\xBF</code> ). On EBCDIC platforms the conversion result including the BOM will finally be converted to EBCDIC. The returned string is valid until the next call to any function, or until an exception is thrown. Clients must copy the string if they need it longer. The memory used for the converted string will be managed by PLOP.
Bindings	This function is not available in Unicode-capable language bindings.

C++	<code>string utf8_to_utf16(string utf8string, string ordering)</code>
Perl PHP	<code>string PLOP_utf8_to_utf16(resource p, string utf8string, string ordering)</code>
C	<code>const char *PLOP_utf8_to_utf16(PLOP *p, const char *utf8string, const char *ordering, int *size)</code>
<hr/>	
Convert a string from UTF-8 format to UTF-16.	
<b>utf8string</b> The string to be converted, which must contain a valid UTF-8 sequence (on EBCDIC platforms it must be encoded in EBCDIC). If a Byte Order Mark (BOM) is present, it will be removed.	
<b>ordering</b> Specifies the byte ordering of the result string:	
<ul style="list-style-type: none"><li>▶ <i>utf16</i> or an empty string: the converted string will not have any BOM, and will be stored in the platform's native byte order.</li><li>▶ <i>utf16le</i>: the converted string will be formatted in little endian format, and will be prefixed with the little-endian BOM (<code>\xFF\xFE</code>).</li><li>▶ <i>utf16be</i>: the converted string will be formatted in big endian format, and will be prefixed with the big-endian BOM (<code>\xFE\xFF</code>).</li></ul>	
<b>size</b> (C language binding only) C-style pointer to a memory location where the length of the returned string (in bytes) will be stored.	
Returns	The converted UTF-16 string. In C it will be terminated by two null bytes. The returned string is valid until the next call to any function, or until an exception is thrown. Clients must copy the string if they need it longer. The memory used for the converted string will be managed by PLOP.

*Bindings* This function is not available in Unicode-capable language bindings.



# A Combining PDFlib with PLOP/ PLOP DS

Depending on the PDFlib version number it may make sense to combine PLOP and PDFlib, PDFlib+PDI or PDFlib Personalization Server (PPS). Table A.1 summarizes the availability of encryption, linearization, optimization/repair mode, and digital signature in the PDFlib family. Attaching PLOP or PLOP DS to PDFlib makes sense in all situations where you need a feature which is not supported by your PDFlib version.

Table A.1 Encryption, linearization, optimization, and signature support in various PDFlib versions

PDFlib version	Encryption	Linearization	Optimization and repair mode	Digital Signature
PDFlib Lite (all versions)	–	–	–	–
PDFlib/PDFlib+PDI/PPS 5	yes	–	–	–
PDFlib/PDFlib+PDI/PPS 6	yes	yes	–	–
PDFlib/PDFlib+PDI/PPS 7	yes	yes	yes	–

PLOP has been designed for easy interoperability with PDFlib for dynamically generating and post-processing PDF. In this chapter we discuss how you can combine both products. Although it is possible to use the PLOP command-line tool to post-process documents generated with PDFlib, it is recommended to use the PLOP library to do so.

*Note Since PDFlib 6 and 7 do not create Appearance streams for form fields, you cannot use PLOP to sign PDFlib-generated documents containing form fields unless you remove the form fields with the sacrifice option.*

**File-Based Combination.** The file-based method is recommended if you deal with very large PDF documents, or if you need to reduce the total memory requirements of the PDFlib/PLOP combination. Simply generate a PDF file on disk with appropriate PDFlib routines, and subsequently process it with `PLOP_open_document()`.

**Memory-Based Combination.** The memory-based method is faster, but requires more memory. It is recommended for dynamic PDF generation and signature in Web applications unless you deal with very large documents. Instead of generating a PDF file on disk with PDFlib, use in-core PDF generation by supplying an empty file name to `PDF_begin_document()`, fetch the contents of the buffer containing the generated PDF data using `PDF_get_buffer()`, and pass this buffer to PLOP/PLOP DS using `PLOP_open_document_mem()`. Note that it is not possible to fetch the PDFlib buffer contents in multiple portions since the full document must be supplied to PLOP/PLOP DS in a single buffer. Therefore you must call `PDF_get_buffer()` between `PDF_end_document()` and `PDF_delete()`.

The *hellosign* programming sample, which is included in all PLOP packages, demonstrates how to use PDFlib for dynamically creating a PDF document and passing it to PLOP (in memory) for applying a digital signature.

# B PLOP Library Quick Reference

The following tables contain an overview of all PLOP API functions. The prefix (C) denotes C prototypes of functions which are not available in the Java language binding.

## General Functions

Function prototype	page
(C) PLOP *PLOP_new(void)	79
void delete()	79
void create_pvf(String filename, byte[] data, String optlist)	79
int delete_pvf(String filename)	80

## Document Input and Output

Function prototype	page
int open_document(String filename, String optlist)	81
(C) int PLOP_open_document_callback(PLOP *plop, void *opaque, size_t filesize, size_t (*readproc)(void *opaque, void *buffer, size_t size), int (*seekproc)(void *opaque, long offset), const char *optlist)	82
int open_document_mem(byte[] data, String optlist)	82
int create_file(String filename, String optlist)	83
close_document(int doc)	88
byte[] get_buffer()	88

## Error Handling

Function prototype	page
int get_errnum()	89
String get_errmsg()	89
String get_apiname()	89

## Option Handling

Function prototype	page
void set_option(String optlist)	91

## pCOS Functions

Function prototype	page
double pcos_get_number(int doc, String path)	92
String pcos_get_string(int doc, String path)	92
byte[] pcos_get_stream(int doc, String optlist, String path)	93

## Unicode Conversion Functions

Function prototype	page
(C) const char *PLOP_utf16_to_utf8(PLOP *p, const char *utf16string, int len, int *size)	95
(C) const char *PLOP_utf8_to_utf16(PLOP *p, const char *utf8string, const char *ordering, int *size)	95

# C Revision History

Revision history of this manual

Date	Changes
December 05, 2008	► Updates for XMP, PVF, and PKCS#11 (smartcard) support in PLOP 4.0 and PLOP DS 4.0
July 15, 2007	► Updates for PLOP 3.0 and PLOP DS 3.0
September 27, 2004	► Updates for PLOP 2.1
December 01, 2003	► Updated for new major release PLOP 2.0
November 23, 2002	► Added a description of the Perl binding for PSP
November 7, 2002	► Added a section on the use of PSP with ILE-RPG
October 22, 2002	► Minor changes for PSP 1.0.1
September 17, 2002	► First edition for PSP 1.0.0



# Index

## A

*Ad Ticket scheme* 18  
*arrays* 65

## B

*byteserving* 13

## C

*C binding* 27  
*C++ binding* 29  
*certificate organization in Windows* 51  
*certified PDF* 55  
*COM binding* 30  
*cracking protected PDF* 41  
*crypto engines* 50  
*cryptographic details of PLOP DS signatures* 54  
*cryptographic tokens* 50, 53

## D

*damaged input PDFs* 15  
*dictionaries* 65  
*dictionary attack* 41  
*digital IDs* 50  
*digital signatures* 49  
    *validation in Acrobat* 56  
*document info entries* 17  
*document info fields* 61  
*DSA-based signature* 54

## E

*electronic signatures* *see* *digital signatures*  
*encrypted file attachments* 21, 40  
*encrypted PDF documents* 75  
*encryption status* 62  
*evaluation version* 5  
*examples*  
    *document info fields* 61  
    *encryption status* 62  
    *fonts in a document* 62  
    *number of pages* 61  
    *page size* 62  
    *pCOS paths* 61  
*exception handling* 89  
*exit codes* 25  
*external crypto engine* 50

## F

*file attachments, encrypted* 40  
*font optimization* 14  
*fonts in a document* 62  
*form fields in the input document* 21

## G

*garbage collection* 14  
*Ghent Workgroup (GWG)* 18

## I

*incremental update* 55  
*installing PLOP/PLOP DS* 5

## J

*Java binding* 31

## L

*license key* 6  
*linearized PDF* 13

## M

*master password* 39  
*Microsoft Cryptographic API* 50

## N

*nesting exceptions* 28  
*.NET binding* 33  
*noaccessible* 44  
*noannots* 44  
*noassemble* 44  
*nocopy* 44  
*noforms* 44  
*nohiresprint* 44  
*nomodify* 44  
*noprint* 44  
*number of pages* 61

## O

*optimization* 14  
*optimized PDF* 13  
*option lists* 77  
*owner password* 39

## P

- page size* 62
- page-at-a-time download* 13
- password file for digital IDs* 54
- passwords* 39
  - good and bad* 41
- passwords for digital IDs* 54
- pCOS* 61
  - API functions* 92
  - data types* 63
  - encryption* 75
  - path syntax* 66
  - pseudo objects* 68
- PDF Reference Manual* 61
- PDF security features* 39
- PDF version of the generated output* 20
- PDF/A* 20
  - and XMP metadata* 18
- PDF/X* 20
- PDFlib and PLOP/PLOP DS* 97
- performance* 21
- Perl binding* 34
- permission settings* 39
  - enforcing* 41
- permissions password* 39
- PDF format* 50
- PHP binding* 35
- PKCS#11* 50, 53
- PKCS#12* 50
- plainmetadata* 44
- PLOP and PLOP DS command-line tool*
  - examples* 26
  - exit codes* 25
  - features* 9
  - options* 23
- PLOP and PLOP DS library*
  - API reference* 77
  - features* 9
  - quick reference* 98
- PLOP DS signatures, cryptographic details* 54
- PLOP\_CATCH()* 90
- PLOP\_close\_document()* 88
- PLOP\_create\_file()* 83
- PLOP\_create\_pvf()* 79
- PLOP\_delete()* 79
- PLOP\_delete\_pvf()* 80
- PLOP\_EXIT\_TRY()* 28, 90
- PLOP\_get\_apiname()* 89
- PLOP\_get\_buffer()* 88
- PLOP\_get\_errmsg()* 89
- PLOP\_get\_errnum()* 89
- PLOP\_new()* 79

- PLOP\_open\_document()* 81
- PLOP\_open\_document\_callback()* 82
- PLOP\_open\_document\_mem()* 82
- PLOP\_pcos\_get\_number()* 92
- PLOP\_pcos\_get\_stream()* 93
- PLOP\_pcos\_get\_string()* 92
- PLOP\_RETHROW()* 90
- PLOP\_set\_option()* 91
- PLOP\_TRY()* 90
- PLOP\_utf16\_to\_utf8()* 95
- PLOP\_utf8\_to\_utf16()* 95

## R

- Reader-enabled PDF* 21
- repair mode for damaged PDFs* 15
- response file* 25
- RPG binding* 37
- RSA-based signature* 54

## S

- sacrificing properties of the input document* 20
- security recommendations* 42
- signatures*
  - in the input document* 20
- signatures see digital signatures*
- signing PDF documents* 52
- smartcards* 50, 53
- stream optimization* 14
- strength of PDF encryption* 41

## T

- temporary disk space requirements* 21

## U

- unused objects* 14
- user password* 39

## V

- validating digital signatures in Acrobat* 56

## W

- web-optimized PDF* 13

## X

- XMP metadata* 17, 18, 62
- plaintext* 40