PDFlib GmbH

PLOP and PLOP DS

Version 4.1

PDF Linearization, Optimization, Protection, and Digital Signature

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

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

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

Contents

0 First Steps with PLOP and PLOP DS

0.1 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 25, 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 31, 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 (for COM and .NET) or in the installation directory (for other language bindings).

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 password 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 the PLOP package.

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.

0.2 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.

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.

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

- In COM/VBScript:

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

- In .NET/C#:

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

- In C and Python:

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

- In C++ and Java:

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

- In Perl and PHP:

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

- In RPG:

```
d licenseopt     s           20
c                eval        licenseopt=%ucs2('license=... your license key ...')
c                callp       PLOP_set_option(PLOP:licenseopt:0)
```

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

```
# Licensing information for PDFlib GmbH products
PDFlib license file 1.0
PLOP        4.1             ...your license key...
```

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

- A file called *licensekeys.txt* will be searched in all default locations (see »Default file search paths«, page 7).
- You can specify the *licensefile* parameter with the *set_option()* API function:

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

- Use the *--plopopt* option of the PLOP command-line tool and supply the *licensefile* option with the name of a license file:

  ```
  plop --plopopt "licensefile /path/to/your/licensekeys.txt" ...
  ```

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

  ```
  tet --tetopt "licensefile {/path/to/your/license file.txt}" ...
  ```

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

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

- On i5/iSeries systems the license file must be encoded in ASCII (see *asciifile* option). The license file can be specified as follows (this command can be specified in the startup program *QSTRUP* and will work for all PDFlib GmbH products):

  ```
  ADDENVVAR ENVVAR(PDFLIBLICENSEFILE) VALUE(/PLOP/4.1/licensefile.txt) LEVEL(*SYS)
  ```

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

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

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

```
HKLM\SOFTWARE\PDFlib\PLOP4\license
HKLM\SOFTWARE\PDFlib\PLOP4\4.1\license
```

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

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

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

Default file search paths. On Unix, Linux, Mac OS X and i5/iSeries systems some directories will be searched for files by default even without specifying any path and directory names. The following directories will be searched:

```
<rootpath>/PDFlib/PLOP/4.1/resource/cmap
<rootpath>/PDFlib/PLOP/4.1/resource/codelist
<rootpath>/PDFlib/PLOP/4.1/resource/glyphlst
<rootpath>/PDFlib/PLOP/4.1/resource/fonts
<rootpath>/PDFlib/PLOP/4.1/resource/icc
<rootpath>/PDFlib/PLOP/4.1
<rootpath>/PDFlib/PLOP
<rootpath>/PDFlib
```

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

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

```
licensekeys.txt             (license file)
```

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

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

```
PDFlib license file 2.0
# Licensing information for PDFlib GmbH products
PLOP        4.1     ...your license key...      ...serial number of machine 1...
PLOP        4.1     ...your license key...      ...serial number of machine 2...
```

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

Licensing options. Different licensing options are available for PLOP use on one or more servers, and for redistributing PLOP with your own products. We also offer support and source code contracts. Please contact us if you are interested in obtaining a commercial PLOP license or have any questions:

PDFlib GmbH, Licensing Department
Franziska-Bilek-Weg 9, 80339 München, Germany
www.pdflib.com
phone +49 • 89 • 452 33 84-0
fax +49 • 89 • 452 33 84-99
Licensing contact: *sales@pdflib.com*
Support for PDFlib licensees: *support@pdflib.com*

1 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 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. PDFlib PLOP and 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 (unless when applying digital signatures).

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 and Windows 7 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 25. 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 29 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.pdflib.com/pcos-cookbook*.

Details of the pCOS interface are documented in the pCOS Path Reference which is included in the PLOP package.

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 47. 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 52). 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 4.3, »Securing PDF Documents on the Command Line«, page 54.

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 byteserving. It allows Acrobat (running as a browser plugin) to retrieve individual parts of a PDF document separately. The result is that the first page of the document will be presented to the user without having to wait for the full document to download from the server. This provides enhanced user experience.

Note that the Web server streams PDF data to the browser, not 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 8/9/X: *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 whitespace 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
```

Invalid XMP metadata. PLOP repairs certain kinds of problems in XMP metadata. However, some problems cannot be repaired. For example XML parsing errors caused by XMP metadata always imply that the XMP is unusable. PLOP provides the *xmppolicy* option for controlling the processing behavior when invalid XMP is encountered. See »Dealing with invalid XMP metadata«, page 19, for more details.

1.7 Query Document Information with pCOS

The pCOS interface is covered in detail in the pCOS Path Reference. 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: 1.6
    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:20100616003116Z'
       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*[1]*)* 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/knowledge-base/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*.
- 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_ad_ticket.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

Dealing with invalid XMP metadata. PDF documents sometime contain invalid XMP metadata which is either invalid on the XML level or the XMP/RDF level. PLOP will by default reject such documents and stop processing. In order to provide more fine-grain control for such input documents the *xmppolicy* option for *PLOP_open_document( )* can be used to distinguish the following cases:

- *xmppolicy=rejectinvalid:* by default, invalid XMP prevents PLOP from generating PDF output.
- *xmppolicy=ignoreinvalid:* ignore invalid XMP and include the text of the XML parsing error message in the generated output XMP as a debugging aid. Note that no PDF/A or PDF/X-3/4/5 output can be created with this option.
- *xmppolicy=remove:* remove input XMP. This may be useful to delete unwanted metadata.

For example, if you don't want invalid XMP metadata to disrupt batch processing of documents you can ignore problems caused by invalid XMP in the input document:

```
plop --inputopt "xmppolicy=ignoreinvalid" --outfile output.pdf input.pdf
plop --inputopt "xmppolicy=ignoreinvalid" -o output.pdf input.pdf
```

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 57. 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 *demo2048.p12*. The password for the digital ID is contained in the file *pw.txt:*

```
plop --signopt "digitalid={filename=demo2048.p12} passwordfile=pw.txt" ↵
        --outfile signed.pdf input.pdf
plop -S "digitalid={filename=demo2048.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=mscapi digitalid={certstore={store=My subject={DEMO PLOP User 2048}}}" ↵
         --outfile signed.pdf input.pdf
plop -S "engine=mscapi digitalid={certstore={store=My subject={DEMO PLOP User 2048}}}" ↵
         -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 61.

1.11 PLOP Processing Details

Acceptable input documents. PLOP accepts, processes, and creates the following PDF flavors:

- PDF 1.4 (Acrobat 5) and all older versions
- PDF 1.5 (Acrobat 6)
- PDF 1.6 (Acrobat 7)
- PDF 1.7 (Acrobat 8), technically identical to ISO 32000-1
- PDF 1.7 Adobe extension level 3 (Acrobat 9)
- PDF 1.7 Adobe extension level 8 (Acrobat X)
- PDF 2.0 as specified in ISO 32000-2

Depending on the desired operation the password will be required for encrypted documents. 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 *--outputopt* option of the PLOP command-line tool. If this option has not been specified, PLOP uses 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 the document (option *sign)* pushes the version to PDF 1.3.
- Inserting XMP metadata (option *metadata)* pushes the version to PDF 1.4.
- The *plainmetadata* keyword for the *permissions* option pushes the version to PDF 1.5.

PDF/A implementation basis. The following standards and documents form the basis of the PDF/A implementation in PLOP:

- The PDF/A standard (ISO 19005-1:2005)
- Technical Corrigendum 1 (ISO 19005-1:2005/Cor 1:2007)
- Technical Corrigendum 2 (ISO 19005-1:2005/Cor.2:2010)
- All relevant TechNotes published by the PDF/A Competence Center.

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 *--outputopt* 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 7.4, page 79, 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={pdfa1}* option, though.

- PDF/X: PDF/X-1a/3/4/5 don't allow encryption, or visible signature fields on the page. In these situations PLOP will raise an exception, but 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.

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

- 2 GB file size limit: PDF/A-1 and other standards limit the file size to 2 GB. If a document gets larger than this limit, PLOP will throw an exception when creating PDF/A-1, PDF/X-4 or PDF/X-5 output. Otherwise documents beyond 2 GB can be created.
- 10 GB file size limit: the cross-reference table in PDF documents is limited to 10 decimal digits and therefore 10^{10}-1 bytes, which equates to roughly 9.3 GB. PLOP cannot create documents beyond this limit.
- Number of objects: while the object count in a document is not limited by PDF in general, the PDF/A-1, PDF/X-4 and PDF/X-5 standards limit the number of indirect objects in a document to 8.388.607. If a document requires objects beyond this limit, PLOP will throw an exception when creating PDF/A-1, PDF/X-4 or PDF/X-5 output. Otherwise documents with more objects can be created. The number of objects in PDF depends on the complexity of the page contents, number of interactive elements, etc. Since typical high-volume documents with simple contents require ca. 4-10 objects per page on average, documents with ca. 1-2 million pages can be created without exceeding the object limit.

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 71. 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[1]		*Specify a response file with options; for a syntax description see »Response files«, page 28. 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 PDF version of the generated PDF output document:* ***1.4*** *PDF 1.4 requires Acrobat 5 or above.* ***1.5*** *PDF 1.5 requires Acrobat 6 or above.* ***1.6*** *PDF 1.6 requires Acrobat 7 or above.* ***1.7*** *PDF 1.7 is specified in ISO 32000-1 and requires Acrobat 8 or above.* ***1.7ext3*** *PDF 1.7 extension level 3 requires Acrobat 9 or above.* ***1.7ext8*** *PDF 1.7 extension level 8 requires Acrobat X.* ***2.0*** *PDF 2.0 is specified in ISO 32000-2.* *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 rules detailed in Table 7.4, page 79.* *Default: the PDF version of the input document, or a higher version as mandated by the processing rules.*
--fast, -f		*Disable optimization step for faster processing.*

Table 2.1 PLOP command-line options

option	parameters	function
--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 7.3, page 76)*
--master[2,3], -m	<password>	*Output master password; missing option means no password.*
--noreplace, -n		*If the output file already exists, it will not be overwritten and an exception will be thrown. Default: existing output files will be overwritten.*
--outfile, -o	<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.*
--outputopt	<option list>	*Additional option list for PLOP_create_file( ) (see Table 7.4, page 79)*
--password[2], -p	<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.*
--permissions[2,3]	<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 52). In addition, the following keyword can be used (default: no permission restrictions):* **keep** *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`.
--plopopt	<option list>	*Additional option list for PLOP_set_option( ) (see Table 7.7, page 87). This can be used to pass the* `license` *or* `licensefile` *options.*
--recsize, -R	<blocksize>	*(MVS only) The record size of the output file. Default: 0 (unblocked)*
--tempfilename, -T	<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*
--tempdirname	<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*
--searchpath, -s[1]	<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*
--signopt, -S	<option list>	*(Only available in PLOP DS) Option list for the* `sign` *option of PLOP_create_file( ) for digitally signing documents (see Table 7.6, page 82).*
--targetdir, -t	<dirname>	*(One of* `--outfile` *and* `--targetdir` *must be supplied) Output directory name; the directory must already exist.*
--user, -u[2,3]	<password>	*Output user password; missing option means no password.*

Table 2.1 PLOP command-line options

option	*parameters*	*function*
--verbose, -v	0, 1, 2, 3	*Verbosity level (default: 1):* ***0*** *no output* ***1*** *only errors* ***2*** *errors and file names* ***3*** *detailed reporting*
--webopt, -w		*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 *--plopopt)*.
- 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 throw an exception 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 particular 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.

On Windows all file name options accept Unicode strings, e.g. as a result of dragging files from the Explorer to a command prompt window.

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. *@filename* can be used in another response file.

Response files may contain Unicode strings for file name and password arguments. Response files can be encoded in UTF-8, EBCDIC-UTF-8, or UTF-16 format and must start with the corresponding BOM. If no BOM is found, the contents of the response file will be interpreted in EBCDIC on zSeries, and in ISO 8859-1 (Latin-1) on all other systems including i5/iSeries.

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 4.3, »Securing PDF Documents on the Command Line«, page 54
- Section 5.3, »Signing PDF Documents with PLOP DS«, page 61.

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 (unless digital signatures are created at the same time), 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 *demo2048.p12*. The password for the digital ID is contained in the file *pw.txt:*

```
plop --signopt "digitalid={filename=demo2048.p12} passwordfile=pw.txt" ↵
        --outfile signed.pdf input.pdf
plop -S "digitalid={filename=demo2048.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

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

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

Using PLOP as a DLL loaded at runtime. While most clients will use PLOP as a statically bound library or a dynamic library which is bound at link time, you can also load the DLL at runtime and dynamically fetch pointers to all API functions. This is especially useful to load the DLL only on demand. PLOP supports a special mechanism to facilitate this dynamic usage. It works according to the following rules:

- Include *ploplibdl.h* instead of *ploplib.h*.
- Use *PLOP_new_dl( )* and *PLOP_delete_dl( )* instead of *PLOP_new( )* and *PLOP_delete( )*.
- Use *PLOP_TRY_DL( )* and *PLOP_CATCH_DL( )* instead of *PLOP_TRY( )* and *PLOP_CATCH( )*.
- Use function pointers for all other PLOP calls.
- Compile the auxiliary module *ploplibdl.c* and link your application against the resulting object file.

The dynamic loading mechanism is demonstrated in the *encryptdl.c* sample.

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

Exception handling. The PLOP API provides a mechanism for acting upon exceptions thrown by the library in order to compensate for the lack of native exception handling in the C language. Using the *PLOP_TRY( )* and *PLOP_CATCH( )* macros client code can be set up such that a dedicated piece of code is invoked for error handling and cleanup when an exception occurs. These macros set up two code sections: the try clause with code which may throw an exception, and the catch clause with code which acts upon an exception. If any of the API functions called in the try block throws an exception, program execution will continue at the first statement of the catch block immediately. The following rules must be obeyed in PLOP client code:

- *PLOP_TRY( )* and *PLOP_CATCH( )* must always be paired.
- *PLOP_new( )* will never throw an exception; since a try block can only be started with a valid PLOP object handle, *PLOP_new( )* must be called outside of any try block.
- *PLOP_delete( )* will never throw an exception, and therefore can safely be called outside of any try block. It can also be called in a catch clause.

- Special care must be taken about variables that are used in both the try and catch blocks. Since the compiler doesn't know about the transfer of control 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 this kind of problem: Variables used in both the try and catch blocks must be declared *volatile*. Using the *volatile* keyword signals to the compiler that it must not apply dangerous optimizations to the variable.
- If a try block is left (e.g., with a return statement, thus bypassing the invocation of the corresponding *PLOP_CATCH( ))*, the *PLOP_EXIT_TRY( )* macro must be called before the return statement to inform the exception machinery.
- As in all PLOP language bindings document processing must stop when an exception was thrown.

The following code fragment demonstrates these rules with the typical idiom for dealing with PLOP exceptions in client code (full samples can be found in the PLOP package):

```
if ((plop = PLOP_new()) == (PLOP *) 0)
{
        printf("out of memory\n");
        return(2);
}
PLOP_TRY(plop)
{
        /* statements that directly or indirectly call API functions */
}
PLOP_CATCH(plop)
{
        printf("Error %d in %s() on page %d: %s\n",
            PLOP_get_errnum(plop), PLOP_get_apiname(plop),
                pageno, PLOP_get_errmsg(plop));
}
PLOP_delete(plop);
```

Unicode handling for name strings. The C language does not natively support Unicode. Some string parameters for API functions may be declared as *name strings*. These are handled depending on the *length* parameter and the existence of a BOM at the beginning of the string. In C, if the *length* parameter is different from 0 the string will be interpreted as UTF-16. If the *length* parameter is 0 the string will be interpreted as UTF-8 if it starts with a UTF-8 BOM, or as EBCDIC UTF-8 if it starts with an EBCDIC UTF-8 BOM, or as *host* encoding if no BOM is found (or *ebcdic* on all EBCDIC-based platforms).

Unicode handling for option lists. Strings within option lists require special attention since they cannot be expressed as Unicode strings in UTF-16 format, but only as byte arrays. For this reason UTF-8 is used for Unicode options. By looking for a BOM at the beginning of an option PLOP decides how to interpret it. The BOM will be used to determine the format of the string. More precisely, interpreting a string option works as follows:

- If the option starts with a UTF-8 BOM *(\xEF\xBB\xBF)* it will interpreted as UTF-8.
- If the option starts with an EBCDIC UTF-8 BOM *(\x57\x8B\xAB)* it will be interpreted as EBCDIC UTF-8.

- If no BOM is found, the string will be treated as *winansi* (or *ebcdic* on EBCDIC-based platforms).

Note The PLOP_convert_to_unicode( ) utility function can be used to create UTF-8 strings from UTF-16 strings, which is useful for creating option lists with Unicode values.

3.2 C++ Binding

Note *For .NET applications written in C++ we recommend to access the PLOP .NET DLL directly instead of via the C++ binding (except for cross-platform applications which should use the C++ binding). The PLOP distribution contains C++ sample code for use with .NET CLI which demonstrates this combination.*

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*. Since *plop.hpp* contains a template-based implementation no corresponding *plop.cpp* module is required. 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.

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

```
#define PLOPCPP_DL    1
```

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

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

String handling in C++. PLOP 4.1 introduces a new Unicode-capable C++ binding. The new template-based approach supports the following usage patterns with respect to string handling:

- Strings of the C++ standard library type *std::wstring* are used as basic string type. They can hold Unicode characters encoded as UTF-16 or UTF-32. This is the default behavior in PLOP 4.1 and the recommended approach for new applications unless custom data types (see next item) offer a significant advantage over *wstrings*.
- Custom (user-defined) data types for string handling can be used as long as the custom data type is an instantiation of the *basic_string* class template and can be converted to and from Unicode via user-supplied converter methods.
- Plain C++ strings can be used for compatibility with existing C++ applications which have been developed against PLOP 4.0 or earlier versions. This compatibility variant is only meant for existing applications (see below for notes on source code compatibility).

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

Adjusting applications to the new C++ binding. Existing C++ applications which have been developed against PLOP 4.0 or earlier versions can be adjusted to PLOP 4.1 as follows:

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

```
using namespace pdflib;
```

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

```
const wstring docoptlist = L"password=foo";
```

- Suitable *wstring*-capable methods *(wcerr* etc.) must be used to process PLOP error messages and exception strings *(get_errmsg( )* method in the *PLOP* and *PLOP::Exception* classes).
- The *plop.cpp* module is no longer required for the PLOP C++ binding. Although the PLOP distribution contains a dummy implementation of this module, it should be removed from the build process for PLOP applications.

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

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

```
#define PLOPCPP_PLOP_WSTRING 0
```

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

```
#define PLOPCPP_USE_PDFLIB_NAMESPACE 0
```

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

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

```
try {
        ...some PLOP instructions...
} catch (PLOP::Exception &ex) {
            wcerr << L"Error " << ex.get_errnum()
            << L" in " << ex.get_apiname()
            << L"(): " << ex.get_errmsg() << endl;
}
```

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 39) 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.

Exception Handling in Java. All PLOP/PLOP DS methods will throw an exception of type *PLOPException* 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 (PLOPException 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

Note *Detailed information about the various flavors and options for using PLOP with the .NET Framework can be found in the PDFlib-in-.NET-HowTo.pdf document which is contained in the distribution packages and also available on the PDFlib Web site.*

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 *(PLOP_dotnet.dll)* contains the actual library plus meta information.

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.

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 *PLOP_dotnet.PLOPException* with the members *get_errnum*, *get_errmsg*, and *get_apiname*.

Using PLOP with C++ and CLI. .NET applications written in C++ (based on the *Common Language Infrastructure* CLI) can directly access the PLOP.NET DLL without using the PLOP C++ binding. The source code must reference PLOP as follows:

```
using namespace PLOP_dotnet;
```

3.6 Perl Binding

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

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

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

```
perl -I/path/to/plop encrypt.pl
```

Unix. Perl will search *plop_pl.so* (on Mac OS X: *plop_pl.bundle)*, *plop_pl.pm* and *PDFlib/PLOP.pm* in the current directory, or the directory printed by the following Perl command:

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

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

```
/usr/lib/perl5/site_perl/5.10/i686-linux
```

Windows. PLOP supports the ActiveState port of Perl 5 to Windows, also known as ActivePerl. The DLL *plop_pl.dll* and the modules *plop_pl.pm* and *PDFlib/PLOP.pm* will be searched in the current directory, or the directory printed by the following Perl command:

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

Typical output of the above command looks like

```
C:\Program Files\Perl5.10\site\lib
```

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=plop_php.so            ; for Unix and Mac OS X
extension=plop_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("plop_php.so");               # for Unix and Mac OS X
dl("plop_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 (PLOPException $e) {
    print "PLOP exception occurred:\n";
```

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

3.8 Python Binding

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

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

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

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

3.9 RPG Binding

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.

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

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

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

Compiling and binding RPG programs for PLOP. 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 QRPGLESRC,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/QRPGLESRC,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/QRPGLESRC) SRCMBR(*PGM) DFTACTGRP(*NO)
BNDDIR(PLOPLIB/PLOPLIB)
```

Error Handling in RPG. PLOP clients written in ILE-RPG can use the *monitor/on-error/endmon* error handling mechanism that ILE-RPG provides. Another way to monitor for exceptions is to use the **PSSR* global error handling subroutine in ILE-RPG. If an excep-

tion occurs, the job log shows the error number, the function that failed and the reason for the exception. PLOP sends an escape message to the calling program.

```
c     eval      p=PLOP_new
*
c     monitor
*
c     eval      doc=PLOP_open_document(p:%ucs2('/tmp/my.pdf'):inputoptlist)
:
:
*     Error Handling
c     on-error
*     Do something with this error
*     don't forget to free the PLOP object
c     callp     PLOP_delete(p)
c     endmon
```

4 PDF Security

4.1 PDF Security Features

PDF documents can be protected with password security which offers the following protection features:

- The user password (also referred to as open password) is required to open the file for 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 password.
- Permission settings restrict certain actions for the PDF document, such as printing or extracting text.
- An attachment password can be specified to encrypt only file attachments, but not the actual contents of the document itself.

If a PDF document uses any of these protection features it will be encrypted. In order to display or modify a document's security settings with Acrobat, click *File, Properties..., Security, Show Details...* or *Change Settings...*, respectively. Figure 4.1 shows the security settings dialog in Acrobat.

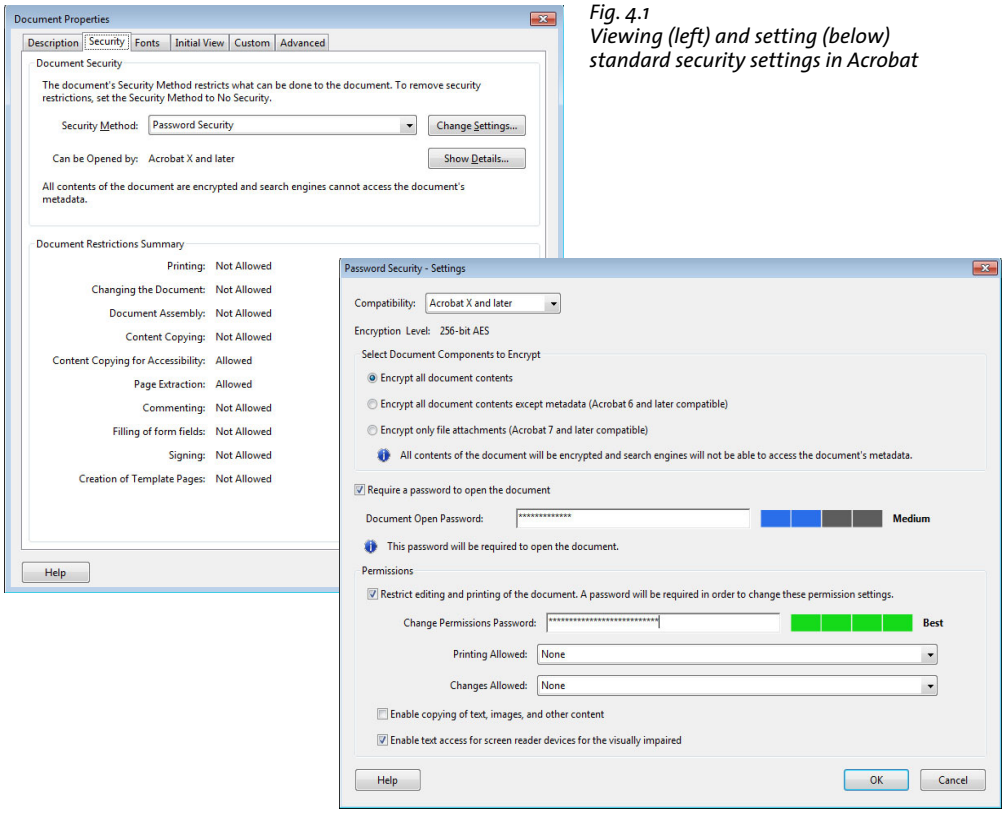

Fig. 4.1
Viewing (left) and setting (below) standard security settings in Acrobat

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

- RC4, a symmetric stream cipher (i.e. the same algorithm can be used to encrypt and decrypt). RC4 is a proprietary algorithm.
- AES (Advanced Encryption Standard) specified in the standard FIPS-197. AES is a modern block cipher which is used in a variety of applications.

Since the actual encryption keys are unwieldy binary sequences, they are derived from more user-friendly passwords which consist of plain characters. In the course of PDF and Acrobat development the PDF encryption methods have been enhanced to use stronger algorithms, longer encryption keys, and more sophisticated passwords. Table 4.1 details encryption, key and password characteristics for all PDF versions.

Table 4.1 Encryption algorithms, key lengths, and passwords in PDF versions

PDF and Acrobat version, pCOS algorithm number	***encryption algorithm and key length***	***max. password length and password encoding***
PDF 1.1 - 1.3 (Acrobat 2-4), algorithm 1	*RC4 40-bit (weak, should not be used)*	*32 characters (Latin-1)*
PDF 1.4 (Acrobat 5), algorithm 2	*RC4 128-bit*	*32 characters (Latin-1)*
PDF 1.5 (Acrobat 6), algorithm 3	*same as PDF 1.4, but different application of encryption method*	*32 characters (Latin-1)*
PDF 1.6 (Acrobat 7) and PDF 1.7 = ISO 32000-1 (Acrobat 8), algorithm 4	*AES-128*	*32 characters (Latin-1)*
PDF 1.7ext3 (Acrobat 9), algorithm 9	*AES-256 with weakness in password handling*	*127 UTF-8 bytes (Unicode)*
PDF 1.7ext8 (Acrobat X) and PDF 2.0 = ISO32000-2, algorithm 11	*AES-256 with improved password handling*	*127 UTF-8 bytes (Unicode)*

PDF encryption doesn't use the user or master password directly for encrypting the document contents, but calculates an encryption key from the password and other parameters including the permission settings. The length of the encryption key used for actually encrypting the document is independent from the length of the password (see Table 4.1).

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

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

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

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

Permission settings. PDF can encode various restrictions on document operations which can be granted or denied individually (some 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 search such documents with Acrobat 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. This setting is declared as deprecated in PDF 2.0; content extraction for accessibility purposes is based on the *Content Copying and Extraction* setting.
- *Document Assembly:* If this is disabled, inserting, deleting or rotating pages, or creating bookmarks and thumbnails is prohibited.

Specifying access restrictions for a document, such as *printing prohibited* will disable the respective function in Acrobat. However, this not necessarily holds true for third-party PDF viewers or other software. It is up to the developer of PDF tools whether or not access permissions will be honored. Indeed, several PDF tools are known to ignore permission settings altogether; commercially available PDF cracking tools can be used to disable all access restrictions. This has nothing to do with cracking the encryption; there is simply no way that a PDF file can make sure it won't be printed while it still remains viewable. This is described as follows in ISO 32000-1:

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

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

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

Security recommendations. The following should be avoided because the resulting encryption is weak and could be cracked:

- Passwords consisting of 1-6 characters should be avoided since they are susceptible to attacks which try all possible passwords (brute-force attack against the password).
- Passwords should not resemble a plain text word since the password would be susceptible to attacks which try all plaintext words (dictionary attack). Passwords should contain non-alphabetic characters. Don't use your spouse's or pet's name, birthday, or other items which are easy to determine.
- 40-bit RC4 according to PDF 1.3 (Acrobat 4) encryption should be avoided since it is susceptible to attacks which try all possible keys (brute-force attack against the encryption key).
- The modern AES algorithm is preferable over the older RC4 algorithm.
- AES-256 according to PDF 1.7ext3 (Acrobat 9) should be avoided because it contains a weakness in the password checking algorithm which facilitates brute-force attacks against the password. For this reason Acrobat X and PLOP 4.1 never use Acrobat 9 encryption for protecting new documents (only for decrypting existing documents).

In summary, AES-256 according to PDF 1.7ext8/PDF 2.0 or AES-128 according to PDF 1.6/1.7 should be used, depending on whether or not Acrobat X is available. Passwords should be longer than 6 characters and should contain non-alphabetic characters.

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

4.2 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. In order to decrypt a document the appropriate master password is required.

Encryption algorithm and key length. The encryption algorithm and key length used to protect a document depends on the PDF version of the generated document, which in turn depends on the PDF version of the input document and the *compatibility* option of *PLOP_create_file( )*. The encryption algorithm will be selected as follows:

- PDF versions 1.3 and older will be pushed to PDF 1.4 if any of the protection options *userpassword*, *masterpassword* or *permissions* is applied. RC4 40-bit will never be used.
- PDF 1.4 and 1.5: the respective flavor of RC4 encryption with 128-bit keys will be used.
- PDF 1.6, PDF 1.7 and PDF 1.7ext3: AES-128 will be used. Note that AES-256 according to PDF 1.7ext3 (Acrobat 9) will never be used due to its known weaknesses.
- PDF 1.7ext8 and PDF 2.0: AES-256 according to Acrobat X will be used.

Since it is widely known that 40-bit encryption keys are not secure PLOP always uses 128-bit keys, and never applies 40-bit keys for encryption. 40-bit-encrypted documents are acceptable as input, however.

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

- Querying the encryption status with the pCOS pseudo object *encrypt/algorithm* etc. is always possible, regardless of any password.
- Querying document properties with the pCOS interface is governed by the pCOS mode. For example, XMP document metadata, document info fields, bookmarks, and annotation contents can be retrieved without the master password if the document does not require a user password (or only the user password has been supplied). The pCOS Path Reference discusses this in more detail.
- 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 (pCOS pseudo object »encrypt«)*	*query document info, XMP metadata, bookmarks, annotation contents with pCOS*	*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 requires either the user or master password depending on the desired operation according to Table 4.2. If the input document could successfully be opened (either because it was unprotected or because the appropriate password has been 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.
- Unicode passwords are allowed for AES-256. All older encryption algorithms require passwords which are restricted to the Latin-1 character set. An exception will be thrown for older encryption algorithms if the supplied password contains characters outside the Latin-1 character set.
- Passwords will be truncated to 127 UTF-8 bytes for AES-256, and to 32 characters for older encryption algorithms.

Setting permissions with 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 a user password, but a master password is required. Table 4.3 lists the supported permission keywords.

Table 4.3 Access restriction keywords for the `permissions` *option of PLOP_create_file()*

keyword	*explanation*
noprint	*Acrobat will prevent printing the file.*
nomodify	*Acrobat will prevent users from adding form fields or making any other changes.*
nocopy	*Acrobat will prevent copying and extracting text or graphics, and will disable accessibility.*
noannots	*Acrobat will prevent adding or changing comments or form fields.*
noforms[1]	*(Implies* `noannots`*) Acrobat will prevent form field filling, even if noannots hasn't been specified.*
noaccessible[1]	*(Deprecated in PDF 2.0) Acrobat will prevent extracting text or graphics for accessibility purposes.*
noassemble[1]	*(Implies* `nomodify`*) Acrobat will prevent inserting, deleting, or rotating pages and creating bookmarks and thumbnails, even if* `nomodify` *hasn't been specified.*
nohiresprint[1]	*Acrobat will prevent high-resolution printing. If* `noprint` *hasn't been specified printing is restricted to the »print as image« feature which prints a low-resolution rendition of the page.*
plainmetadata[2]	*Keep document metadata unencrypted even for encrypted documents.*

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)

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 PDF password security, but not certificate-based encryption, or any third-party encryption or digital rights management systems for PDF (such as Adobe Digital Editions or FileOpen).
- PLOP is not a Digital Rights Management (DRM) system: you can't tie a document to individual computers, users, or CPUs.

4.3 Securing PDF Documents on the Command Line

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 command-line tool is *--permissions*.

Note *On Windows passwords on the command line may contain Unicode characters outside the Latin-1 character set.*

Encryption examples. The sample command-line calls below are shown both with long and abbreviated command-line options.

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 (to follow option list syntax) and with straight quote characters (to follow shell syntax) 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 »Security recommendations«, page 50), not short ones as in this example:

```
plop --compatibility 1.6 --password old --master DEMO --outputfile strong.pdf weak.pdf
plop -c 1.6 -p old -m DEMO -o 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 encryption 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 PDFlib 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 custom requirements regarding certificates or signature generation, and to hook up an existing cryptographic hardware or software module. 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.
- An 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*), the *keytool* application which

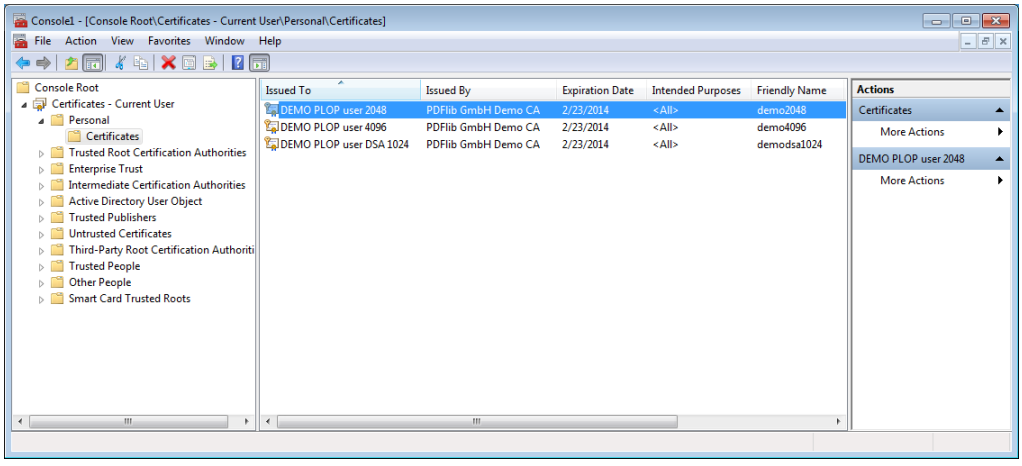

Fig. 5.1
Managing the Windows certificate store with the management console MMC.

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:
  Acrobat X: *Tools, Sign&Certify, More Sign & Certify, Security Settings, Digital IDs, Add ID, A new digital ID I want to create now;*
  Acrobat 9: *Advanced, Security Settings, Digital IDs, Add ID, A new digital ID I want to create now;*
  Acrobat 8: *Advanced, Security Settings, Digital IDs, Add ID, create a self-signed digital ID for use with Acrobat*
  In the next step you can specify a PKCS#12 disk file or the Windows certificate store as target. Both methods are supported in PLOP DS.

Managing the Windows certificate store. The Windows operating system 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 Import Wizard. You can try this with the demo certificates in the PLOP DS package, using the password *demo*

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...*
- 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 *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( )):*

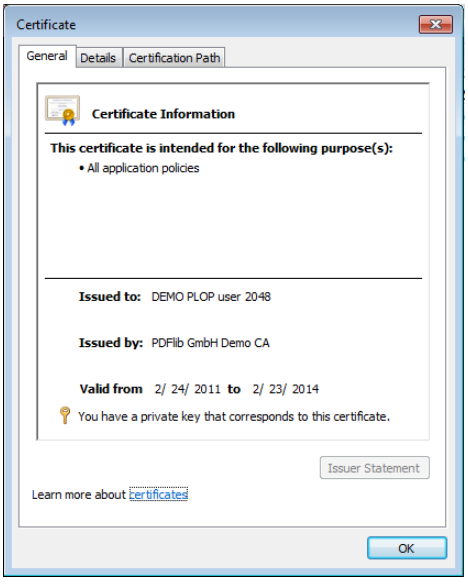

Fig. 5.2
Certificate properties in Windows

```
engine=mscapi digitalid={certstore={store=My subject={Demo PLOP User 2048}}}
```

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...* . This will launch the Certificate Export Wizard.

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

Windows supports the following key lengths:

- RSA algorithm: 384 bit to 16384 bit with the Microsoft Enhanced Cryptographic Provider, and 384 bits to 512 bits with the Microsoft Base Cryptographic Provider.
- DSA algorithm: 512 bit to 1024 bit (unlike Acrobat which supports DSA up to 4096 bit)

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 *(Acrobat X: View, Show/Hide..., Navigation Panes, Signatures...; Acrobat 8/9: 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 58). 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 *demo2048.p12* and *demo2048.pfx* with the password *demo*, and digital IDs in the Windows certificate store for the hypothetical user name *DEMO PLOP user 2048*. Sample digital ID files are included in the distribution packages.

Create an invisible signature for a PDF document, using a digital ID from the file *demo2048.p12*. The password for the digital ID is contained in the file *pw.txt:*

```
plop --signopt "digitalid={filename=demo2048.p12} passwordfile=pw.txt" ↵
       --outfile signed.pdf input.pdf
plop -S "digitalid={filename=demo2048.p12} passwordfile=pw.txt" -o signed.pdf input.pdf
```

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=demo2048.p12} password={demo}" --outfile signed.pdf input.pdf
plop -S "appearance={fieldname=Signature1 rect={10 10 200 100} } ↵
       digitalid={filename=demo2048.p12} password={demo}" -o signed.pdf input.pdf
```

Note You must double-click and install the digital ID in the file `demo2048.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=mscapi digitalid={certstore={store=My subject={Demo PLOP User 2048}}}" ↵
        --outfile signed.pdf input.pdf
```

```
plop -S "engine=mscapi digitalid={certstore={store=My subject={Demo PLOP User 2048}}}" ↵
         -o signed.pdf input.pdf
```

(Windows only) Create an invisible signature for a PDF document, using a certificate in the file *demo2048.pfx:*

```
plop --signopt "engine=mscapi digitalid={filename=demo2048.pfx} passwordfile=pw.txt" ↵
        --outfile signed.pdf input.pdf
plop --S "engine=mscapi digitalid={filename=demo2048.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=demo2048.p12} password={demo}" ↵
        --outfile signed.pdf input.pdf
plop --m SECRET --S "digitalid={filename=demo2048.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
```

Selecting a digital ID on a token. A smartcard or other cryptographic token may contain multiple digital IDs, e.g. one for encrypting E-mails and another one for digitally signing documents. In this situation you can select the target ID for the PLOP DS signature with the *keyusage* suboption of the *sign* option of *PLOP_create_file()*. It accepts a keyword which selects a digital ID based on the usage flags which are encoded in the certificate's *KeyUsage* extension (see RFC 3280 for a detailed description of the *KeyUsage* extension). For example, if a smartcard contains two IDs where the ID with the *nonrepudiation* flag must be used for signing, the following suboption list for the *sign* option can be used:

```
digitalid={filename=cryptoki.dll keyusage={nonrepudiation=set}}
```

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:Y*ou 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.

5.4 Cryptographic Properties of PLOP DS Signatures

Encryption algorithm and key length for signatures. The encryption algorithm and key length for generating signatures 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
- DSA with up to 4096 bit key length; DSA only supports the SHA-1 message digest.

Message digest (hash function). The message digest algorithm (hash function) used for creating digital signatures depends on several factors. PLOP DS tries to use the secure SHA-256 algorithm if possible, otherwise SHA-1.

When selecting SHA-256 or SHA-1 two aspects are relevant: the availability of a SHA-256 implementation and the kind of generated PDF output. Availability of SHA-256 depends on the selected crypto engine:

- SHA-256 is always available for *engine=builtin*.
- SHA-256 may or may not be available for *engine=pkcs#11* depending on the capabilities of the cryptographic token. Refer to the token's documentation or contact the token vendor for information regarding the token's cryptographic features.
- SHA-256 availability for *engine=mscapi* depends on the Windows version: according to Microsoft documentation it should be available with Windows XP SP3 and newer. However, there are cases where SHA-256 is not available in XP SP3.

If SHA-256 is available it will only be used if the generated PDF output meets one of the following requirements:

- The PDF output version determined based on the PDF input version, requested operations and *compatibility=1.6* or above.
- PDF/A-1 output is being generated. This is a special exception to the PDF 1.6 rule above which allows secure digital signatures of PDF/A documents.

If the SHA-256 algorithm is not available or the generated PDF output does not meet the conditions listed above, the SHA-1 algorithm will be used instead. PLOP DS never uses the MD5 algorithm for signatures since it is not considered secure enough.

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.

5.5 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 8 or above and Adobe Reader 8 or above. 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 uses different icons for individual signatures as well as for the signature status of a document:

- A check mark  indicates that the signature is valid, i.e. the signer has been verified and the document has not changed.
- A red X  indicates that the signature is invalid, i.e. the signer's certificate could not be verified (e.g. it was expired or revoked), or the document has been changed.
- A triangle  indicates that the document is problematic because the signer's identity could not be verified or document updates have been applied after the signature was applied.

Acrobat 8 also displays the signature status in the field area of a visible signature. However, Acrobat 9/X no longer display 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\10.0\Security\cPubSec\iDisplayValidIcon
```

to the value 0 (digit zero). This will restore Acrobat 8 behavior in Acrobat 9/X. 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.

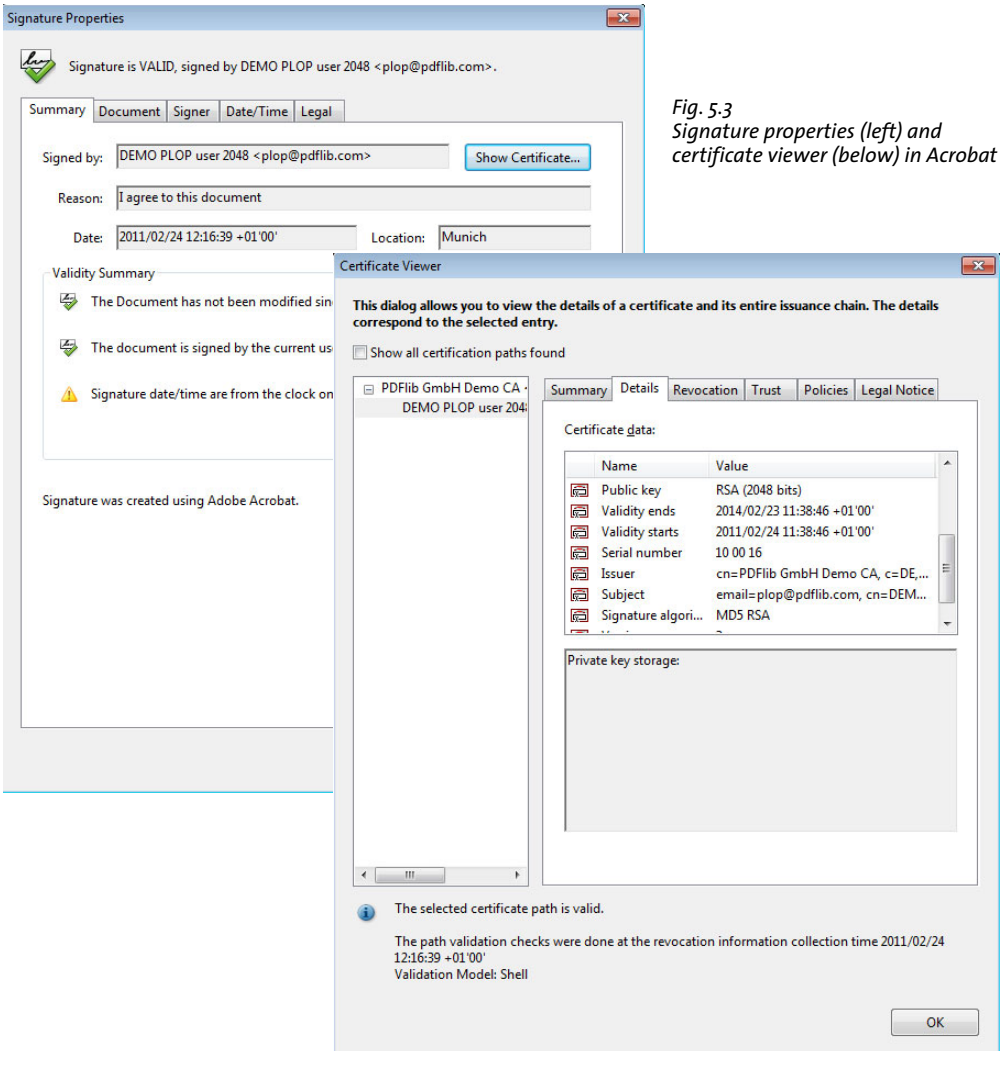

Fig. 5.3
Signature properties (left) and certificate viewer (below) in Acrobat

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 59), 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 8/9/X proceed as follows (see Figure 5.4): *Edit, Preferences, [General...], Security, Advanced Preferences..., Windows Integration*. 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*.

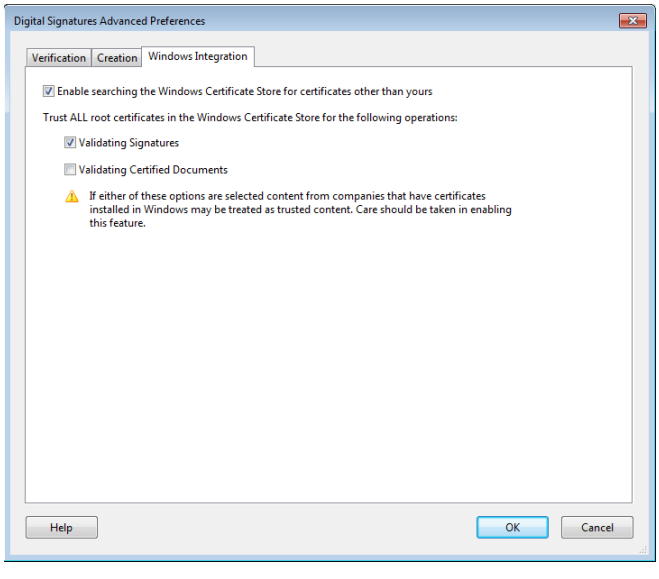

Fig. 5.4
Configure Acrobat to access the Windows Certificate Store

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 X this can be achieved with the following steps: *Tools, Sign & Certify, More Sign & Certify, Manage Trusted Identities..., Add Contacts, Browse...* .
  Acrobat 8/9: *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/X this can be achieved with the following steps: open a signed PDF, open the *Signatures* tab (Acrobat X: *View, Show/Hide, Navigation Panes, Signatures),* right-click on the signature and select *[Show Signature] Properties*; Acrobat 8/9: *View, Navigation Panels, Signatures)* 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 and/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 *Use this certificate as a trusted root* (Acrobat 9/X) or *Add to Trusted Identities...,* and finally check *Signatures and as a trusted root* (Acrobat 8).

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. Examples for using the pCOS interface and a description of the pCOS path syntax are contained in the pCOS Path Reference which is available as a separate document. Additional examples can be found in the pCOS Cookbook at *www.pdflib.com/pcos-cookbook/*

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 *noname* 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 i5/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 i5/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

C ***PLOP * PLOP_new(void)***

Create a new PLOP context.

Returns A handle to the new context, or NULL if not enough memory is available. The context must be supplied to all other API functions.

Bindings Not available in object-oriented language bindings where it will be called automatically when a new PLOP object is created.

Java ***void delete( )***
C# ***void Dispose( )***
C ***void PLOP_delete(PLOP *plop)***

Delete a PLOP context and release all its internal resources.

Details All open documents in the context are closed automatically. It is good programming practice, however, to close documents explicitly with *PLOP_close_document( )* when they are no longer needed.

Bindings In C this function must not be called within a *PLOP_TRY( )/PLOP_CATCH( )* clause.

In Java this method will be called by the finalizer method of PLOP. However, it is strongly recommended to explicitly call *delete( )* for proper cleanup. The same holds true when an exception occurred.

In Perl, PHP and COM this function will be called automatically when the PLOP object is destroyed.

In .NET *Dispose( )* should be called at the end of processing to clean up unmanaged resources.

C++ ***void create_pvf(string filename, const void *data, size_t size, string optlist)***
C# Java ***void create_pvf(String filename, byte[ ] data, String optlist)***
Perl PHP ***create_pvf(string filename, string data, string optlist)***
VB ***Sub create_pvf(filename As String, data, optlist As String)***
C ***void PLOP_create_pvf(PLOP *plop,***
const char *filename, int len, const void *data, size_t size, const char *optlist)

Create a named virtual read-only file from data provided in memory.

filename (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.

len (C language binding only) Length of *filename* (in bytes) for UTF-16 strings. If *len=0* 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 This function may be useful for repeatedly used digital IDs or XMP metadata. 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
copy	*(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 delete_pvf(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 (in PHP: 0) 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.

C++ ***double info_pvf(string filename, string keyword)***

C# Java ***double info_pvf(String filename, String keyword)***

Perl PHP ***float info_pvf(string filename, string keyword)***

VB ***Function info_pvf(filename As String, keyword As String) As Double***

C ***double PLOP_info_pvf(PDF *p, const char *filename, int len, const char *keyword)***

Query properties of a virtual file or the PDFlib Virtual File system (PVF).

filename (Name string) The name of the virtual file. The filename may be empty if *keyword=filecount*.

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

keyword A keyword according to Table 7.4.

Table 7.2 Keywords for PLOP_info_pvf( )

option	description
filecount	*Total number of files in the PDFlib Virtual File system maintained for the current PLOP object. The* `filename` *parameter will be ignored.*
exists	*1 if the file exists in the PDFlib Virtual File system (and has not been deleted), otherwise 0*
size	*(Only for existing virtual files) Size of the specified virtual file in bytes.*
iscopy	*(Only for existing virtual files) 1 if the* `copy` *option was supplied when the specified virtual file was created, otherwise 0*
lockcount	*(Only for existing virtual files) Number of locks for the specified virtual file set internally be PLOP functions. The file can only be deleted if the lock count is 0.*

Details This function returns various properties of a virtual file or the PDFlib Virtual File system (PVF). The property is specified by *keyword*.

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++ ***int open_document(string filename, string optlist)***

C# Java ***int open_document(String filename, String optlist)***

Perl PHP ***int open_document(string filename, string optlist)***

VB ***Function open_document(filename As String, optlist As String) As Long***

C ***int PLOP_open_document(PLOP *plop, const char *filename, int len, const char *optlist)***

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 71) 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 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 7.3 Options for PLOP_open_document( )*

option	*description*
inmemory	*(Boolean; only for PLOP_open_document( )) 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`
password	*(String; required for encrypted documents except with* `requiredmode`*) The user or master password for the document. As detailed in Table 4.2, page 51, 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* `ebcdic` *encoding or EBCDIC-UTF-8.*
repair	*(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`*):* ***force*** *Unconditionally try to repair the document, regardless of whether or not it has problems.* ***auto*** *Repair the document only if problems are detected while opening the PDF.* ***none*** *No attempt will be made at repairing the document. If there are problems in the PDF the function call will fail.*

Table 7.3 Options for PLOP_open_document()*

option	description
requiredmode	*(Keyword) The minimum pcos mode* (`minimum`/`restricted`/`full`) *which is acceptable when opening the document. The call will fail if the resulting pCOS mode would be lower than the required mode. If the call succeeds it is guaranteed that the resulting pCOS mode is at least the one specified in this option. However, it may be higher; e.g.* `requiredmode=minimum` *for an unencrypted document will result in* `full` *mode. Default:* `full`
xmppolicy	*(Keyword) Control treatment of invalid document-level XMP in the input document. Invalid XMP implies that no standard identifier can be found, e.g. PDF/A documents will not be treated as such. Supported keywords (default:* `rejectinvalid`*):* ***rejectinvalid*** *Throw an exception for invalid XMP which includes the XML parsing error message, and stop processing.* ***ignoreinvalid*** *(Implies* `sacrifice={pdfa1 pdfx}`*) Treat invalid XMP as if there was no XMP present. Output XMP will be generated based on document info entries; it will also include the XML parsing error message in the* `<pdfx:invalid_source_XMP_exception>` *element.* ***remove*** *Unconditionally ignore input XMP, regardless of its validity. The output XMP will be generated from scratch. This may be useful to delete unwanted metadata. However, standard identifiers (e.g. for PDF/A) will still be read from the input XMP and copied to the output.*

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 71) 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.

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

C++ ***int create_file(string filename, string optlist)***

C# Java ***int create_file(String filename, String optlist)***

Perl PHP ***int create_file(string filename, string optlist)***

VB ***Function create_file(filename As String, optlist As String) As Long***

C ***int PLOP_create_file(PLOP *plop, const char *filename, int len, const char *optlist)***

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 71) according to Table 7.4.

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 4.2, »PDF Security Features in PLOP«, page 51, for conditions which will be enforced for the user and master passwords.

The document will be encrypted if any of the *userpassword, masterpassword*, or *permissions* options has been supplied. The encryption algorithm will be selected based on the PDF version of the input document and the *compatibility* option (see »Encryption algorithm and key length«, page 51).

Table 7.4 Options for PLOP_create_file( )

option	description
compatibility	*(Keyword) Specify the PDF version of the generated PDF output document:* ***1.4*** *PDF 1.4 requires Acrobat 5 or above.* ***1.5*** *PDF 1.5 requires Acrobat 6 or above.* ***1.6*** *PDF 1.6 requires Acrobat 7 or above.* ***1.7*** *PDF 1.7 is specified in ISO 32000-1 and requires Acrobat 8 or above.* ***1.7ext3*** *PDF 1.7 extension level 3 requires Acrobat 9 or above.* ***1.7ext8*** *PDF 1.7 extension level 8 requires Acrobat X.* ***2.0*** *PDF 2.0 is specified in ISO 32000-2.* *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:* ▸ *Digitally signing the document (option* `sign`*) pushes the version to PDF 1.3.* ▸ *Encryption, i.e. any of the options* `userpassword`*,* `masterpassword`*, and* `permissions`*, pushes the version to PDF 1.4.* ▸ *Inserting XMP metadata (option* `metadata`*) pushes the version to PDF 1.4.* ▸ *The* `plainmetadata` *keyword for the* `permissions` *option pushes the version to PDF 1.5.* *Default: the PDF version of the input document, or a higher version as mandated by the rules above.*
docinfo	*(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):* ***Subject*** *Subject of the document* ***Title*** *Title of the document* ***Author*** *Author of the document* ***Keywords*** *Keywords describing the contents of the document* ***Trapped*** *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`*.* ***any name other than Creator, CreationDate, Producer, ModDate, GTS_PDFXVersion, GTS_PDFXConformance, ISO_PDFEVersion*** *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.*
flush	*(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 PLOP_get_buffer( ). When the* `linearize` *option is* `true`*, the flushing strategy must be* `none`*. (Default:* `none`*):* ***none*** *The returned buffer is guaranteed to contain all data comprising the output document.* ***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 parts of the output document.* ***heavy*** *PLOP_get_buffer( ) will process smaller portions, and will be called more frequently.*
linearize	*(Boolean; can not be combined with* `sign` *or* `metadata`*) 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`
masterpassword[1]	*(String) 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 or EBCDIC-UTF-8. Default: empty*

Table 7.4 Options for PLOP_create_file( )

option	description
metadata	*(Option list; can not be combined with* `linearize`*) Supply XMP metadata for the document. PDF/A-1 and PDF/X identification entries are not allowed in the supplied XMP. The option list may contain the following options:* ***filename*** *(Name string; required) The name of a file containing well-formed XMP metadata in UTF-8 format.* ***validate*** *(Keyword) The XMP metadata will be validated according to the keyword:* ***none*** *No validation* ***xmp2004*** *Validation according to the XMP 2004 specification* ***pdfa1*** *Like* `xmp2004`*, plus testing for predefined properties and schemas, and extension schema validation according to PDF/A-1.* *Default:* none*, but will be forced to* `pdfa1` *if the input conforms to PDF/A-1 and the* `sacrifice` *option does not include* `pdfa1`
optimize	*(Keyword) The optimization steps to be applied while processing the document (default:* none *if the* `sign` *option was supplied, otherwise* `all`*):* ***all*** *Apply all implemented optimizations.* ***none*** *Don't apply any optimization; this will slightly speed up processing at the expense of file size.*
permissions	*(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 52). Default: empty*
recordsize	*(Integer; MVS only) The record size of the output file. Default: 0 (unblocked output)*
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, but throw an exception instead. However, you can sacrifice some property of the document in order to allow processing. The keywords listed in Table 7.5. 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):*
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 7.6.*
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*
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*
user-password[1]	*(String; 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 or EBCDIC-UTF-8. Default: empty*

1. Characters outside of Winansi encoding are only allowed in passwords if PDF 1.7 extension level 3 or higher is generated.

Table 7.5 Suboptions for the `sacrifice` *option of PLOP_create_file( )*

option	*description*
encrypted-attachments	*(Input trigger: the document is not encrypted, but contains one ore 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.*
fields	*(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.*
pdfa1	*(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).*
pdfx	*(Input trigger: the document conforms to PDF/X-1a or PDF/X-3/4/5; 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).*
signatures	*(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.*

Table 7.6 Suboptions for the sign *option of PLOP_create_file( ) (only available in PLOP DS)*

option	description
appearance	*(Option list) Specifies the visual appearance of the form field which will hold the signature:*
	fieldname *(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
	page *(Integer) Number of the page on which the (visible or invisible) signature field will be created. The first page has number 1. Default: 1*
	rect *(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*
contactinfo	*(Text string) Information provided by the signer to enable a recipient to contact the signer to verify the signature (e.g. a phone number)*
digitalid	*(Option list; required) Specifies the signer's digital ID (certificate and private key) with exactly one of the following suboptions:*
	filename *(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 PLOP_create_pvf( ). For* engine=pkcs#11 *this option contains the name of a PKCS#11 DLL/shared library for the cryptographic token, e.g. smartcard.*
	certstore *(Option list; only for* engine=mscapi*) Options for locating an ID in Windows' certificate store:*
	subject *(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.*
	store *(String) Name of the certificate store (common names:* My, root, trust, CA*). Default:* My
	keyusage *(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 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. The following keywords are supported:* clear, ignore, set. *The default is* ignore *for all entries.*
	digitalsignature *(Keyword) One of the keywords* clear/ignore/set *to specify handling of the* digitalsignature *key usage extension (i.e. bit 0).*
	nonrepudiation *(Keyword) One of the keywords* clear/ignore/set *to specify handling of the* nonrepudiation *key usage extension (i.e. bit 1).*
engine	*(Keyword) Specifies the crypto engine to be used for digital signatures (default:* builtin*):*
	builtin *Use the built-in crypto engine; digital IDs must be fetched from a disk file (PFX or PKCS#12).*
	mscapi *(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).*
	pkcs#11 *(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.*
location	*(Text string) Physical location of the signing*
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.*

Table 7.6 Suboptions for the `sign` *option of PLOP_create_file( ) (only available in PLOP DS)*

option	*description*
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*
subfilter	*(Keyword) Kind of PDF signature (default:* `adbe.pkcs7.detached`*):* ***adbe.pkcs7.detached*** *No data will be encapsulated in the PKCS#7 signed-data field. This method covers dynamic document modifications, e.g. a date field populated by JavaScript code.* ***adbe.pkcs7.sha1*** *The SHA-1 digest of the data is encapsulated in the PKCS#7 signed-data field. This method will not cover dynamic document modifications.*

C++ ***const char *get_buffer(long *size)***

C# Java ***byte[ ] get_buffer( )***

Perl PHP ***string get_buffer( )***

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.

C++ ***void close_document(int doc)***

C# Java ***close_document(int doc)***

Perl PHP ***close_document(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 *PLOPException* object which offers several members to access detailed error information.

C++ ***int get_errnum( )***
C# Java ***int get_errnum( )***
Perl PHP ***int get_errnum( )***
VB ***Function get_errnum( ) As Long***
C ***int PLOP_get_errnum(PLOP *plop)***

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 *PLOPException* object.
In Java this method is also available as *get_errnum( )* in the *PLOPException* object.

C++ ***string get_errmsg( )***
C# Java ***String get_errmsg( )***
Perl PHP ***string get_errmsg( )***
VB ***Function get_errmsg( ) As String***
C ***const char *PLOP_get_errmsg(PLOP *plop)***

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 *PLOPException* object.
In Java this method is also available as *getMessage( )* in the *PLOPException* object.

C++ ***string get_apiname( )***
C# Java ***String get_apiname( )***
Perl PHP ***string get_apiname( )***
VB ***Function get_apiname( ) As String***
C ***const char *PLOP_get_apiname(PLOP *plop)***

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 *PLOPException* object.
In Java this method is also available as *get_apiname( )* in the *PLOPException* object.

C *PLOP_TRY(PLOP *plop)*

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

Details See »Exception handling«, page 31.

C *PLOP_CATCH(PLOP *plop)*

Catch an exception; must always be paired with *PLOP_TRY( )*.

Details See »Exception handling«, page 31.

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«, page 31.

C *PLOP_RETHROW(PLOP *plop)*

Re-throw an exception to another handler.

Details See »Exception handling«, page 31.

7.5 Option Handling

C++ ***void set_option(string optlist)***

C# Java ***void set_option(String optlist)***

Perl PHP ***set_option(string optlist)***

VB ***Sub set_option(optlist As String)***

C ***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 7.7. 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 7.7. For unmarked options the new value will override the old one.

Table 7.7 Global options for PLOP_set_option( )

option	*description*
filename-handling	*(Keyword; not required on Windows) Target encoding for file names. On Windows this option will be applied to supplied file names, but not to the names of generated files (default:* `unicode` *on Mac OS X, otherwise* `honorlang`*):* ***ascii*** *7-bit ASCII* ***basicebcdic*** *Basic EBCDIC according to code page 1047, but only Unicode values <= U+007E* ***basicebcdic_37*** *Basic EBCDIC according to code page 0037, but only Unicode values <= U+007E* ***honorlang*** *The environment variables LC_ALL, LC_CTYPE and LANG will be interpreted and applied to file names if it specifies* `utf8`, `UTF-8`, `cpXXXX`, `CPXXXX`, `iso8859-x`, *or* `ISO-8859-x`. ***legacy*** *Use* `auto` *encoding (i.e. the current system encoding) to interpret the file name and interpret the LANG variable if the* `honorlang` *parameter is set.* ***unicode*** *Unicode encoding in (EBCDIC-) UTF-8 format* ***all valid encoding names*** *Any (internal or user-defined) encoding recognized by PLOP* *File names supplied in non-Unicode aware language bindings without a UTF-8 BOM and with length=0 will be interpreted according to the* `filenamehandling` *option.*
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 PDFLIBLICENSEFILE 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[1]	*(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*

Table 7.7 Global options for PLOP_set_option( )

option	description	
shutdown-strategy	*(Integer) Strategy for releasing global resources which are allocated once for all PLOP objects. Each global resource is initialized on demand when it is first needed. This option must be set to the same value for all PLOP objects in a process; otherwise the behavior is undefined (default: 0):*	
	0	*A reference counter keeps track of how many PLOP objects use the resource. When the last PLOP object is deleted and the reference counter drops to zero, the resource is released.*
	1	*The resource is kept until the end of the process. This may slightly improve performance, but requires more memory after the last PLOP object is deleted.*

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 the pCOS Path Reference 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 pcos_get_number(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 pcos_get_string(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*. 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.

Java and .NET: the result will be provided as Unicode string. If no more text is available a null object will be returned.

Perl and PHP language bindings: the result will be provided as UTF-8 string. If no more text is available a null object will be returned.

RPG language binding: the result will be provided as UTF-8 string.

C++ ***const unsigned char *pcos_get_stream(int doc, int *length, string optlist, string path)***

C# Java ***byte[ ] pcos_get_stream(int doc, String optlist, String path)***

Perl PHP ***string pcos_get_stream(long doc, string optlist, string path)***

VB ***Function pcos_get_stream(doc as Long, optlist As String, path As String)***

C ***const unsigned char *PLOP_pcos_get_stream(PLOP *plop, int doc, int *length, const char *optlist, const char *path, ...)***

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 7.8.

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 ASCIIHex filters which will be removed.

Details This function will throw an exception if pCOS does not run in full mode. 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 7.8 Options for PLOP_pcos_get_stream( )

option	description
convert	*(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*)*:
	none *Treat the contents as binary data without any conversion.*
	unicode *Treat the contents as textual data (i.e. exactly as in PLOP_pcos_get_string( )), and normalize it to Unicode. In non-Unicode-aware language bindings this means the data will be converted to UTF-8 format without BOM.* *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).*

7.7 Unicode Conversion Function

C++ ***string convert_to_unicode(string inputformat, string input, string optlist)***

C# Java ***string convert_to_unicode(string inputformat, byte[ ] input, string optlist)***

Perl PHP ***string convert_to_unicode(string inputformat, string input, string optlist)***

VB ***Function convert_to_unicode(inputformat as String, input, optlist as String) As String***

C ***const char *PLOP_convert_to_unicode(PLOP *p,***
const char *inputformat, const char *input, int inputlen, int *outputlen, const char *optlist)

Convert a string in an arbitrary encoding to a Unicode string in various formats.

inputformat Unicode text format or encoding name specifying interpretation of the input string:

- Unicode text formats: *utf8, ebcdicutf8, utf16, utf16le, utf16be, utf32*
- All internally known 8-bit encodings, encodings available on the host system, and the CJK encodings *cp932, cp936, cp949, cp950*
- The keyword *auto* specifies the following behavior: if the input string contains a UTF-8 or UTF-16 BOM it will be used to determine the appropriate format, otherwise the current system codepage will be assumed.

input String (COM: Variant) to be converted to Unicode.

inputlen (C language binding only) Length of the input string in bytes. If *inputlen = 0* a null-terminated string must be provided.

outputlen (C language binding only) C-style pointer to a memory location where the length of the returned string (in bytes) will be stored.

optlist An option list specifying options for input interpretation and Unicode conversion:

- Input filter options according to Table 7.9: *charref, escapesequence*
- Unicode conversion options according to Table 7.9: *bom, errorpolicy, inflate, outputformat*

Returns A Unicode string created from the input string according to the specified parameters and options. If the input string does not conform to the specified input format (e.g. invalid UTF-8 string) an empty output string will be returned if *errorpolicy=return*, and an exception will be thrown if *errorpolicy=exception*.

Details This function may be useful for general Unicode string conversion. It is provided for the benefit of users working in environments which do not provide suitable Unicode converters.

Scope *any*

Bindings C binding: the returned strings will be stored in a ring buffer with up to 10 entries. If more than 10 strings are converted, the buffers will be reused, which means that clients must copy the strings if they want to access more than 10 strings in parallel. For example, up to 10 calls to this function can be used as parameters for a *printf( )* statement since the return strings are guaranteed to be independent if no more than 10 strings are used at the same time.

Table 7.9 Options for PLOP_convert_to_unicode( )

option	description
bom	*(Keyword; will be ignored for* `outputformat=utf32`*) Policy for adding a byte order mark (BOM) to the output string. Supported keywords (default:* `none`*):* ***add*** *Add a BOM.* ***keep*** *Add a BOM if the input string has a BOM.* ***none*** *Don't add a BOM.* ***optimize*** *Add a BOM except if* `outputformat=utf8` *or* `ebcdicutf8` *and the output string contains only characters in the range < U+007F.*
charref	*(Boolean) If* `true`*, enable substitution of numeric and character entity references and glyph name references. Default:* `false`
errorpolicy	*(Keyword) Behavior in case of conversion errors (default:* `exception`*):* ***return*** *The replacement character will be used if a character reference cannot be resolved. An empty string will be returned in case of conversion errors.* ***exception*** *An exception will be thrown in case of conversion errors.*
escapesequence	*(Boolean) If* `true`*, enable substitution of escape sequences in strings. Default:* `false`
inflate	*(Boolean; only for* `inputformat=utf8`*; will be ignored if* `outputformat=utf8`*) If* `true`*, an invalid UTF-8 input string will not trigger an exception, but rather an inflated byte string in the specified output format will be generated. This may be useful for debugging. Default:* `false`
outputformat	*(Keyword) Unicode text format of the generated string:* `utf8`, `ebcdicutf8`, `utf16`, `utf16le`, `utf16be`, `utf32`*. An empty string is equivalent to* `utf16`*. Default:* `utf16` *Unicode-aware language bindings: the output format will be forced to* `utf16`*.* *C++ language binding: only the following output formats are allowed:* `utf8`, `utf16`, `utf32`.

C++ ***string utf16_to_utf8(string utf16string)***
Perl PHP ***string utf16_to_utf8(string utf16string)***
C ***const char *PLOP_utf16_to_utf8(PLOP *p, const char *utf16string, int len, int *size)***

Deprecated, use *PLOP_convert_to_unicode( )*

C++ ***string utf8_to_utf16(string utf8string, string ordering)***
Perl PHP ***string utf8_to_utf16(string utf8string, string ordering)***
C ***const char *PLOP_utf8_to_utf16(PLOP *p, const char *utf8string, const char *ordering, int *size)***

Deprecated, use *PLOP_convert_to_unicode( )*

A Combining PDFlib with PLOP or 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/PDFlib+PDI/PPS 5	yes	–	–	–
PDFlib/PDFlib+PDI/PPS 6	yes	yes	–	–
PDFlib/PDFlib+PDI/PPS 7, 8	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 7 and 8 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 create a virtual file with *PLOP_create_pvf( )* . The file name used for the virtual file can then be passed to PLOP/PLOP DS using *PLOP_open_document( )* without having to create a physical file on disk. 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

Document Input and Output

Error Handling

Option Handling

pCOS Functions

Unicode Conversion Function

C Revision History

Revision history of this manual

Date	*Changes*
March 04, 2011	► *Major overhaul for PLOP 4.1 and PLOP DS 4.1*
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

P

R

S

T

U

V

W

X

PDFlib GmbH

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

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

Licensing contact
sales@pdflib.com

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