PDF Prep Tool Suite

Version 4.12

User Manual

Contact: pdfsupport@pdf-tools.com

Owner: **PDF Tools AG**

Kasernenstrasse 1
8184 Bachenbülach
Switzerland

http://www.pdf-tools.com

Table of Contents

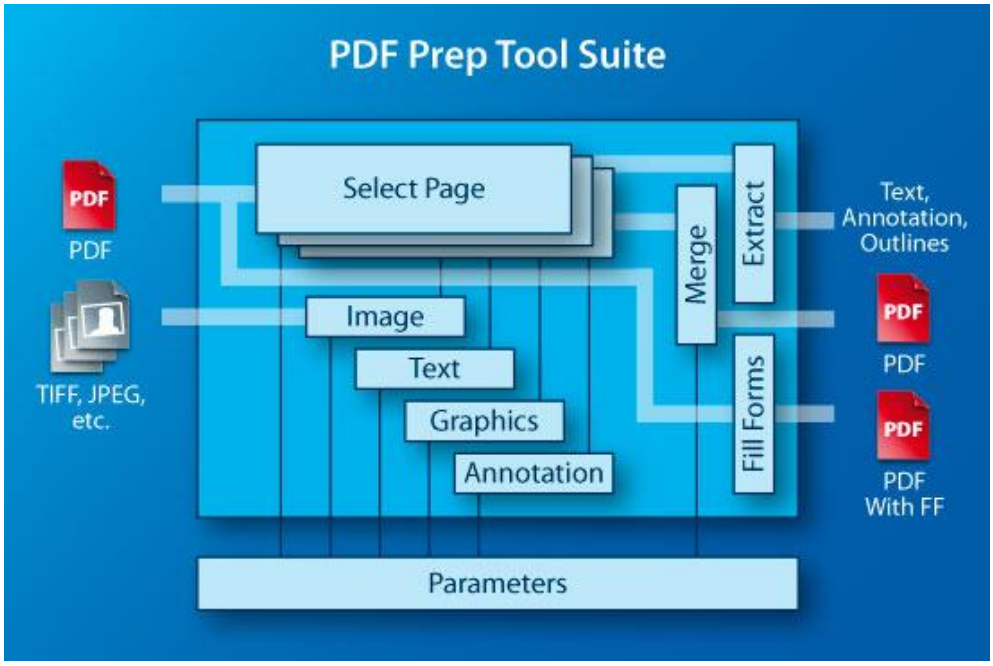
1 Introduction

The PDF Prep Tool Suite is a programming library for creating, splitting and merging PDF documents. It can be used to add content such as text, images and vector graphics. Interactive elements such as links, form fields and bookmarks can be added and processed. The component is used for the following tasks:

- Assemble PDF documents
- Personalize documents
- Fill in form fields

PDF documents can be created from scratch – for instance on the basis of a template to which data is added from a source such as a database.

Properties such as position, font, size and color are freely selectable. Once created, PDF documents can be encrypted and optimized for fast web-based viewing.



1.1 Functions

- Merge any number of pages from one or multiple PDF documents
- Apply content to the background or foreground of an existing or new page
- Text (page number, address, customer number, etc.)
- Image (company logo, scanned signature)
- Vector graphic (line, square, curve)
- Extract text including font and positioning information
- Add, fill in, delete and read out form fields

- Add internal and external links and comments
- Copy content from multiple pages to one page including positioning, scaling and rotation
- Set and get outlines (bookmarks) in PDF documents
- Define and extract document properties such as title, author, date of creation, etc.
- Read encrypted PDF documents
- Encrypt PDF documents with a password and set permission flags
- Optimize PDF files for fast web view (linearization)
- Set color as RGB or CMYK
- Set page size (media box) and visible area (crop box)
- Remove viewer access rights

1.2 SDK

The PDF Prep Tool Suite constitutes a specialized module based on the PDF Library SDK. It facilitates the generation of PDF documents based on existing PDF files or parts thereof, controlled by a simple API. It is also possible to create pages via API calls, and to add header or footer text onto pages from input files.

To facilitate the use with Microsoft Visual Basic, a COM interface is available on Windows platforms. Java applications can make use of the component via a Java interface based on JNI via a Java API package.

This document is not an introduction to PDF. You will need to refer to ISO 32000 or an Adobe PDF specification as a complementary source of information.

1.3 Installation

The PDF Prep Tool Suite package comes as compressed archive file (ZIP on Windows, tar.gz on Unix platforms). Extract the contents to the file system. You will find a bin subfolder containing another subfolder named Win32, x86 and/or x64 containing the executable images (32 and/or 64 bit). The Windows kit's bin folder also contains the "Any CPU" .NET assemblies for the Image to PDF Converter API.

You may copy the binaries to a suitable location on the computer. If COM interfaces shall be used on Windows, make sure to register the DLLs using the REGSVR32 tool (as Administrator).

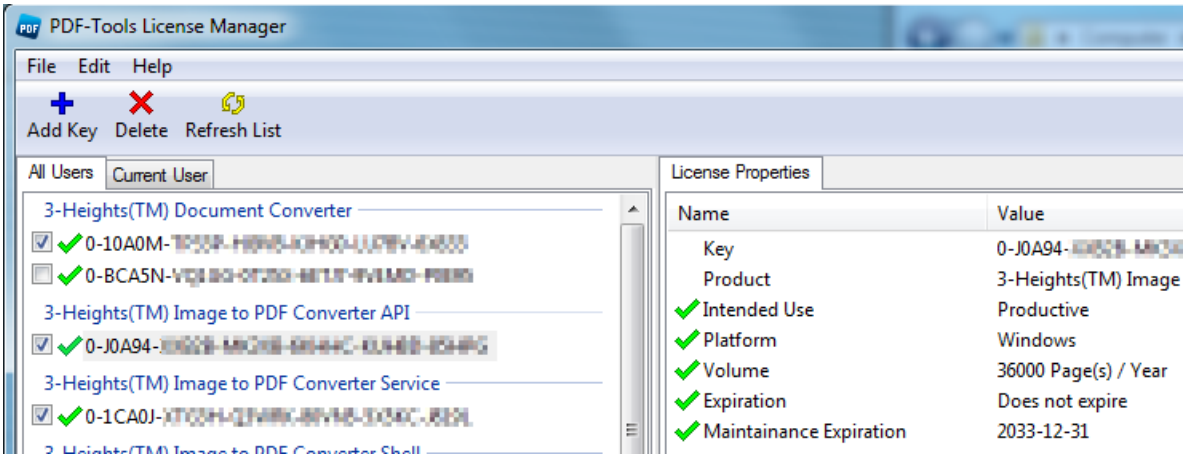
2 License Management

There are three possibilities to pass the license key to the application:

1. The license key is installed using the GUI tool (Graphical user interface). This is the easiest way if the licenses are managed manually. It is only available on Windows.
2. The license key is installed using the shell tool. This is the preferred solution for all non-Windows systems and for automated license management.
3. The license key is passed to the application at runtime via the "LicenseKey" property. This is the preferred solution for OEM scenarios.

2.1 Graphical License Manager Tool

The GUI tool *LicenseManager.exe* is located in the *bin* directory of the product kit.



List all installed license keys

The license manager always shows a list of all installed license keys in the left pane of the window. This includes licenses of other PDF Tools products.

The user can choose between:

- Licenses available for all users. Administrator rights are needed for modifications.
- Licenses available for the current user only.

Add and delete license keys

License keys can be added or deleted with the "Add Key" and "Delete" buttons in the toolbar.

- The "Add key" button installs the license key into the currently selected list.
- The "Delete" button deletes the currently selected license keys.

Display the properties of a license

If a license is selected in the license list, its properties are displayed in the right pane of the window.

Select between different license keys for a single product

More than one license key can be installed for a specific product. The checkbox on the left side in the license list marks the currently active license key.

2.2 Command Line License Manager Tool

The command line license manager tool *licmgr* is available in the *bin* directory for all platforms except Windows.

A complete description of all commands and options can be obtained by running the program without parameters:

```
licmgr
```

List all installed license keys

```
licmgr list
```

The currently active license for a specific product ist marked with a star '*' on the left side.

Add and delete license keys

Install new license key

```
licmgr store X-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX
```

Delete old license key

```
licmgr delete X-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX
```

Both commands have the optional argument `-s` that defines the scope of the action:

- `g`: For all users
- `u`: Current user

Select between different license keys for a single product

```
licmgr select X-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX-XXXXX
```

2.3 License Key Storage

Depending on the platform the license management system uses different stores for the license keys.

Windows

The license keys are stored in the registry:

- HKLM\Software\PDF Tools AG (for all users)
- HKCU\Software\PDF Tools AG (for the current user)

Mac OS X

The license keys are stored in the file system:

- /Library/Application Support/PDF Tools AG (for all users)
- ~/Library/Application Support/PDF Tools AG (for the current user)

Unix / Linux

The license keys are stored in the file system:

- /etc/opt/pdf-tools (for all users)
- ~/.pdf-tools (for the current user)

Note: The user, group and permissions of those directories are set explicitly by the license manager tool.

It may be necessary to change permissions to make the licenses readable for all users. Example:

```
chmod -R go+rx /etc/opt/pdf-tools
```

2.4 Setting the License Key via the API

When deploying applications that use the PrepTool API, you may prefer to pass the license key at runtime, rather than register the key on all potential target systems. The typical call sequence in the application will be as follows:

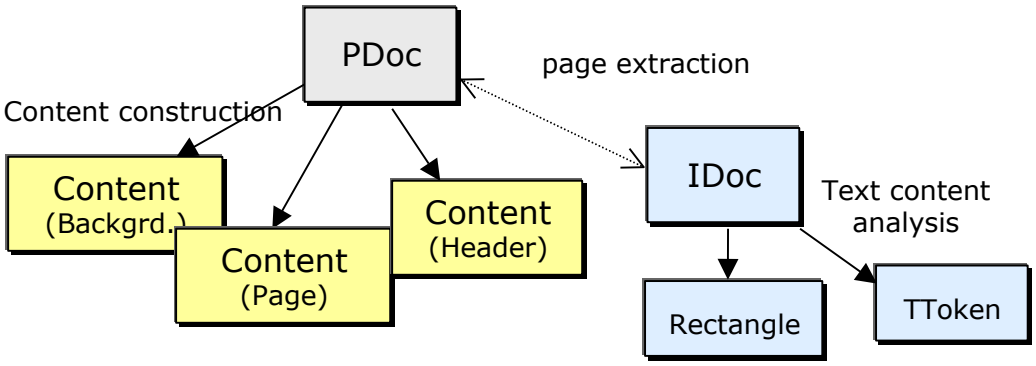
```
/* Initialize; this will load a license key stored on the computer */
PTInitialize();
/* Set the license key */
PTSetLicenseKey("0-12345-ABCDE-67890-FGHIJK-12345-ABCDE");
/* License check */
if (!PTGetLicenseIsValid())
{
    printf("no valid license found.\n");
    PTUninitialize();
    return 10;
}
```

Note: the COM and Java bindings automatically perform the “PTInitialize()” function. If a license key is installed on the computer, the application can detect that by directly calling “PTLib.getLicenseIsValid()” (Java) or querying the “LicenseIsValid” property of an IDoc, PDoc or PDFLinearizer Object (COM).

3 Object Model

The core entities in the PDF Prep Tool Suite are PDF files - either existing ones serving for input, or new ones being created. In the COM interface, these are the IDoc and PDoc types.

Another important entity for creating PDF files are content objects. A content object represents a layer of text and graphics objects that is used to construct a PDF page. The Prep Tool Suite uses content objects to construct PDF pages via API calls, and also to put a layer containing text, images etc. on top selected pages that are copied from existing PDF files.



There are also some auxiliary object types which are used to return structured information about PDF files, like text tokens on a page, or rectangle coordinates of media boxes or form field locations.

4 Processing Model

The processing model of the PDF Prep Tool Suite with regard to PDF creation is batch oriented. Pages are written sequentially without much buffering in memory. Contrary to interactive models where a document is opened, then modified randomly, and finally saved, the Prep Tool Suite works differently: Any modifications to be made to existing pages of PDF files are prepared either by reading the corresponding PDF objects into a cache where they are modified, or by posting modifications that are to be made when transferring PDF objects to output. After that, a copy operation saves the range of pages to output.

The reason for this model is resource conservation and speed.

5 Language Bindings

There are three different bindings to the Prep Tool Suite: A conventional native library interface (DLL on Win32), a COM interface for Win32, and Java wrapper classes based on JNI. The C++ classes of the implementation are not exposed directly in any API.

The native interface is most suitable for C/C++ applications on any platform (Windows or Unix), but can also be used from Visual Basic on Win32.

The COM interface is most suitable for Visual Basic applications, but can also be used by any other development environment that can make use of COM objects, such as Delphi. Unlike the other APIs, COM allows for optional and default parameters. You will get the appropriate hints in the Visual Basic development environment.

Object type/type of interface	***Native***	***COM***	***Java***
Reference to PDF output file	`Handle`	`PDoc`	`PTDoc`
Reference to PDF input file	`InputHandle`	`IDoc`	`PTInput`
Reference to current page	`ContentHandle`	`content`	`PTContent`
Reference to current header or background layer	`ContentHandle`	`content`	`PTContent`
Reference to current outline	`BookmarkHandle`	`Bookmark`	`PTBookmark`
Reference to current form field	`-`	`FormField`	`PTFormBox`, `PTFormData`
Reference to current linearized file	`-`	`PDFLinearizer`	`PTLinearizer`
Reference to current text token	`PTTokenInfo`	`TToken`	`PTTextToken`

The native binding uses the type VBSTR to return character strings from the Prep Tool Suite to the application. VBSTR is compatible with Visual Basic, i. e. Visual Basic will correctly free these strings again.

C applications need to explicitly free strings obtained from the Prep Tool Suite by calling PTFreeVBSTR. This function will call the Win32 function *SysFreeString()*.

On UNIX systems, PTFreeVBSTR simply calls the standard *free()* function from *stdlib.h*.

6 Getting Started

This chapter gives a brief overview of how the PDF Prep Tool Suite can be used. Important to know is:

- The PDF Prep Tool Suite never modifies an input file. Modifications are always applied and visible in the created output.
- Only one output file can be opened at a time. Several input files can be opened at once, but only one can be attached to an output file at a time.
- "Attached" means that when calling to *InputCopyPages*, or *InputCopyAll*, the pages of the attached input file are copied to the output file.

There are basically two possibilities to create a document:

6.1 Create a Document from Scratch

The PDF Prep Tool is not intended to be used as a PDF Creator, even though it provides the functionality to add text, raster graphics and vector graphics such as lines or rectangles.

When creating a document from scratch, content is to be written on the Page layer. The *Header* and *Background* layers cannot be used at this time.

In Visual Basic 6, creating a document from scratch could look like this:

```
Dim outdoc As New PREPTOOLLib.PDoc
outdoc.New "C:\temp\hello.pdf"
outdoc.Page.SetFont "Helvetica", 50
outdoc.Page.PrintText "Hello World.", 100, 300
outdoc.Close
```

6.2 Add Content to an Existing Input File

An input file can be opened by either creating an IDoc object, open a document and attach the IDoc to the PDoc object, or by a call to *InputOpen* of the PDoc object. Adding new content can be achieved by writing on the Header or Background layer. After a call to *InputCopyAll* or *InputCopyPages*, the content of the pages of the input document is merged with the Header and Background layers.

The Page layer cannot be used at this time to add content to the page. Writing on the Page layer creates a new page.

Here is a Visual Basic 6 sample:

```
Dim outdoc As New PREPTOOLLib.PDoc
outdoc.New "C:\temp\output.pdf"
outdoc.InputOpen "C:\temp\hello.pdf"
outdoc.Header.SetFont "Helvetica", 50
outdoc.Header.PrintText "Hello again.", 100, 400
outdoc.InputCopyPages 1, 1
outdoc.Close
```

7 Output PDF Creation

Creation of an PDF file for output is performed as shown in the table below. Note that the COM interface requires two steps, because a COM object cannot be created with parameters.

Native	`Handle PDocNew(const char* Filename, short PgWidth, short PgHeight, PTError* err)`

COM	`Dim Obj As New PDoc`
	`Dim Obj As Object: Set Obj = CreateObject("PrepTool.PDoc")`
	`New(Filename As String, Width As Integer, Height As Integer) As Boolean`

Java	`new PTDoc(String Filename)` `new PTDoc()`

The procedure *PDocNew* creates a new PDF file that is initially empty. The pages created and written to via the API will all have the specified page height and width.

Note that the PDF coordinate system has its origin at the left bottom of the page. The European format A4 has a width of 595 (points) and a height of 842.
A return value of 0 for the Handle means that the output file could not be created. In the native interface, you must use the *PTError** parameter to obtain the error code which is necessary to determine the reason why creation failed.

In the Java binding, the page format must be set by a separate method (*setPageSize*).

To create a PDF file in memory without writing it to disk, you can omit the file (i. e. specify NULL / 0 for this parameter). The Java API exhibits a constructor with no parameter for this.

The *CloseB* function will retrieve the byte array corresponding to the contents of the PDF file.

7.1 Set the PDF Version

Native	`PTError PDocSetPDFVersion(Handle h, const char* version)`
COM	`SetPDFVersion(Version As String)`
Java	`void setPDFVersion(String Version)`

This function sets the PDF version stored at the beginning of each PDF file. The default value is "1.4".

Note that the PDF version must be set before writing anything else to the output file.

7.2 Encryption

To provide a certain protection of PDF files, Adobe has specified "Standard Security" in the PDF specifications. This is based on encryption algorithms and is available in the

Prep Tool Suite.

Native	`PTError PDocSecurity(Handle h, const char* ownerPw, const char* userPw, const char* flags)`
COM	`SetSecurity(OwnerPassword As String, UserPassword As String, Flags As String)`
Java	`void SetSecurity(String Ownerpassword, String Userpassword, String Flags)`

This method will set the passwords and protection flags of the file to be created. It must be called immediately after the *New* method (before any objects are written to output).

The "Flags" parameter sets the protection attributes. It can contain a combination (or none) of the following characters:

"p": do not print the document from Acrobat

"c": changing the document is denied in Acrobat

"s": selection and copying of text and graphics is denied

"a": adding or changing annotations or form fields is denied

The following flags are defined for 128 bit encryption (PDF 1.4, Acrobat 5.0):

"i": disable editing of form fields

"e": disable extraction of text and graphics

"d": disable document assembly

"q": disable high quality printing

The flag "5" can be used in combination with one of the "old" flags to force 128 bit encryption without setting any of the i, e, d, or q flags. Note that using any of these Acrobat 5 related flags will produce a file that cannot be opened with older versions of Acrobat.

The flag "6" can be used to have the output AES-256 encrypted.

Omitting these flags will result in a PDF file that is fully usable when opened using the user password.

7.3 Disable Stream Compression

Native	`void PDocCompress(Handle h, short Yes)`
COM	not available (default: compression enabled)
Java	enabled if environment variable PDPREP_OPT_NC is defined

The function *PDocCompress* can be used to disable the compression of content streams generated by Prep Tool. By default, compression is enabled.

7.4 Font Renaming

Acrobat viewers before 4.05 had the problem of incorrectly rendering text with fonts

that were multiply defined in a file. PDF Prep Tool automatically renames such fonts to work around this viewer problem. This renaming can create new problems when printing the resulting file, and when the font is not embedded in the file. In these cases, you should use *SetPreserveFontNames* method to disable the renaming feature.

Native	`PTError PDocSetPreserveFontNames(Handle h, short on)`
COM	`SetPreserveFontNames(on As Boolean)`
Java	`void setPreserveFontNames(boolean on)`

7.5 Error Handling

Error handling is implemented via a "get last error" method for PDF input and output objects.

Native	`int PDocLastError(Handle h)`
COM	`ErrCode() As ErrorType`
Java	`int getLastError()`

The COM interface defines its own error codes which are defined in the COM interface.

The native interface returns the normal "errno" codes of the operating system where appropriate, and a set of special Prep Tool errors that are defined in the include file.

NOTE: the "success" error code has been changed to conform with "errno", i.e. a value of 0 (zero) corresponds to successful operation, rather than the value 1 which previously was returned in most cases. Please refer to the file *pdptdef.h*.

The Java interface uses Java exceptions combined with the native error codes. Please refer to the Java class definitions.

7.6 Open a PDF File for Input

You can open a PDF file to retrieve information from it via the API, or to use it as a resource to copy pages to an output file, or both.

This is how to open the input file by referring to an output file object:

Native	`PTError PDocInputOpen(Handle h, const char* inputFile)`
COM	`InputOpen(Filename As String) As Boolean`
Java	`Boolean inputOpen(String Filename)` `Boolean inputOpen(String URL)`

A call to *PDocInputOpen* makes resources of an existing PDF file available - either to copy a non built-in font into the output file, or to copy pages to the output file.

Only one input file can be active at a time. A subsequent call to *PDocInputOpen* will automatically close the previous input file.

A return value of !=PTSuccess (Java/COM: false) means that the input file could not be opened.

In Java it is possible to provide an URL instead of a file name.

If you want to open a PDF file for input without need to create some other PDF file, you can do this as follows:

Native	`InputHandle IDocOpen(const char* Filename, PTError* errCode)` `InputHandle IDocMemOpen(const char* pdfBytes, int len, PTError* errCode)`

COM	`Dim Obj1 As New IDoc`
	`Dim Obj2 As Object`
	`Set Obj2 = CreateObject("PrepTool.IDoc")`
	`Obj1.Open(Filename As String) As Boolean`
	`Obj1.OpenMem(Bytes [As Byte()]) As Boolean`

Java	`new PTInput(String Filename)`
	`new PTInput(String URL)`
	`new PTInput(byte[] pdfBytes)`

It is possible to open a PDF "file" stored in memory rather than referring to the file system using the *MemOpen* function. In Java, the PTInput constructor taking a byte array can be used for this.

When using *IDocMemOpen*, the "pdfBytes" are copied during this call and can be disposed of as needed (all language bindings).

In the COM interface the following construct can be used to ensure that the PDoc and its corresponding IDoc are running in the same appartment:

COM	`Dim Obj1 as New PDoc`
	`Dim Obj2 as IDoc`
	`Set Obj2 = Obj1.CreateIDoc`

To open a password protected (encrypted) PDF file, you need the following API calls:

Native	`InputHandle IDocOpenPw(const char* inputFile, const char* password, PTError* errCode)` `InputHandle IdocMemOpenPw(const char* pdfBytes, int len, const char* password, PTError* errCode)`
COM	`Open(Filename As String, Password As String) As Boolean` `OpenMem(Bytes, Password As String) As Boolean`

Java	`new PTInput(String Filename, String Password)`
	`new PTInput(String URL, String Password)`
	`new PTInput(byte[] pdfBytes, String Password)`

The COM API uses the same methods to open encrypted and non-encrypted files. The password parameter is optional.

7.7 Attach an Input File

When you have previously opened an existing PDF file using *IDocOpen*, you may later want to use it as a source for pages or other resources to create an output file.

Native	`PTError PDocAttach(Handle h, InputHandle hIDoc)`
COM	`Attach(Input As IDoc) As Boolean`
Java	`void attachInput(PTInput input)`

Note: Once you have attached an input file to an output PDF, you must not attach it to another output PDF file; also, you must not close it, because it will be closed automatically when the output PDF is closed.

Attaching a new input file to an output PDF will also close the previous input file (except when using the COM API).

7.8 Accessing the Current Input File

The make use of the full set of PDF file analysis features, you may want to know the input file object reference of an output file object.

Native	`InputHandle PdocGetInputHandle(Handle h)`
COM	`Input() As IDoc`
Java	`PTInput getInput()`

7.9 Set the Page Size and Orientation

Native	`PTError PDocPageSize (Handle h, short Width, short Height)`
COM	`PageSize(Width As Integer, Height As Integer)`
Java	`void setPageSize(short Width, short Height)`

Use this function to set the dimension of pages to be created. The width and height are specified in points corresponding to the standard PDF coordinate system. The MediaBox of the page will be set as [0 0 <width> <height>]. The default values are 595 by 842, i. e. A4 portrait. There are minimum and maximum values that vary between different versions of the Acrobat viewers.

If you want to create landscape pages, you can either set the width and height accordingly, or turn the coordinate system by printing from bottom to top while specifying a value of 90 for the Rotate attribute of the page. Please read the explanations about the PDF and text coordinate system in the specifications.

Native	`PTError PDocPageRotate(Handle h, short orientation)`
COM	`SetPageRotate(Orientation As Integer)`
Java	`void setPageRotate(int Orientation)`

The orientation for viewing the content of a page can be set using this function. Legal values that can be specified are 0 (default) and multiples of 90 (e. g. 270, -90, 180,

etc.).

These settings do not affect pages copied from existing PDF files, see *SetInputRotate* (below) for this.

To change the format or orientation of such pages, you can create empty pages of the desired format and add the content of the existing file using the "Logo" functions. This allows you to use arbitrary coordinate transformations for positioning and scaling the page. This method will not work to copy annotations (such as form fields, links, etc.).

Native	`PTError PDocSetInputRotate(Handle h, short orientation)` `PTError PDocClearInputRotate(Handle h)`
COM	`SetInputRotate(Orientation As Integer)` `ClearInputRotate`
Java	`void setInputRotate (short Orientation)` `void clearInputRotate()`

SetPageRotate has the effect to replace the page rotation stored in input PDFs with the value specified when copying pages into the output PDF. To restore the behavior of keeping the value as in the input file, use *ClearInputRotate*.

7.10Set the Crop Box

The Crop Box is the displayed part of the PDF. This function allows the setting the Crop Box for pages that are created or copied. The Crop Box must never be larger than the Media Box.

Native	`PTError PDocSetCropBox(Handle h, float Left, float Bottom, float Right, float Top)`
COM	`SetCropBox(Left As Single, Bottom As Single, Right As Single, Top As Single)`
Java	`void setCropBox(float Left, float Bottom, float Right, float Top)`

The crop box can be set for a newly created page. It is also applied to the pages that are copied using *InputCopyPages* or *InputCopyAll*.

7.11Adding a New Page

A new page is automatically added to the output file when you request its handle for the first time or after a call to *PDocNewPage*.

Native	`ContentHandle PDocGetContentHandle(Handle h)`
COM	`Page() As content`
Java	`PTContent getPageContent()`

The content handle and the Java PTContent object are only valid as long as the page is in construction. Once it is written to output, it is invalid and may no longer be used.

The COM object can be reused to access the next page after a call to the *NewPage* method only. In all other cases, a new Content object reference must be obtained from

the PDoc object.

7.12 Accessing the Current Header or Background Content Layer

In order to construct the header content layer, you need the corresponding object reference from the output file object.

Native	`ContentHandle PDocGetHeaderHandle(Handle h)` `ContentHandle PDocGetBackgroundHandle(Handle h)`
COM	`Header() As content` `Background() As content`
Java	`PTContent getHeaderContent()` `PTContent getBackgroundContent()`

A header (or background) content reference is valid as long as the header is not cleared. After a call to *HeaderClear* or *BackgroundClear*, it becomes invalid and may no longer be used.

The header content layer will be placed on top of pages copied into an output PDF (using *InputCopyPages*), while the background layer will be placed behind. Note that the background content may be hidden by non-transparent pages of an input file.

8 Retrieving File Information

In the native interface, you refer to a handle of type "InputHandle".

In the COM interface, you refer to an object of type "IDoc".

In the Java binding, you refer to an object of class "PTInput".

You can obtain this kind of object reference in one of the ways described above.

8.1 Obtain the PDF Version

Native	`VBSTR IDocPDFVersion(InputHandle h)`
COM	`GetVersion() As String`
Java	`String getPDFVersion()`

Returns the PDF version of the file, as stored in the file header.

8.2 Obtain the File Name

Native	`VBSTR IDocGetFileName(InputHandle h)`
COM	`GetFileName() As String`
Java	`String getFileName()`

Retrieves the name of the PDF file. If the file was opened from memory, a unique string is returned starting with "internal: ". If the file is not open, an empty string is returned.

8.3 Obtain the Keys List

Native	`PTError IDocGetInfoKeys(InputHandle h, VBSTR* keys)`
COM	`GetInfoKeys() As String`
Java	`String getInfoKeys()`

This function returns a carriage-return separated list of the keys that are present in the /Info attribute of the PDF file. The keys are returned with the leading slash character – e. g. /Author, /Title, etc.

8.4 Obtain Document Attributes

Native	`PTError IDocGetInfoAttr(InputHandle h, const char* key, VBSTR* value)` `PTError IDocGetInfoAttrU(InputHandle h, const char* key, PDBSTR value)`

COM	`GetInfoAttr(ByVal Key As String) As String`
Java	`String getInfoAttr(String Key)`

This function returns the value of a document attribute stored in the /Info attribute of the PDF file.

8.5 Get Meta Data

Native	`VBSTR IDocGetMetaData(InputHandle h)`
COM	`GetMetaData() As String`
Java	`String getMetaData()`

GetMetaData returns the XML meta data stored in the PDF document.

8.6 Get the Name and Current Data of a Form Field

Native	`PTError IDocGetFormData(InputHandle h, short FieldNum, VBSTR* Name, VBSTR* Data, VBSTR* Description, int* FormFlags, int* AnnotFlags, VBSTR* FieldType)`
COM	`GetFormData(ByVal FieldNum As Integer, Name As String, Data As String, Descr As String, Multiline As Boolean) As Boolean`
Java	`PTFormData getFormData(int FieldNum)`

The parameter FieldNum (default=1) is an iterator by which you can obtain the names and current data of all text form fields. FieldNum runs from 1 to the number of form fields. A result of !=PTSuccess/False/null will be returned, if you go beyond the last form field.

PTFormData.Name = Name as String

PTFormData.Data = Data as String

PTFormData.Description = Description as String

The native and the Java interface also supply type information (FieldType). This type information is composed of the field type of the field itself, followed by the export values (separated by new-line characters).

Note that there can be more than one instance of a form field. If this is the case, each instance can have different flags, and you need to use *GetFormBox* to check the individual settings.

8.7 Get the Position of a Form Field

Native	`PTError API IDocGetFormBox(InputHandle h, const char* fieldName, float box[], short inst, int* page, short* fontID DEFAULT_NULL, float* fs, short* al, int* formFlags, int* annotFlags)`
COM	`GetFormBox(ByVal FieldName As String, ByVal Instance As Short, X`

	`As Single, Y As Single, W As Single, H As Single, Page As Integer, FontID As PtFormFontType, Fontsize As Single, Alignment As Short, Formflag As PTFormFlags, Annotflags As PTAnnotFlags) As Boolean`
Java	`PTFormBox getFormBox(String Fieldname, int Instance)` `PTFormBox getFormBox(String Fieldname)`

There can be more than one field occurrence with a certain name. All these occurrences share the same data and also other attributes like description or multi-line. Individual form fields have their own location, display text in a different font and with different alignment. *GetFormBox* returns the latter information that belongs to individual form fields. The "instance" parameter serves to distinguish different occurrences. Instance numbers start at 1. The function will return a *PTSuccess* (True/non-null) result if the instance is found.

The parameters are:

- Fieldname: the name of the form field (IN)
- Instance: a numerator to distinguish between form fields that have the same name (IN)
- box,

  box[0-3] =X/Y/W/H: the coordinates of the rectangle occupied by the form field

  if (box.length > 4)

  Page = (int) box [4];

  if (box.length >= 10) {

  FontID = (int) box [5];

  FontSize = box [6];

  Alignment = (short) box [7];

  FormFlags = (short) box [8];

  AnnotFlags = (short) box [9];
- Page: the number of the page on which the field is located (1..number of pages)
- FontID: the identification of the font used to display text (e. g. Helvetica = 0, see declaration of font constants)
- FontSize: the size of the text being displayed
- Alignment: the alignment for displaying the form text (0 = left, 1 = centered, 2 = right)
- Formflags: the flags set for the form ("/Ff" entry in the form field's dictionary, see *AddTextField*)
- Annotflags: the general annotation flags ("/F") set for the field

The form specific flags being returned are described in the PDF specification:

- 1: read-only (flag & 1 != 0)
- 2: required (flag & 2 != 0)

- 3: no export (flag & 4 != 0)
- 13: multi-line (flag & 4096 != 0), etc.

The general annotation flags are

- 1: invisible
- 2: hidden
- 3: printable, etc.

8.8 Get Information about Pages

Native	`int IDocNumPages(InputHandle h)`
	`PTError IDocAcquirePage(InputHandle h, int Page)`
	`PTError IDocPageBox(InputHandle h, float* X, float* Y, float* Width, float* Height)`
	`PTError IDocMediaBox(InputHandle h, float* X, float* Y, float* Width, float* Height)`
	`short IDocPageRotate(InputHandle h)`

COM	`NumPages() As Long`
	`GoPage(PageNum As Long) As Boolean`
	`GetVisibleBox(Left, Bottom, Right, Top)`
	`GetMediaBox(Left, Bottom, Right, Top)`
	`GetRotate() As Integer`

Java	`int getNumPages()`
	`boolean acquirePage(int PageNumber)`
	`PTRectangle getPageBox()`
	`PTRectangle getMediaBox()`
	`short getPageRotate()`

This set of functions can be used to retrieve information about individual pages in a PDF file.

The "visible box" corresponds to the crop box; if none is present, the media box is returned.

The "Rotate" attribute of a page tells a viewer application that the page shall be rotated for presentation.

8.9 Retrieve Text from a PDF File

Native	`PTError IDocReadText(InputHandle h, VBSTR* text, PTTokenInfo* m, VBSTR* font)` `PTError IDocReadTextU(InputHandle h, PDBSTR* text, PTTokenInfo*`

	`m, VBSTR* font)`
COM	`GetToken() As TToken`
Java	`PTTextToken readTextToken()`

This function retrieves text fragments from a PDF file's pages. The metrics structure contains the coordinates, font size, width and orientation of the retrieved character string. The page number is also contained, because *ReadText* passes automatically to the next page when no more text is found on a page.

The *PTTokenInfo* structure contains a float array indicating to position of each individual character of the retrieved string (CharRightPos). This float array is dynamically allocated and must be initialized before calling *IDocReadText* and again afterwards when not used any more. This is done with the functions *PTInitToken()* and *PTFreeToken()*. For C++ programmers, the class *CPTTokenInfo* is available which takes care of initializing the structure and freeing the allocated memory again.

8.10Retrieve Bookmarks from a PDF File

Native	`BookmarkHandle IDocGetBookmarkRoot(InputHandle h)`
	`PTError PBMGoNext(BookmarkHandle h)`
	`PTError PBMGoUp(BookmarkHandle h)`
	`PTError PBMGoDown(BookmarkHandle h)`
	`PTError PBMReset(BookmarkHandle h)`
	`PTError PBMGetTitle(BookmarkHandle h, VBSTR* title)`
	`PTError PBMGetTitleU(BookmarkHandle h, PDBSTR* title)`
	`PTError PBMGetLevel(BookmarkHandle h, int* level)`
	`PTError PBMGetNumChildren(BookmarkHandle h, int* numChildren)`
	`PTError PBMKidsVisible(BookmarkHandle h, bool* kidsVisible)`
	`PTError PBMGetInfo(BookmarkHandle h, VBSTR* info)`
	`PTError PBMRelease(BookmarkHandle h)`
	`BookmarkHandle API PBMClone(BookmarkHandle h)`

COM	`GetBookmarkRoot() As Bookmark`
Java	`PTBookmark getBookmarkRoot()`

The bookmark root node can be retrieved trough the function *GetBookmarkRoot*. Java and COM uses the classes PTBookmark and Bookmark to encapsulate the appropriate native functions (*GetTitle*, *GoNext*, ..)

Navigate trough the tree by using the functions *GoNext* to go to the next bookmark, *GoDown* to go one level deeper, *GoUp* to go one level up and reset to move to the root bookmark. *GoDown*, *GoUp* and *GoNext* return false if there isn't a next bookmark or the node has no children (*GoDown*) or is no parent (*GoUp*).

To retrieve information about the current bookmark use the get functions. *GetTitle* returns the bookmark title. The native method *GetTitleU* returns the title in Unicode.

Java and COM methods always return Unicode strings. *GetLevel* returns the current level of the bookmark. The root level is –1. *GetNumChildren* returns the number of children for the current bookmark. Use the Clone function to get a copy of the current bookmark. For Java and the native API you have to release a bookmark to free the memory. Use the release function to do this.

To release the title string in the native API, use the functions PTFreeVBSTR and PTFreePDBSTR.

The *GetInfo* function returns additional information about a bookmark. This information is returned in a character string. The string content depends on the type of action or destination attached to the bookmark. The following types are supported:

GoTo: Go to a destination in the current document (Starting at 0).

GoToR: Go to a destination in another document.

Launch: Launch an application.

URI: Open an Internet link.

The action types have to be interpreted as follows:

Action type	**String**
GoTo	GoTo *Destination*
GoToR	GoToR *file Destination*
Launch	Launch *file*
URI	URI web-link

A destination can be one of the following:

page /XYZ *left top zoom*
page /Fit
page /FitH *top*
page /FitV *left*
page /FitR *left bottom right top*
page /FitB
page /FitBH *top*
page /FitBV *left*

For more information about action types and destinations, refer to the PDF-Reference.

Use a parser to split up the string into the tokens. The separation between two arguments is the blank character.

8.11 Retrieve Annotations from a PDF File

Native	`PTError IDocGetAnnotation(InputHandle h, PTAnnotType* AnnotType, float rect[], int* BorderStyle, int* pIdentification)`
	
	`PTError IDocGetAnnotationInfo(InputHandle h, VBSTR* AnnotInfo)`
	
	`PTError IDocGetAnnotationInfoU(InputHandle h, PDBSTR* AnnotInfo)`

COM	`GetAnnotation(Type As PTAnnotType, Info As String, Left As`

	`Single, Bottom As Single, Right As Single, Top As Single, BorderStyle As Long, Identification As Long) As Boolean`
Java	`PTAnnotData readAnnotation()`

These functions are used to retrieve annotations from a PDF document. Two types of annotations can be retrieved, text and link annotations. The type is retrieved through the *PTAnnotType* structure.

The function *GetAnnotation* returns one annotation per call for the current page. Call the function again to retrieve the next annotation. The function will return an error if it has no next annotation.

Use the *GetAnnotationInfo* function in the native API to retrieve the info string from the current annotation. The *GetInfoAnnotationU* function, retrieves the info string in Unicode. Use PTFreeVBSTR and PTFreePDBSTR to free these strings.

The return values are interpreted as follows:

AnnotType: the type of the annotation (eText or eLink)
rect[]: the location of the annotation on the page (left, bottom, right, top)

Info: If it's a text annotation this holds the text. If it's a Link annotation this holds an action or named destination. See 'Retrieve Bookmarks from a PDF File' for more information about the structure of the info string in this case.
Java encapsulates the return values in the class *PTAnnotData*.

8.12 Retrieve the Border Style from Annotations

Native	n.a.
COM	`GetBorderStyle(ID As Long) As IBorderStyle`
Java	n.a.

If the annotation has a Border Style dictionary (entry BS), this function returns an *IBorderStyle* interface, otherwise nothing is returned. ID is the identification of the annotation which is received using the method *GetAnnotation*.

IBorderStyle has the following properties:

String BS	Describes the border style. The following substrings are possible: "S" (Solid), "D" (Dashed), "B" (Beveled), "I" (Inset), "U" (Underline).
Long ColorRGB	The color as RGB value. ColorRGB = red + 256 * green + 256 * 256 * blue. Where red, green and blue are values 0-255.
String DashArray	A dash array defining a pattern of dashes and gaps to be used in drawing a dashed border. The array is returned a string, the separator is the blank. For example, a "1 2" string specifies a border drawn with 1-point dashes alternating with 2-point gaps.
Integer DashOff	The size of the gaps. See DashArray.
Integer DashOn	The size of the dash. See DashArray.
Integer Width	The border width in points. If this value is 0, no border is drawn.

8.13Get List of Fonts

Native	`PTError IDocGetFonts(InputHandle h, int PageNumber, VBSTR* Fonts)`
COM	`GetFonts(Optional ByVal PageNumber As Long) As String`
Java	`String getFonts(int Page)`

This function returns a "\r" (Chr$(13)) separated list of the fonts contained in the PDF file. If the Page parameter is specified as 0, the whole document is searched for fonts.

8.14Get Color Information

Native	`short IDocNumColorSpaces(InputHandle h)` `VBSTR IDocGetSeparation(InputHandle h, short index)`
COM	`NumColors() As Long` `GetColor(ByVal Index As Long) As String`
Java	`int getNumColors()` `String getColor(int Index)`

These functions return information about ColorSpace entries in the resources dictionary of the current page of the input file (*AquirePage* must previously be called).

The index to retrieve the names of the color space separation runs from 1 to the number of colors.

8.15Save File Attachment

Native	`PTError IDocSaveAttachment(InputHandle h, const char* filename)`
COM	`SaveFileAttachment(FileName As String) As Boolean`
Java	`void saveAttachment(OutputStream os)`

This function permits retrieval of the file that is embedded in a *FileAttachment* annotation.

Note that *SaveFileAttachment* depends on the *GetAnnotation* function, and will only work when the last annotation returned by *GetAnnotation* is a file attachment.

8.16Close the File

Native	`PTError IDocClose(InputHandle h)`
COM	`Close() As Boolean`
Java	`void close()`

This function closes the input file and releases all resources associated with it. When the COM object's reference count goes to zero, an automatic close is performed.

When using the Java API, you must be careful: call "close" only, if you obtained the

PTInput object using "new". If you obtained it via *PTDoc.getInput()*, the input file will be closed when closing the PTDoc object.

8.17Get UserUnit

Native	`float IDocUserUnit(InputHandle h)`
COM	`GetUserUnit() As Single`
Java	`n.a.`

Returns the UserUnit as float if defined in the PDF document. If no UserUnit is defined, 1 is returned.

Content Construction

The following methods refer to objects of type content, and can thus be equally applied to "print" to a page or to construct the content layer of a header.

In the native interface, you refer to a handle of type "ContentHandle".

In the COM interface, you refer to an object of type "Content".

In the Java binding, you refer to an object of class "PTContent".

8.18Set the Font for Text Output

Native	`PTError PConSetFont(ContentHandle h, const char* fontName, float fontSize)`
COM	`SetFont(ByVal FontName As String, ByVal FontSize As Single) As Boolean`

Java	`boolean setFont(String Name, float Size)`
	`boolean setFont(String Name)`
	`boolean setFont(float Size)`

In the native and COM interfaces, the parameters Name and Size are optional. Once you have set the font's name, it is possible to change its size by just passing the new size. For missing arguments, you can specify 0.

The procedure *PConSetFont* must be called prior to *PConPutText* to set the font to be used and its size. Only predefined Acrobat fonts can be specified here ("Helvetica", "Helvetica-Bold", "Helvetica-Oblique", "Times-Roman", "Times-Italic", "Times-Bold", "Courier", "Courier-Oblique", "Courier-Bold", "Symbol", "ZapfDingbats").

The fonts "Helvetica-BoldOblique", "Courier-BoldOblique" and "Times-BoldItalic" are built in fonts, but cannot be used because their definition requires additional information which is not yet supported. However, if these fonts or any other non standard font is defined in the current input file, PT will copy that font to the output file.

SetFont returns FALSE if the font cannot be set (i. e. is not a standard font and is not found in the current input file).

Note the following issues about using fonts:

When there is no current input file (see *PDocInputOpen*), you must only use standard built-in fonts like Helvetica, Times-Roman, etc. (see *PConSetFont*).

When there is a current input file, *SetFont* tries to find a font with this name in the input file and copy the font data to the output file. It is then legal to use this font.

To use a non standard font, you can thus create a template file containing the font data. As Acrobat optimizes the font data to what is actually necessary, make sure you place the full variety of characters that you later need into the file. To refer to the font, you specify its name in the "fontName" parameter. It is actually sufficient to specify only a significant portion of the name (matching is case sensitive - check the spelling of the font in the template file!).

Some fonts will have the effect that the encoding of individual characters in the PDF file is different from the corresponding ASCII code. Currently, you can only use fonts that conform to certain conventions. The standard fonts "Helvetica", "Helvetica-Bold", "Helvetica-Oblique", "Times-Roman", "Times-Italic", "Times-Bold", "Courier", "Courier-Oblique", "Courier-Bold", "Symbol", "ZapfDingbats" should always work. Other fonts will be embedded into the PDF file. Their encoding depends on the tool which produced the PDF file. PDFWriter on Windows produces a standard ASCII encoding (WinAnsiEncoding) for a font for which Distiller Assistant will create an encoding which shifts codes by 29. (You will notice this also in Acrobat, when you select text, copy it to the clipboard, and try to use it in another application).

The Prep Tool DLL uses a heuristic to determine if there is a code shift by evaluating the "FirstChar" key of the font dictionary. It uses this value to shift the code, assuming that this code corresponds to the first printable character which is a blank space (ASCII 32). When you prepare a template PDF, make sure it contains a blank space (plus all other characters you want to have available).

8.19Set Text Spacing

Native	`PTError PConSetCharSpacing(ContentHandle h, float value)`
	`PTError PConSetWordSpacing(ContentHandle h, float value)`
	`PTError PConSetTz(ContentHandle h, float value)`

COM	`SetCharSpacing(ByVal Value As Single)`
	`SetWordSpacing(ByVal Value As Single)`
	`SetTz(ByVal Value As Single)`

Java	`void setCharSpacing(float value)`
	`void setWordSpacing(float value)`
	`void setTz(float value)`

The character spacing (Tc) adds some space between each character of a text string. The measure is in points. It does not scale with the text's font size. The word spacing is an additional spacing that is applied to space characters only.

The "Tz" value controls the horizontal scaling of text. The default value is 100.

8.20Set the Gray Level for Lines and Filling

Native	`PConSetGray(ContentHandle h, float line, float fill)`
COM	`SetGrayLevel(ByVal GrayLine As Single, ByVal GrayFill As Single)`

Java	`void setGray(float line, float fill)`
	`void setGrayLine(float value)`
	`void setGrayFill(float value)`

Text characters consist of a line shape (that is usually not drawn) and the fill area. Thus, you can set the gray level of text by setting the gray level for filling ("g" operator

in PDF).

.. etc.

8.21Set the Color for Lines

Native	`PTError PConLineColor(ContentHandle h, float red, float green, float blue)` `PTError PConLineColorCMYK(ContentHandle h, float cyan, float magenta, float yellow, float black)`
COM	`SetLineColor(ByVal red As Single, ByVal green As Single, ByVal blue As Single)` `SetLineColorCMYK(ByVal cyan As Single, ByVal magenta As Single, ByVal yellow As Single, ByVal black As Single)`
Java	`void setLineColor(float red, float green, float blue)` `void setLineColorCMYK(float cyan, float magenta, float yellow, float black)`

This method sets the color of lines. The values of r, g, b must lie in the range of 0 and 1. They correspond to the contributions of red, green and blue. 0,0,0 corresponds to black, 1,0,0 to red, etc. Alternatively the color can be set using CMYK (Cyan, Magenta, Yellow, Black) parameters. The range of the CMYK color parameters lies between 0 and 1.

8.22Set the Color for Filling

Native	`PTError PConFillColor(ContentHandle h, float red, float green, float blue)` `PTError PConFillColorCMYK (ContentHandle h, float cyan, float magenta, float yellow, float black)`
COM	`SetFillColor(ByVal red As Single, ByVal green As Single, ByVal blue As Single)` `SetFillColorCMYK(ByVal cyan As Single, ByVal magenta As Single, ByVal yellow As Single, ByVal black As Single)`
Java	`void setFillColor(float red, float green, float blue)` `void setFillColor(float cyan, float magenta, float yellow, float black)`

This method sets the color for filling shapes. It also affects the color of text.

8.23Set the Alpha Transparency for Filling and Stroking

Native	`PTError PConSetFillAlpha(ContentHandle h, float alpha)` `PTError PConSetStrokeAlpha(ContentHandle, float alpha)`

COM	`SetFillAlpha(ByVal alpha b As Single) As Boolean` `SetStrokeAlpha(ByVal alpha b As Single) As Boolean`
Java	`n.a.`

This method sets the alpha transparency for filling shapes and stroking lines. It also affects text.

8.24Using Color Spaces

It is also possible to use color spaces to set the fill and line colors. In order to have a specific color space available for use, it must be defined in the current input file, or it must have been previously copied to the current output file from some other input file (see *PDocInputCopyColor*).

Native	`PTError PConSetFillCS(ContentHandle h, const char* color, float scn)` `PTError PConSetLineCS(ContentHandle h, const char* color, float scn)`
COM	`SetFillCS(ByVal Color As String, ByVal scn As Single) As Boolean` `SetLineCS(ByVal Color As String, ByVal scn As Single) As Boolean`

Java	`void setFillCS(String Color, float scn)`
	`void setFillCS(String Color)`
	`void setLineCS(String Color, float scn)`
	`void setLineCS(String Color)`

These functions return a boolean indicating successful setting of the color. To obtain a list of all available colors, you can use the functions *IDocNumColorSpaces* and *IDocGetSeparation*.

8.25Placement of Character Strings

Native	`PTError PConPutText(ContentHandle h, const char* text)`
	`PTError PconPutTextU(ContentHandle h, const PDNSTR text)`
	`PTError PConPutLn(ContentHandle h)`

COM	`PrintText(Text As String, x As Single, y As Single)` `PrintNewLine()`
Java	`void putText(String Text)` `void putLn()`

Print a text string using the current font. You previously need to set the location (Text Matrix).

The COM interface optionally accepts new coordinates for the text.

PutLn adds a T* operator to the stream.

8.26Placement of a Logo

Native	`PTError PDocLogo(Handle h, const char* logoFile, short backGround)`
	`PTError PDocLogoFile(Handle h, const char* logoFile, PTClipType ct)`
	`PTError PDocLogoInput(Handle h, InputHandle hIDoc, PTClipType ct)`
	`PTError PConPrintLogo(ContentHandle h, long id)`
	`InputHandle PDocGetLogoHandle(Handle h)`

COM	`{PDoc.}SetLogoFile(ByVal Filename As String, Optional Clipping As PTClipType) As Boolean`
	`{PDoc.}SetLogoInput(LogoInput As IDoc, Optional Clipping As PTClipType) As Boolean`
	`{PContent.}PrintLogo(Num As Long) As ErrorType`
	`{PDoc.}Logo() As IDoc`

Java	`boolean {PTDoc.}setLogoFile(String Filename)`
	`boolean {PTDoc.}setLogoFile(String Filename, int cliptype)`
	`boolean {PTDoc.}setLogoFile(String URL)`
	`boolean {PTDoc.}setLogoFile(PTInput input, int cliptype)`
	`void {PTContent.}putLogo(int LogoPageNum)`
	`PTInput {PTDoc.}getLogoFile()`

First, you need to define which PDF file to extract logos from. Subsequently, you can select any page of the logo file as the logo to be placed either on the page content or on the header layer.

The box, which should be applied when copying the logo page can be set to any box (*pdClipTrimBox*, *pdClipCropBox*, *pdClipMediaBox*, *pdClipBleedBox*). The default is the *TrimBox*.

The native interface works slightly different for backward compatibility reasons: *PDocLogo* implicitly also prints the logo from page one, and it is possible to put it in the background. The new function *PDocLogoFile* only opens the file and leaves it up to *PConPrintLogo* to use it.

A PTNullRef value returned by the PrintLogo function indicates that the page being used as a logo does not contain any contents and thus has no effect on the output. The Java function putLogo will not raise an exception in this case as it would when encountering some other error (such as PTFailed when passing an invalid page number).

Please note that pages merged from existing PDF files may not be transparent and thus cover the background logo. On the other hand, the logo may not be transparent and hide existing contents if placed in the foreground. The best technique is thus to make sure the logo is transparent as required and place it in the foreground. As a help to this, it is possible to apply a crop box to the logo file (see below). Unfortunately,

Acrobat insists on a minimal size for cropped pages. You may need other ways to reduce the crop box further (the "pdcat" tool can do it). With Adobe Acrobat, you can remove any background rectangles with the TouchUp Object Tool. PrepTool inspects the logo's content stream and removes a white background if it is the first object in the stream.

There is no coordinate transformation when placing the logo, i. e. it will be shown at the same offsets to the coordinate system origin (0,0 - left, bottom) as in the uncropped logo file. You can use the DrawCmd function to set a coordinate system transformation (PDF "cm" operator), if you want to set the position of the logo via the API.

The bounding box (clip rectangle applied to the logo when being placed on a page) for the logo corresponds to the TrimBox if specified - otherwise the MediaBox of the logo file).

Note: the same logo can be applied to several PDF files to be merged.

Several logo files can be used to contribute to the construction of a PDF document. An output document keeps the logo files open, and you can switch back to a previously used logo file by setting it again. The *PrintLogo* (*putLogo*) method applies to the currently active logo file.

8.27Placement of an Image

An image imported via *CreateImage* can be placed into a page (or header) content using *PrintImage*.

The X/Y/W/H parameters can be omitted. In this case, no coordinate transformation to place the image in the specified rectangle is generated. If you want to rotate the image, it would be necessary that you explicitly generate the transformation matrix before placing the image.

Native	`PTError PConPrintImage(ContentHandle h, int ident, float X, float Y, float Width, float Height)`
COM	`PrintImage(ByVal Ident As Long, ByVal X As Single, ByVal Y As Single, ByVal Width As Single, ByVal Height As Single) As Boolean`
Java	`void PintImage(int ident, float X, float Y, float Width, float Height)`

To place an image via a coordinate transformation, you need to issue the following PDF operator sequence:

DrawCmd("q")	save the current coordinate system state
DrawCmd("1 2 3 4 5 6 cm")	set the coordinate transformation using the "cm" operator
PrintImage(1)	generate the XObject placement into the stream
DrawCmd("Q")	restore to the saved coordinate system state

Note that (1, 2, 3, 4, 5, 6) is just an example showing the syntax of the command. The actual numbers will be determined by the scaling, rotation, and positioning parameters you have.

8.28Embedding any PDF Text Operator

Native	`PTError PConTextOp(ContentHandle h, const char* command)`
COM	`TextCmd(ByVal Command As String)`
Java	`void putTextOp(String Command)`

You can pass any legal PDF text operator directly to the PDF stream. Correctness of the command is not checked. PT only makes sure that your command will be surrounded by "BT" and "ET" operators.

8.29Set the Spacing of Text Lines

Native	`PTError PConSetLineSpacing (ContentHandle h, float value)`
COM	`SetLineSpacing(TL As Single)`
Java	`void setLineSpacing(float TL)`

This function will send a "TL" operator to the PDF stream. A useful line spacing would be equal to the current font size.

8.30Set the Text Matrix

Native	`PTError PConSetTm(ContentHandle h, float a, float b, float c, float d, x float, y float)`
COM	`SetTm(a As Single, b As Single, c As Single, d As Single, x As Single, y As Single)`
Java	`void setTm(float a, float b, float c, float d, float x, float y)`

Set the text matrix. The default text matrix is [1 0 0 1 0 0]. The first 4 numbers determine the orientation of the text being written subsequently. [1 0 0 1] means text is written in increasing x direction and constant y coordinate. The last two numbers in the text matrix define the coordinates of the starting point for text.

8.31Set a Relative Starting Position for Text (Tab)

Native	`PTError PConPutTab(ContentHandle h, float a, float b)`
COM	n.a.
Java	`void putTab(float a, float b)`

Issues a “Td” operator with the specified arguments. (See PDF specification).

8.32Calculate the Width for a Character String

Native	`float PConGetTextWidth(ContentHandle h, const char* text)` `float PConGetTextWidthU(ContentHandle h, const PDBSTR text)`

COM	`GetTextWidth(ByVal Text As String) As Single`
Java	`float getTextWidth(String Text)`

This function calculates the length that the specified text string would need with the current font settings. You can use this to adjust the starting coordinates for center or right alignment.

Please note that the function does not take into account any character or word spacing that you might have set using the *TextOp* function.

8.33Text Tables

Native	`short PConTableHeight(ContentHandle h, short nrRows)`
	
	`PTError PConTableDraw(ContentHandle h, short Left, short Top, short NumRows, short NumCols, short ColumnWidths[])`
	
	`PTError PConTableText(ContentHandle, short Row, short Column, const char* Text, short Alignment)`
	
	`PTError PConTableTextU(ContentHandle, short Row, short Column, const PDBSTR Text, short Alignment)`

COM	`GridHeight(nRows As Integer) As Integer`
	
	`PrintGrid(x As Integer, y As Integer, nRows As Integer, col1Width As Integer, col2Width As Integer, col3Width As Integer, col4Width As Integer)`
	
	`GridText(row As Integer, col As Integer, Text As String, Alignment As Integer)`

Java	`short calucateGridHeight(short nRows)`
	
	`void drawGrid(short x, short y, short nRows, short colWidths[])`
	
	`void putGridText(short row, short col, String text, short Alignment)`

This set of functions lets you draw the border lines of a simple table and fill the table with text. You need to set the text font and size first. This setting will determine the vertical dimensions of the table.

PrintGrid must always be called prior to printing text. If you do not want any grid lines to be drawn, set the line width to zero (*SetLineWidth(0)*).

The column widths need to be specified explicitly. In the COM interface, you can have at most 4 columns. The parameters are optional. You will get as many columns as you specify widths.

8.34Draw a Line or Polygon

Native	`PTError PConSetLineWidth(ContentHandle h, float Width)`
	
	`PTError PConMoveTo(ContentHandle h, float X, float Y)`
	
	`PTError PConDrawTo(ContentHandle h, float X, float Y)`

COM	`SetLineWidth(Width As Single)`
	`MoveTo(x As Single, y As Single)`
	`DrawTo(x As Single, y As Single)`

Java	`void setLineWidth(float Width)`
	`void moveTo(float x, float y)`
	`void drawTo(float x, float y)`

Use these functions to draw a line or line polygon.

Note that there are different possible settings for line joins. Please refer to the PDF specifications ("j" operator).

8.35Draw a Rectangle

Native	`PTError PConRectangle(ContentHandle h, float x, float y, float width, float height, short how)`
COM	`DrawRect(x As Single, y As Single, w As Single, h As Single, how As ShapeFlags)`
Java	`void drawRectangle(float Left, float Bottom, float Width, float Height, int FillType)`

Draw a rectangle with the specified location and dimensions. The parameter "how" determines, if the rectangle is filled and if the border is drawn: 0=fill area only, 1=both, 2=border only

8.36Draw Curves

Native	`PTError PConCurveTo(ContentHandle h, float xy[], short type)`
COM	`CurveTo(x1 As Single, y1 As Single, x2 As Single, y2 As Single, Optional x3 As Single, Optional y3 As Single, type As PTCurveType)`
Java	`void curveTo(float xy[], char type)`

Draw a Bézier curve of the specified type ('c', 'v', or 'y'; see PDF specifications).

The 'c' type curve requires 3 coordinate pairs, the other types only 2.

This function can be used to extend the current path – just like *drawTo*.

8.37Area Filling and Clipping

Native	`PTError PConDrawArea(ContentHandle h, short clip)`
COM	`DrawArea(Optional Clip As Boolean)`
Java	`void drawArea()` `void drawAndClipArea()`

Close the path constructed with *drawTo* (and/or *curveTo* calls, and fill with the current color. Optionally, the clip area is also set to this area.

8.38Embedding any PDF Non-Text Commands

Native	`PTError PConDrawOp(ContentHandle h, const char* command)`
COM	`DrawCmd(ByVal Command As String)`
Java	`void putDrawOp(String Command)`

Pass the specified PDF command string as is to the content stream. You can use this to make use of many PDF features that are not available by specific API calls.

9 Form Fields, Annotations

9.1 Set the Data

Native	`PTError PDocInputSetFormData(Handle h, const char* fieldName, const char* fieldData, short Formflags, short Annotflags)`
COM	`InputSetFormData(Fieldname As String, Data As String, Ff As PTFormFlag, Af as PTAnnotFlag) As Boolean`

Java	`void inputSetFormData(String Fieldname, String Data)`
	`void inputSetFormData(String Fieldname, String Data, boolean noRO)`
	`void inputSetFormData(String Fieldname, String Data, short Formflags, short Annotflags)`

Use this to populate the text fields of an input file with data, after opening the form template using *PDocInputOpen* and before calling *PDocInputCopyPages* to generate the output containing the new data.

This method is called in the context of the output file, because the data is not actually set in the input file first, but rather added on the fly when the pages are copied to the output file. You will not get an error when specifying an invalid field name or a field name that is not copied, because the page containing the field is not in the range of pages that you specify in *InputCopyPages*.

Note that it is possible to define multiple fields with the same name in Acrobat. All these fields have the data in common, but may differ how they appear (placement, font, alignment, etc.). *PDocInputSetFormData* will set the data in all instances, respecting their individual appearance settings.

The *NoReadOnly* parameter allows you to leave the *ReadOnly* attribute of the fields (use 1). Specifying a value of 0 will set all instances of the field to "read only".

The attributes of the form fields can not be set via the API. Set font, font size, alignment and so on using Acrobat Exchange in the template file.

Note that no text formatting is supported, and only the standard Acrobat fonts can be used (unless the field has been created using *AddTextField* – see below).

Text wrapping will be performed automatically in multi-line fields. You may also supply already formatted data (e. g. for numbers and dates). To explicitly mark a newline in multi-line text, use "\r" (Chr$(13)). With this exception, you must use only printable characters.

You can also print data on a page using output to the header layer. If you want to place a bar code or image on the page, this is the way to do it.

PT allows you to re-use a specific page from the input file as a template that is filled with data and copied to output many times. Please be aware of the fact that you have form fields with identical names but different data in the output file. Once you open this file in Acrobat, a change of the data of one field will affect all other fields with this name. As a precaution, you may thus want to set these fields to read-only.

The value of a check box is set using the export string (checked) or the constant string "Off" (unchecked).

Radio buttons are set by specifying the export string (value of the button to be "on"). "Off" can be used to set all to the "off" state.

9.2 Define a Custom Font

Native	`PTError IDocSetFormFont(InputHandle h, short fontID, const char* Basefont)`
COM	`SetFormFont(FontID As PTFormFontType, ByVal BaseFontName As String) As Boolean`
Java	`void setFormFont(short fontID, String basefont)`

This function defines a custom font that can be used for text form fields. This function only applies to fonts of form fields.

9.3 Get a Font Name

Native	`PTError IDocGetFontName(InputHandle h, short fontID, VBSTR* name)`
COM	`GetFontName(FontID As PTFormFontType) As String`
Java	`String getFontName(short fontID)`

This function returns the name of the base font that corresponds to the specified font number. This function only applies to fonts of form fields.

9.4 Delete a Form Field

Native	`PTError IDocDeleteFormField(InputHandle h, const char* fieldName)`
COM	`DeleteFormField(ByVal Fieldname As String) As Boolean`
Java	`void deleteFormField(String Fieldname)`

This function deletes all instances of a form field from a template file. Note that the enumerator of the function *IDocGetFormData* is affected. When field 1 is deleted, field 2 becomes number 1 etc.

9.5 Add a Text Form Field

Native	`PTError IDocAddTextField(InputHandle h, const char* fieldName, const char* fieldDescr, float box[], int page, short fontID, float fontSize, short alignment, int FormFlags, int AnnotFlags, int borderRGB, int backgroundRGB, int rotate, int textRGB)`
COM	`AddTextField(ByVal FieldName As String, ...) As Boolean`
Java	`void addTextField(String fieldName, ...)`

	`void addTextFieldEX(String fieldName, ...)`

This function adds a text form field to a PDF file. Note that this field is put into the transient memory cache of an input file which cannot be saved as such, but must be copied to an output file. To fill in data into a text field that is added this way, you can use the *SetFormData* method. The name of the text field may not contain a "." (period).

The colors (borderRGB, backgroundRGB) are encoded in the following way:

RGB = RED[0..255] + 256*GREEN[0..255] + 256*256*BLUE[0..255]

or as Hex numbers: RGB = 0xBBGGRR (&H00BBGGRR in Visual Basic)

For example: &H000000FF is red, &H0000FF00 is green, etc. (as in the Visual Basic color settings)

9.6 Copy a Form Field

Native	`PTError PDocAddFieldFromLogo(Handle h, const char* fieldName, int pageNumber, float X, float Y, float Width, float Height, const char* newName)`
COM	`AddFieldFromLogo(ByVal FieldName As String, PageNumber As Long Optional X As Single, Y As Single, Width As Single, Height As Single, NewName As String) As Boolean`

Java	`void addFieldFromLogo(String fieldName, int pageNumber)`
	`void addFieldFromLogo(String fieldname, int pageNumber, float X, float Y, float Width, float Height)`
	`void addFieldFromLogo(String fieldname, int pageNumber, float X, float Y, float Width, float Height, String newName)`

This function copies a form field from an existing PDF document to an output PDF document. The file where the field is taken from is the current logo file (s. *SetLogoFile*). You cannot change anything about the field except the coordinates and the name.

9.7 Form Flattening

Native	`PTError PDocSetFlatten(Handle h, short On, int mode)`
COM	`SetFlatten(ByVal On As Boolean, Mode As PTFlattenMode)`
Java	`void setFlatten(boolean on)` `void setFlatten(Boolean on, int mode)`

This output file setting has the effect that text fields are rendered into the page content during subsequent *InputCopyPages* calls. This means that the form field is eliminated, and its content now constitutes part of the page content.

9.8 Add a Text Annotations

This function adds text annotations to an input handle.

Native	`PTError IDocAddTextAnnotation(InputHandle h, const char* label, const char* content, float rect[X1, Y1, X2, Y2], int page, float color[R, G, B], int annotFlags)`
COM	`AddTextAnnotation(Name As String, Content As String, X As Single, Y As Single, Width As Single, Height As Single, Page As Long, R As Single, G As Single, B As Single, Flags As PTAnnotFlags) As Boolean`
Java	`void addTextAnnotation(String title, String content, PTRectangle location, int colorRGB)` `void addTextAnnotation(String title, String content, PTRectangle location, int colorRGB, int annotFlags)`

The "Name" is the title of the text annotation, the "Content" is the text that is visible when the annotation is opened. The three values for the colors (R: Red, G: Green, B: Blue) are in the range from 0 to 1. The text annotation is closed per default. The default for the annotation flags is *PTFlagAnnotPrintable*.

Native: The four values of the position rectangle mark the lower left corner (X1, Y1) and the upper right corner (X2, Y2), 0, 0 being in the lower left. The parameter of the page number is zero based.

COM: The four values of the position mark the lower left corner (X, Y) and the width and height of the opened text annotation, 0, 0 being in the lower left. The parameter of the page number is non-zero based.

9.9 Delete an Annotation

This function deletes a text annotation.

Native	`PTError IDocDeleteAnnotation(InputHandle h, int Identification)`
COM	`DeleteAnnotation(Identification As Long) As Boolean`
Java	`boolean deleteAnnotation(int id)`

The parameter Identification is the value that can be retrieved using the method *GetAnnotation*.

9.10 Delete Viewer Extension Rights

A PDF document can have so called Viewer Extension Rights, which allow the document to be modified (do form filling) and save it with the Acrobat Reader. Modifying such a document with the Prep Tool Suite will destroy the Viewer Extension Rights. A warning message will therefore appear when the document is opened in Acrobat Reader. This function deletes the Viewer Extension Rights and therefore there will be no warning message when opened in Acrobat Reader.

Native	`int IDocDeleteViewerRights(InputHandle h)`

COM	`DeleteViewerRights() As Boolean`
Java	n.a.

9.11 Add an Image Annotation

This function is only available for the COM interface. It adds an annotation containing an image. It is applied to the output handle. The type of the annotation can be either a standard stamp annotation or a custom stamp annotation. Prior to calling *AddImageAnnotation*, it is required to create an image and get its ID using *CreateImage* or *CreateImageEx*.

Native	n.a.
COM	`{PDoc.}AddImageAnnotation(Page As Long, r1 As Single, r2 As Single, r3 As Single, r4 As Single, ImageID As Long, Optional SubType As Integer)`
Java	n.a.

Parameters:

Page	The page number where the annotation is to be placed.
r1, r2, r3, r4	Positioning parameters in PDF points (left, bottom, width, height), 0/0 at lower left.
ImageID	The image ID which is returned from *CreateImage* or *CreateImageEx*.
SubTypes (optional)	The available sub types are: 0 = Standard Stamp Annotation 1 = CstmStamp Annotation other = Unkown Subtype

9.12 Set the Line Spacing in a Form Field

The spacing between the text lines in a form field can be defined by using the following method.

Native	`void IDocSetFormLineSpacing(InputHandle h, float fLineSpacing)`
COM	`InputSetFormLineSpacing(float LineSpacing)`
Java	`void setFormLineSpacing(float value)`

9.13Get the Name of the Font in a Form Field

The name of the font used in a form field can be queried by using the following method. The value of the id parameter can be retrieved by invoking the **IDocGetFormBox** method.

Native	`PTError IDocGetFontName(InputHandle h, short id, VBSTR* name)`
COM	`GetFontName(PTFormFontType FontID) As String`
Java	`String getFontName(int FontID)`

10 Generate Output

10.1 Create Another Page

Native	`PTError PDocNewPage(Handle h)`
COM	`NewPage()`
Java	`void newPage()`

A page is automatically create when you access the page object – either at the beginning of a file, or after you copied pages from an input file. If you need a new page (i.e. a page brake), you call the *NewPage* function. The dimensions of the page are inherited from the last setting made (*PDocNew*, *PDocPageSize*).

10.2 Copy Pages from the Input File

Native	`PTError PDocInputCopyPages(Handle h, int firstPage, int lastPage)` `PTError PDocMerge(Handle h, const char* inputFile, int firstPage, int lastPage)`
COM	`InputCopyPages(FirstPage As Long, LastPage As Long) As Boolean` `Merge(FileName As String, FirstPage As Integer, LastPage As Integer) As Boolean`
Java	`void inputCopyPages(int FirstPage, int LastPage)` `void merge(String Filename, int Firstpage, int Lastpage)`

InputCopyPages and *Merge* will copy the specified range of pages from the currently open input file to output. If the file contains form fields, the field data will be set according to previous *PDocInputSetFormData* calls. You can repeatedly call *CopyPages* and change the header in between.

The pages being copied can be modified not only by setting form data prior to *InputCopyPages*, but also by setting the new Rotate (page orientation) value (s. *SetInputRotate*).

Note that you should copy all pages containing forms of an input file. Leaving away a page with form fields will result in orphan entries; duplication of pages works for viewing the resulting file, but Acrobat 4.0 may behave unexpectedly if you want to modify a form field of a duplicated page.

Merge (add) pages from an existing PDF file into the output document. The range of pages to be added is specified using the parameters *firstPage* and *lastPage*. The current *Header* will be placed on all of these pages. *PDocMerge* works like *PDocInputOpen* followed by *PDocInputCopyPages*.

PDocMerge returns FALSE (0), if the input file cannot be processed. *PDocMerge* will automatically close the current input file, if one exists.

Be careful with repeated merging of PDF files. The font alias used for the header text is determined before the file to be merged is known. Repeated merging in combination

with placing header text will result in a name conflict. The alias used by Prep Tool is "/FHdr". You can avoid this problem by using *PDocInputOpen* before setting the font, and then copy the pages using *PDocInputCopyPages*. A potential conflict still remains when you add further PDF files while keeping the header with its font.

10.3Copy Color Spaces from the Input File

Native	`PTError PDocInputCopyColor(Handle h, const char* Color)`
COM	`InputCopyColor(ByVal Color As String) As Boolean`
Java	`void inputCopyColor(String Color)`

Use this function to copy a color space object from the current input file to the output file. This feature is useful if you want to use Pantone colors: store the set of colors you need in a PDF file, and use this file at runtime to provide the necessary color definition objects. This function works in conjunction with the *PConSetFillCS* and *PConSetLineCS* functions.

10.4Copy Named Destinations from the Input File

Native	`PTError PDocInputCopyDestNames(Handle h)`
COM	`InputCopyDestNames()`
Java	`void inputCopyDestNames()`

This function will copy all named destination entries from the input to the output file. A situation where this makes sense is when you have bookmarks and links that are also to be copied. If you do not copy the named destinations, the bookmarks and links will work, but loose the zoom level, because resolution only works on the page level. Not copying the named destinations will save space in the resulting file.

10.5Copy Custom Objects from the Input File

Native	`PTErrpor PDocInputCopyCustObjs(Handle h)`
COM	`InputCopyCustomObjs()`
Java	`void inputCopyCustomObjs()`

The input file may contain entries in the Catalog object that are not taken care of by any of the existing copy functions. To copy these entries along with any referenced objects, you can use this method. Note that it will copy e. g. also viewer settings or open actions, if such settings are not specified explicitly for the output documents and if present in the input file. This function can be used to merge JavaScript resources, it returns the error *pdAlreadyWritten* when duplicate JavaScript names are encountered.

10.6Copy All Objects from the Input File

Native	`PTError PDocInputCopyAll(Handle h)`

COM	`InputCopyAll() As Boolean`
Java	`void inputCopyAll()`

This method is equivalent to *InputCopyPages* (for all pages), *InputCopyDestNames*, *InputCopyBookmarks*, and *InputCopyCustomObjs*. In other words, it copies the whole file content from input to output.

10.7Import Bitmap Images

Native	`short PDocCreateImage(Handle h, int Width, int Height, short bits, short color, char* imageData, int imageSize, char* palette, short compression, char* mask)`

COM	`CreateImage(ByVal Width As Long, ByVal Height As Long, ByVal Bits As Integer, ByVal Color As Boolean, ByVal ImageData As Byte(), Optional ByVal Palette As Byte(), Optional ByVal IsJPEG as Boolean, Optional Mask As Byte(), Optional Softmask As Byte()) As Long`
	
	`CreateImageEx (ByVal Width As Long, ByVal Height As Long, ByVal Bits As Integer, ByVal Color As Boolean, ByVal ImageData As Byte(), Optional ByVal Palette As Byte(), Optional Mask As Byte(), Optional CompressionType As Integer) As Long`

Java	`int createImage(int Width, int Height, short bits, boolean color, byte[] image, byte[] palette)`
	
	`int createJPEGImage(int Width, int Height, ..)`
	
	`int createImage(int Width, int Height, short bits, boolean color, byte[] image, byte[] palette, boolean isJPEG )`
	
	`int createImage(int w, int h, short bits, boolean color, byte[] image, byte[] palette, byte[] mask, int image_type)`
	
	`final static image_type_standard = 0`
	
	`final static image_type_JPEG = 1`

The *CreateImage* function creates an image XObject according to the data provided. The format of the data must correspond to one of the PDF standards for color, grayscale or bi-level images. Color images have a palette. The palette size in PDF must be 768. If the effective palette is smaller, the unused part must be set to zero in the native interface. The COM interface will automatically handle smaller palette sizes.

A positive number value returned by *CreateImage* identifies the XObject for reference in *PrintImage* calls. A value of zero indicates failure to create the image.

10.8Add Page Numbers

Native	`PTError PDocPutPageNumbers(Handle h, float X, float Y, const char* Format, long StartPage, short Orientation)`
COM	`PutPageNumbers(Format As String, X As Single, Y As Single,`

	`Startpage As Long, Orientation As Integer)`
Java	`void putPageNumbers(String Format, float X, float Y, int StartPage, int Orientation)`

Expand the page marker "%p" in the format string to reflect the current page number and put this string on each page just like other header text. With the *firstPgNr* parameter, you specify where page numbers should start. The page numbering string will appear on each header that is displayed, starting with the next *PDocMerge*. The first time the page numbering string is displayed, it will carry the page number specified in *firstPgNr*. Note that the header or the page numbering string may be created after having copied some pages to the output file. Any previously set format string will be removed. To stop putting page numbers on a page, you can thus call this function with an empty format string.

The font used for the page number text is the same as for the header. Do not forget to specify a font. If you work with different font sizes, then the last setting will be the one used for the page number string, e. g. if *PDocHeaderFont* was called after *PDocHeaderPgInfo*. The font of page numbers is unpredictable if you do not have a header layer!

Example: PDocHeaderPgInfo(h, 10, 10, "Page %p of 10", 2);

10.9Change the Header or Background

Native	`PTError PDocHeaderClear(Handle h)` `PTError PdocBackgroundClear(Handle h)`
COM	`HeaderClear()` `BackgroundClear()`
Java	`void clearHeader()` `void clearBackground()`

When you want to change the text, graphics objects or logo added to merged pages, call *PDocHeaderClear* and build the new header as desired. You cannot continue to add anything (text, graphics, logos) to a header stream, once it is applied to pages (i.e. after calling *CopyPages* or *Merge*). This is because the header stream is written to the output file at this point, and any changes that you make after that are ignored.

After calling *PDocHeaderClear*, you need to set the font for header text again. It is possible to re-use the last font imported from an input file by specifying the same name again.

10.10Add Bookmarks

Native	`PTError PDocInputCopyBookmarks(Handle h, int level)`
	`PTError PDocAddWebBookmark(Handle h, int level, const char* title, const char* URL, int kidsVisible)`
	`PTError PDocAddGoToBookmark(Handle h, int level, const char* title, int page, short X, short Y, int kidsVisible, float zoom)`

	`PTError PDocAddGoToBookmarkU(Handle h, int level, const PDBSTR, int page, short X, short Y, int kidsVisible, float zoom)`
	`PTError PDocAddGoToRBookmark(Handle h, int level, const char* title, const char *destFile, int destPage, short X, short Y, int kidsVisible, float zoom)`
	`PTError PDocAddGoToRBookmarkU(Handle h, int level, const PDBSTR title, const char *destFile, int destPage, short X, short Y, int kidsVisible, float zoom)`
	`PTError PDocAddOpenFileBookmark(Handle h, int level, const char* title, const char* destFile, int kidsVisible)`
	`PTError PDocAddOpenFileBookmarkU(Handle h, int level, const PDBSTR title, const char* destFile, int kidsVisible)`
	`PTError PDocAddNullBookmark(Handle h, int level, const char* title, int kidsVisible)`
	`PTError PDocAddNullBookmarkU(Handle h, int level, const PDBSTR title, int kidsVisible)`
	`PTError PDocAddJavaScriptBookmark(Handle h, int level, const char* title, const char* script, int kidsVisible)`
	`PTError PDocAddJavaScriptBookmarkU(Handle h, int level, const PDBSTR title, const char* script, int kidsVisible)`

COM	`AddWebBookmark(Title As String, Level As Integer, URL As String, ShowKits As Boolean)` etc.
Java	`void addWebBookmark(String Title, int Level, String URL)` etc.

All these methods have optionally a further parameter for specifying that bookmarks on lower levels shall be visible.

The bookmark tree of the output file can be constructed using the above functions. The first bookmark must be placed on level zero. Subsequent bookmarks can be placed at most one level above the previous level.

An URL is something like "http://www.pdf-tools.com", but it is also possible to put relative links like "../index.html".

"GoTo" targets are pages in the same document as the one being created. “page” is the page number (starting at 1). The "x" and "y" parameters can be used to set the view window according to the /XYZ entry in link annotations (see PDF specification). Specify zero values to disable this feature. The "z" parameter describes the zoom value. 1 stands for 100%, 1.1 for 110%, etc, 0 for keep the current zoom value.

"GoToR" targets are "remote" links, i. e. links to another PDF file (you will note the extra file name parameter).

"OpenFile" targets are files that represent a document or application that is to be launched. Document files are opened with the application that is registered for the document type. On Windows systems, the file extension is used for this.

A Java Script added to a bookmark will be executed when the bookmark is selected.

If bookmarks refer to named destinations, they will be resolved to avoid conflicts between names in different files.

10.11Add Links

Native	`PTError PDocAddWebLink(Handle h, int page, const float rect[], const char* URL, int style)`
	`PTError PDocAddGoToLink(Handle h, int page, const float rect[], int destPage, short x, short y, int style, float zoom)`
	`PTError PDocAddGoToRLink(Handle h, int page, const float rect[], const char* destFile, int destPage, short X, short Y, int style, float zoom)`
	`PTError PDocAddJavaScriptLink(Handle h, int page, const float rect[], const char* script, int style)`
	`PTError PdocAddNamedDestLink(Handle h, int page, const float rect[], int style, const char* destName)`

COM	`AddWebLink(Page As Long, Left As Single, Bottom As Single, Right As Single, ByVal URL As String, Optional Style)` etc.
Java	`void addWebLink(PTRectangle Rect, String URL)` etc.

The action behavior of links corresponds to that of bookmarks. Links are located as an annotation on a page. Therefore, you need to specify the page number and coordinate rectangle where to put the link instead of the hierarchy level in the bookmark tree.

The Prep Tool Suite supports several border styles; the value –1 will suppress the border, 0 will result in a solid black border, 1 is dotted red, 2 red solid, 3 green dashed, 4 green solid, 5 blue dashed, 6 blue solid.

10.12Add File Attachments

Native	`PTError AddFileAttachment(Handle h, int page, float* rect, const char* filepath, const char* icontype, const char* description, const char* author, const char* subject, int rgb, int opacity)`
COM	`AddFileAttachment(Page As Long, Left As Single, Bottom As Single, Right As Single, Top As Single, Filepath As String, IconType As String, Description As String, Optional Author As String, Optional Subject As String, Optional ColorRGB As Long, Optional Opacity As Long)`
Java	`void addFileAttachment(PTRectangle rect, InputStream is, String iconType, String description, String author, String subject, int rgb, int opacity)`

This function adds a file attachment annotation to a PDF file.

Parameters:

- page: the page number (first page is 1)
- Left, Bottom, Right, Top: the annotation's rectangle on the page
- Filepath: the name of the file to be attached
- IconType: the icon's type name ("PushPin","Graph","Paperclip", or "Tag")
- Description: the description field (used as default for the file name when extracting the attachment)
- Author: the author field (optional)
- Subject: the subject field(optional)
- ColorRGB: the RGB value of the color for the icon
- Opacity: the opacity value in percent (0..100); 0 means transparent; 100 means opaque (default)

Note that each standard icon type has its specific rectangle width and height in the Acrobat viewer. Setting other values has the effect that Acrobat viewers will change the appearance when clicking on the icon.

Paperclip size: 7/17 Graph size: 20/20

PushPin size: 14/20 Tag size: 20/16

Implementation restriction:

Creation of the icon appearance stream is not supported when Using opacity less than 100. Acrobat viewers will correctly display these icons, but third party viewers that depend on the appearance stream may not show the icon.

10.13Add Destination

Native	`PTError AddGotoDestination(Handle h, const char* name, int page,` `short X, short Y, float Z)`
COM	`AddGotoDestination(Name As String, Page As Long, X As Integer, Y As Integer, Z As Single) As Boolean`
Java	`void addGotoDestination(String name, int page)` `void addGotoDestination(String name, int page, int X, int Y, float zoom)`

This function adds a Named Destination to the document. A named destination points to a certain location (e.g. the beginning of a chapter) in the PDF. The location is defined by the page number and the X, Y and Z (Zoom) position.

10.14Set Document Action

Native	`PTError PDocSetDocumentAction(Handle h, PTDocumentAction documentaction, const char* Script)`
COM	`SetDocumentAction(ActionType As PTDocumentActionType, Script As String) As Boolean`

Java	`boolean setDocumentAction(int DocumentAction, String Script)`

This function adds a JavaScript to a document action. The five document actions are: 0 on close, 1 before save, 2 after save, 3 before print and 4 after print.

10.15 Set Form Fontsize Range

Native	`PTError PDocSetFormFontSizeRange(Handle h, float Max, float Min)`
COM	`SetFormFontSizeRange(Max As Single, As Single)`
Java	`boolean setFormFontSizeRange(float Max, float Min)`

With SetFormFontSizeRange it is possible to limit the font sizes for auto-sized form fields. The default values are Max = 12 and Min = 5.

10.16 Document Open Settings

The following group of functions facilitates the setting of the page layout and mode and how the first page shall be displayed when opening the document in a viewer (as offered in the "Document Info"-> "Open.. " dialogue of Acrobat).

Native	`PTError PDocSetPageMode(Handle h, const char* Mode)`
COM	`SetPageMode(ByVal Mode As String)`
Java	`void setPageMode(String Mode)`

The page modes currently supported by Acrobat viewers are "UseNone", "UseOutlines", "UseThumbs", and "/FullScreen". The *SetPageMode* function will override any settings from input files that would otherwise be copied during *InputCopyCustomObjs*.

Native	`PTError PDocSetPageLayout(Handle h, const char* Layout)`
COM	`SetPageLayout(ByVal Layout As String)`
Java	`void setPageLayout(String Layout)`

The following layouts can be specified: "SinglePage", "OneColumn", "TwoColumnLeft", "TwoColumnRight".

Native	`PTError PDocSetOpenAction(Handle h, int Page, const char* Magnification)`
COM	`SetOpenAction(ByVal Page As Long, ByVal Magnification As String)`
Java	`void setOpenAction(int Page, String Magnification)`

The page to be shown initially when a file is opened can be specified using this function. At the same time, the zoom factor or type of "fit" can be specified. Legal values for page numbers are 1 through the number of pages that the file contains; the magnification can be a positive integer number representing the zoom factor in percent (100 = normal 100% zoom). The minimum and the maximum is viewer dependent (currently 25 – 1600). Other legal "magnifications" are "Window" for "Fit Window", "Width" for "Fit Width", and "Visible" for "Fit Visible". Any other value will be mapped to "Default".

Native	`PTError PDocClearViewerPreferences(Handle h)` `PTError PDocAddViewerPreference(Handle h, const char* Key, const char* Value)`
COM	`AddViewerPreference(ByVal Key As String, ByVal Value As String, Optional ClearExisting As Boolean)`
Java	`void clearViewerPreferences()` `void addViewerPreference(String Key, String Value)`

The viewer preferences entries can be created (or suppressed) by these functions. For a complete listing of all possible settings, please refer to the PDF specifications.

Viewer preferences are stored in a dictionary. The *AddViewerPreferences* function adds a pair of values consisting of the dictionary key and its associated value. Examples are "/HideToolbar true", "/FitWindow true", "/CenterWindow true", "/NonFullScreenPageMode /UseThumbs".

10.17 Set Document Information Attributes

Several document attribute values can be set via the following methods. Note that the value string will be re-encoded from WinAnsiEncoding to PDFEncoding (see Adobe PDF Reference Manual). This means that only characters existing in both encodings may be contained.

Native	`void PDocSetInfo(Handle h, const char* Title, const char* Subject, const char* Author, const char* Keywords)`
COM	`SetInfo(ByVal Title As String, ByVal Subject As String, ByVal Author As String, ByVal Keywords As String)`
Java	`void setInfo(String Title, string Subject, string Author, String Keywords)`

SetInfo allows you to set some of the document attributes in the information object.

Native	`void PDocSetAttr(Handle h, const char* Key, const char* Value)` `void PDocSetAttrU(Handle h, const char* Key, const PDBSTR Value)`
COM	`SetAttr(ByVal Key As String, ByVal Value As String)`
Java	`void setInfoAttr(String Key, String Value)`

SetAttr permits to set (and add) any value in the information object of the PDF file.

10.18 Set Document Metadata

Native	`PTError PDocSetMetaData(Handle h, const char* data)`
COM	`SetMetaData(Data As String)`
Java	`void setMetaData(String data)`

SetMetaData sets the meta data in the root object of the PDF file. Note that the data string should constitute a valid XML expression.

10.19 Close the Output File

Native	`PTError PDocClose(Handle h)` `PTError PDocRelease(Handle h)`
COM	`Close() As Boolean`
Java	`void close()`

Close the output document. This procedure writes out any pending output and closes the file.

PDocRelease releases the handle and all memory resources associated with it. No further calls are allowed with this handle.

If you want to verify that the file has been successfully closed, you first want to call *PDocClose*, and then *PDocRelease*. If *PDocClose* fails, you can still use the handle to retrieve error information.

In the Java binding, the close method also releases the associated resources. If an error occurs during the close operation, an exception is signaled carrying the error code.

In the COM binding, releasing the last object reference will automatically close the file and release all associated resources.

To retrieve the bytes of a memory resident PDF file, use the following functions:

Native	`VBSTR PDocCloseB(Handle h, int* length)`
COM	`bytes = CloseB()`
Java	`void close()` `byte[] getBytes()`

The *CloseB* functions perform a normal close and return the bytes of the memory resident PDF file. Note that the memory buffer of the file is disposed on close. The memory buffer returned by these functions must be freed by the application.

In Java, the byte array is remains stored with the Java wrapper object and can be multiply accessed through *getBytes()* (until the Java object is "finalized").

10.20 Set the license key at runtime

Set the license key programmatically at runtime instead of installing it on the system.

Native	`int PTSetLicenseKey(const char* szLicenseKey)`
COM	`SetLicenseKey(bstrLicenseKey As String) As Boolean`
Java	`boolean setLicenseKey(String szLicenseKey)`

Parameters: The license key

Return value: True: The license key is valid.

Check whether a valid license key has been installed in the system or passed at runtime.

Native	`int PTGetLicenseIsValid()`
COM	`LicenseIsValid() As Boolean`
Java	`boolean getLicenseIsValid()`

Return value: True: A valid license was found.

11 Linearization

Linearization is the processing performed on a PDF file to optimize it for viewing in a web browser. The elements of the PDF file are regrouped, so that all information necessary to display the first page is located at the beginning of the file. Furthermore, information about file offsets is stored in the header of the file and in the so called hint tables.

Due to the nature of linearization, this process can begin only when a PDF file is created completely. The functions supporting linearization are thus separate from other PDF Prep Tool functions.

Native	**PTError PDLinearize(const char* Input, int Length, const char* InputPassword, const char* OutputFileName, const char* OwnerPassword, const char* UserPassword, const char* Permissions)** **PTError PDLinearizeMem(const char* Input, int Length, const char* InputPassword, VBSTR* OutputBuffer, int* OutputLength, const char* OwnerPassword, const char* UserPassword, const char* outPermissions)**

The native interface offers just these two functions. Input can be provided either as the file name of the input file when specifying a length of 0, or as the memory buffer containing the PDF "file" along with the length of that buffer.

The first function writes the linearized PDF to a file, while the second returns it in a memory buffer. This memory buffer must be freed using PTFreeVBSTR.

The return result of these function is a *PTError*.

COM	**Dim tool As New PDFLinearizer** **Dim tool As Object** **Set tool = CreateObject("PrepTool.PDFLinearizer")** **SetSecurity(ByVal OwnerPassword As String, ByVal UserPassword As String, ByVal Permissions As String)** **OpenInput(ByVal Filename As String, Optional ByVal Password As String) As ErrorType** **OpenMem(ByVal PDFBytes As Variant, Optional ByVal Password As String) As ErrorType** **SaveFile(ByVal Filename As String) As ErrorType** **SaveMem(Optional Result As ErrorType) As Variant**

The COM interface for linearizing PDF files is also quite straight forward. A call of *SetSecurity* is optional and will only be used if the resulting file shall be encrypted.

The COM object can be reused for several linearizations. As the input file resources will be freed on *SaveFile* or *SaveMem*, it is necessary to re-open a file before linearization can be performed again during one of the "Save.. " functions. Password and permission settings are preserved.

Java	`PTLinearizer tool = new ..`
	`  PTLinearizer(String Filename, String Password)`
	`  PTLinearizer(String Filename)`
	`  PTLinearizer(byte[] PDFBytes, String Password)`
	`  PTLinearizer(byte[] PDFBytes)`
	`void setSecurity(String Ownerpassword, String Userpassword, String Flags)`
	`byte[] getLinearizedBytes()`
	`void doLinearization(String Filename)`

The Java API is similar to the COM interface with the difference that no reuse of the *PTLinearizer* object is permitted. Once linearization has been performed, all resources are freed.

12 Return Codes C

0	Success	
1001	PTNotPDF	the file does not start with %PDF
1002	PTTrailer	the trailer of the PDF file could not be found
1003	PTXref	the XRef table could not be found as defined in trailer these two errors indicate that the PDF file has been corrupted as sometimes happens when copied in ASCII mode by FTP
1004	PTNullRef	an object reference could not be resolved (object missing in file)
1005	PTBadParamValue	an illegal parameter value was specified in a method
1006	PTObjRead	a particular PDF object could not be read from the file
1007	PTAlreadyWritten	a particular PDF object was attempted to write twice
1008	PTBadCallSequence	a particular function was called in an inappropriate context
1009	PTInternal	an unexpected situation was encountered that could not be handled
1010	PTUnexpectedVal	an unexpected value was encountered in a PDF object
1011	PTIO	an input/output error was encountered
1012	PTInvalidHandle	the handle specified is not valid
1013	PTDuplicate	an attempt to create a duplicate object is made
1014	PTIllegalFont	an invalid font name was specified
1015	PTNoSuchPage	an invalid page number was specified
1016	PTNotFound	requested information not found for specified criteria
1017	PTFailed	License key invalid or generic error
1018	PTEncrypted	input file is encrypted (password protected)
1019	PTInvalidPassword	the password supplied is not correct