DynaPDF 4.0

Reference & Manual

API Reference Version 4.0.87

April 12, 2024

Legal Notices

DynaForms GmbH
Bokel 3
D-49626 Bippen, Germany
Trade Register HRB 218421, District Court Osnabrueck
CEO Jens Boschulte
Phone: ++49 5435-955 3226
Fax: ++49 5435-956 7922

If you have questions please send an email to info@dynaforms.com, or contact us by phone.

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

Adobe, Acrobat, and PostScript are trademarks of Adobe Systems Inc. AIX, IBM, and OS/390, are trademarks of International Business Machines Corporation. Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation. Apple, Mac OS, and Safari are trademarks of Apple Computer, Inc. registered in the United States and other countries. TrueType is a trademark of Apple Computer, Inc. Unicode and the Unicode logo are trademarks of Unicode, Inc. UNIX is a trademark of The Open Group. Solaris is a trademark of Sun Microsystems, Inc. Tru64 is a trademark of Hewlett-Packard. Linux is a trademark of Linus Torvalds. Other company product and service names may be trademarks or service marks of others.

DynaPDF uses parts or modified versions of the following third-party software:

AiCrypto. This product includes software developed by Akira Iwata Laboratory, Nagoya Institute of Technology in Japan (http://mars.elcom.nitech.ac.jp/).

Antigrain Geometry Version 2.4. Copyright (C) 2002-2005 Maxim Shemanarev (McSeem).

Clipper, Copyright 2010-2012 by Angus Johnson.

Expat, Copyright 1998-2000 Thai Open Source Software Center Ltd and Clark Cooper.

FreeType2, Copyright 2006-2018 by David Turner, Robert Wilhelm, and Werner Lemberg.

LibJPEG, Independent JPEG Group's JPEG Software, Copyright 1991-1998 Thomas G. Lane.

Little CMS Copyright (c) 1998-2011 Marti Maria Saguer

Table of Contents

Data types

DynaPDF is a low level library that uses basic data types only. DynaPDF uses generally no default string class or special C++ extensions such as STL or MFC.

The data types used by DynaPDF are defined as follows (C syntax):

```c
typedef unsigned char  BYTE;
typedef signed short   SI16;
typedef unsigned short UI16;
typedef double         FFLOAT; // Obsolete, do not use this data type.

#ifdef _WINDOWS
   #if defined(WIN64) || defined(_WIN64)
      typedef int           SI32;
      typedef unsigned int  UI32;
   #else
      typedef long          SI32;
      typedef unsigned long UI32;
   #endif
#elif (SIZEOF_INT == 4) // declared in drv_conf.h (Linux/UNIX only)
   typedef int           SI32;
   typedef unsigned int  UI32;
#elif (SIZEOF_LONG == 4) // declared in drv_conf.h (Linux/UNIX only)
   typedef long          SI32;
   typedef unsigned long UI32;
#else
   #error "Only 32 bit and 64 bit targets are supported!"
#endif

typedef SI32 LBOOL; // long boolean (0 = false, not 0 = true)
// This data type is provided for C only since this language does not
// support a Boolean data type.
#ifndef __cplusplus
   typedef enum
   {
      false = 0,
      true = 1
   }bool;
#endif
```

The data type char is not explicitly defined but also used by DynaPDF.

SI32 and UI32 must always be a 32 bit integer. It is not possible to use DynaPDF on a target system that does not support a 32 bit integer type, like MS-DOS. DynaPDF can be used on 32 and 64 bit operating systems. The current version was tested with Android, IBM-AIX, HP-UX, iOS, Linux, Mac OS X, Sun-Solaris, Tru64, and MS-Windows.

With very few exceptions, string values returned by DynaPDF are always null-terminated. A string length returned by DynaPDF is always the length excluding the null-terminator.

Var Parameters

```
#ifdef __cplusplus
    #define ADDR &
#else
    #define ADDR *
#endif
```

Functions in DynaPDF, which pass a value to a function parameter are handled differently in C and C++. C does not support the address operator & so that *var* parameters are defined as normal pointers to pointers in C. DynaPDF checks whether a variable or NULL was passed to a function before the function tries to access the variable. However, C++ does not allow to set a parameter to NULL if it was declared with the address operator &.

Structures

Beginning with DynaPDF 2.5 all structures which can be extended in future versions contain the member **StructSize**. This variable must be set to sizeof(StructureName). The structure size is used to identify the version of a structure so that extensions do not break backward compatibility.

The structure size is automatically set in interfaces for C#, Visual Basic, Visual Basic .Net, and Delphi. C/C++ programmers must set this member before the corresponding function can be called:

Example:

```
...
TPDFCMap cmap;
cmap.StructSize = sizeof(TPDFCMap);
pdfGetCMap(pdf, handle, &cmap);
...
```

Multi-byte Strings

Unicode

DynaPDF supports Unicode strings in UTF-16-LE format on little-endian machines and UTF-16-BE on big-endian machines. On target systems which use UTF-32 (LE or BE) as default string format such as Linux or most UNIX OS, all strings must be converted to UTF-16 before passing to DynaPDF.

You can use the predefined macro ToUTF16 to do this.

ToUTF16 is defined as follows:

```
// declared in drv_conf.h (Linux/UNIX, Mac OS X)
#if (SIZEOF_WCHAR_T == 4)
   #define ToUTF16(IPDF, s)(pdfUTF32ToUTF16((IPDF), (UI32*)(s)))
#else                        // UTF-16
   #define ToUTF16(IPDF, s)((s))
#endif
```

This macro calls pdfUTF32ToUTF16() only if the OS uses UTF-32 as Unicode string format.

On operating systems which use already UTF-16, no conversion is applied; the macro will be removed by the compiler. The function pdfUTF32ToUTF16() holds an array of 4 independent string buffers so that the macro can be used in functions which support up to four string parameters. If DynaPDF will ever support a function with more than 4 string parameters, the number of internal string buffers will be incremented.

However, take care when using the macro to initialize string variables of structures which contain more than 6 string members:

Example:

```
SOME_STRUCT myStruct;
myStruct.String1 = ToUTF16(pdf, L"String1"); // OK
myStruct.String2 = ToUTF16(pdf, L"String2"); // OK
myStruct.String3 = ToUTF16(pdf, L"String3"); // OK
myStruct.String4 = ToUTF16(pdf, L"String4"); // OK
myStruct.String5 = ToUTF16(pdf, L"String5"); // OK
myStruct.String6 = ToUTF16(pdf, L"String6"); // OK
myStruct.String7 = ToUTF16(pdf, L"String7"); // Wrong!
```

The seventh call above overrides the string buffer of String1 because only 6 internal string buffers are available. If you need to store more than 6 string variables then you must copy the converted string into another variable!

Unicode File Paths

Unicode file paths are encoded differently depending on the used operating system. While NT based Windows system use UTF-16 encoded Unicode file paths, non-Windows systems use usually UTF-8 encoded Unicode file paths. All DynaPDF functions which open a file convert UTF-16 strings to UTF-8 on non-Windows operating systems. However, to avoid this conversion step it is usually best to use directly the Ansi version of a function and passing an UTF-8 file path to it.

CJK Multi-byte Strings

CJK multi-byte strings contain mixed 8 bit / 16 bit character codes. A CJK string can be defined as an Ansi string (data type char*) and as multi-byte string (data type UI16*). The multi-byte format

uses two bytes for every character and the byte ordering of the CPU must be considered to get correct results on little-endian and big-endian machines.

However, the multi-byte format is only supported in combination with native CJK fonts and character sets (cpBig5, cpShiftJIS, cpGB2312 and so on), see SetFont() for further information.

The Ansi format is the usual format for CJK strings and supported by all CJK code pages.

When using CJK to Unicode code pages, DynaPDF must convert the incoming CJK string to Unicode before it can be used with the selected font. The required conversion algorithms are only available in the Ansi version of a string function. Because of this it is not possible to use the multi-byte format with CJK to Unicode code pages.

Data types used by different programming languages

Not all programming languages support all data types which are available in C or C++. The following table describes the data types which are used by a specific programming language. Types in black color are not natively supported.

C type	C++ type	Delphi type	VB type	VB .Net type	C#
char*	char*	PAnsiChar	String	String	String
UI16*	UI16*	PWideChar	String	String	String
UI32*	UI32*	Pointer	N/A	N/A	N/A
SI32	SI32	Integer	Long	Integer	int
UI32	UI32	Cardinal	Long	Integer	int
double	double	Double	Double	Double	double
LBOOL	LBOOL	LongBool	Long	Integer	lbool (Int32)
bool	bool	Boolean	Boolean	Boolean	bool
void*	void*	Pointer	Long	IntPtr	IntPtr

VB .Net and C# support also unsigned data types such as unsigned integer and so on. However, the .Net framework requires always an explicit conversion if an unsigned type should be passed to a signed type or vice versa. Because of this, the unsigned data types are used for very few functions in VB .Net and C#.

Prior versions of DynaPDF used the abbreviations LONG, ULONG, SHORT, and USHORT in the C/C++ interfaces. For compatibility reasons these data types are still defined on 32 bit Windows, Linux, and UNIX. However, these declarations conflict with existing declarations on 64 bit Windows operating systems so that these types should no longer be used.

Exception handling

The DynaPDF library uses no native exception handling. It is not required to leave a function block after an error occurred. All DynaPDF functions are leaved immediately when a fatal error occurred without displaying further error messages.

The library always holds its internal state consistent, regardless what happens.

Exception handling in C, C++, C#, Delphi

Error messages and warnings are passed to a callback function (see SetOnErrorProc()) if set. If no callback function is set, you must check the return value of important functions. Negative return values indicate by default that an error occurred. Call GetErrorMessage() to get the last error message in this case.

The Delphi interface uses native language exceptions in the following reasons:

- When loading the dynapdf.dll (Cannot find dynapdf.dll)
- When loading a DynaPDF function (Cannot find function: …)
- When creating a new instance of the wrapper class TPDF (Out of memory).

Only these three exceptions can occur when using DynaPDF with Delphi. All other errors do not raise an exception; the error callback function is called instead if any.

The error callback function is defined as follows (C/C++):

```
typedef SI32 PDF_CALL TErrorProc(
   const void* Data,        // User defined pointer
   SI32 ErrCode,            // Error code
   const char* ErrMessage,  // Error message
   SI32 ErrType)            // Error type

#define PDF_CALL __stdcall // Windows only, otherwise empty
```

All callback functions contain a parameter named "Data" which holds a user defined pointer. *Data* can be set with SetOnErrorProc(). If you don't need this pointer, set it to NULL. The pointer *Data* is always passed unchanged to the callback function.

ErrCode is a positive error number starting at zero. *ErrType* is a bit mask to determine what kind of error occurred. The following constants are defined:

```
#define E_WARNING       0x02000000
#define E_SYNTAX_ERROR  0x04000000
#define E_VALUE_ERROR   0x08000000
#define E_FONT_ERROR    0x10000000
#define E_FATAL_ERROR   0x20000000
#define E_FILE_ERROR    0x40000000
```

At time of publication only one flag is set at any one time. Future versions maybe set multiple flags, e.g. E_SYNTAX_ERROR and E_WARNING.

Because of this, it is required to mask out the error type:

```
if (ErrType & E_SYNTAX_ERROR)
{
   // some code
   return 0; // continue processing
}
```

If the error callback function returns a value other than zero, processing stops immediately.

All callback functions must use the correct calling convention. The C definition above contains the macro PDF_CALL that is defined as __stdcall under Windows. Note that a wrongly defined calling convention causes an access violation!

When using DynaPDF with C or C++, you can change the calling convention to __fastcall or __cdecl by changing the macro PDF_CALL in the main interface dynapdf.h and in the project settings. However, standard call is strongly recommended. Note that the library must be recompiled when changing the calling convention. Delphi, Visual Basic, and VB .Net require standard call!

Exception handling in Visual Basic, Visual Basic .Net

The Visual Basic wrapper class CPDF uses events instead of callback functions to pass error messages and warnings to the user; native language exceptions are not used.

The usage of the event functions is quite easy; just declare a local instance variable of the wrapper class CPDF in the Option Explicit section of the unit as follows:

```
Private WithEvents FPDF As CPDF
```

The command above enables the event support of the class CPDF. The instance variable FPDF is now listed in the left combo box of the VB code editor.
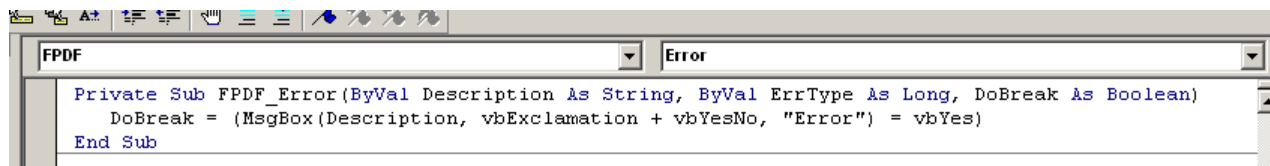
The right combo box contains the available events, when selecting the event "Error" VB adds automatically an empty event procedure to your source code:

```
Private Sub FPDF_Error(ByVal Description As String, ByVal ErrType As
Long, DoBreak As Boolean)
   ' Add your code here
End Sub
```

Now you can enter some code that should be executed when the event is fired. Note that you must still create an instance of the class CPDF before a DynaPDF function can be executed (see Language bindings/Visual Basic for further information).

The handling of events in VB .Net is exactly the same as in VB 6.0.

Special issues in Visual Basic and .Net

The usage of events in Visual Basic or VB .Net is quite easy; however, there is a special behaviour that must be taken into account when developing VB applications. When using the DoEvents procedure in a VB function you must make sure that the function cannot be executed again while a previous call of the function is still running.

DoEvents enables the asynchronous processing of the message loop so that the user interface can be updated and the user can execute something while a function is running (e.g. break processing). DoEvents is often used because it is an easy way to avoid blocking of an application without using of threads.

However, when using DoEvents it is possible that a user clicks on the button again that executes DynaPDF functions while a previous call is still running. This is normally no problem but it is impossible to execute an event function inside of a cloned function. When DynaPDF tries to raise an event inside the cloned function an access violation occurs and VB crashes. VB .Net does not crash but raises a System.NullReference exception in that case.

To avoid such problems check whether the function is still running:

```
Option Explicit
Private WithEvents FPDF As CPDF 'Enable event support
Private FRunning As Boolean

Private Sub Command1_Click()
   If FRunning Then Exit Sub 'Check whether a previous call is running
   FRunning = True
   'Call some DynaPDF functions here...
   DoEvents 'Process messages
   FRunning = False
End Sub
```

The code above simply checks whether a previous call of the function is running before the function can be executed again.

Customized Exception handling

By default, only fatal errors will stop processing. Warnings, syntax errors and so on are all ignored. You can customize the exception handling to your own requirements with the property Get/SetOnErrorMode(). With the following constants you can determine what kind of error should be treated as fatal error:

```
typedef SI32 TErrMode;
#define emIgnoreAll     0x00000000 // default
#define emSyntaxError   0x00000001
#define emValueError    0x00000002
#define emWarning       0x00000004
#define emFileError     0x00000008
#define emFontError     0x00000010
#define emAllErrors     0x0000FFFF
#define emNoFuncNames   0x10000000 // Do not output function names
```

If a specific flag is set, DynaPDF treats this error type as fatal error; the internal resources will be freed and all changes made before are lost.

DynaPDF never produces a damaged PDF file if a warning or error message was ignored, but certain functions may be not executed. For example, if SetFont() cannot find the selected font, it returns with a warning and no font is set, the active font (if any) is left unchanged.

If no other font was set before, it is not possible to output text. All text functions also return then with a warning because there is no active font, but nothing more happens than a warning is issued.

When a fatal error occurred, all functions are leaved immediately. No further warnings or error messages are displayed. It is not possible to execute a function (except global properties which do not change PDF objects directly) after a fatal error occurred.

There is no need to check the return value of each function, and there is no need to leave a function block after a fatal error occurred. The internal error flag is cleared when CreateNewPDF() is called the next time.

The special flag *emNoFuncNames* names can be used to avoid the output of the function name in error messages. Error messages start normally always with the function name in which the error occurred. While this information is useful during development, it is often not useful in an end user application.

Remarks:

The constants are defined as enum in Visual Basic, VB .Net, and C#.

Custom Library Changes

This section is only of interest if you have a copy of the source codes. If you use a version without source codes you can skip this chapter.

Compiler Switches

DynaPDF supports several compiler switches to disable unnecessary features. The following macros disable or enable one of the image libraries used by DynaPDF as well as other features. The macros are defined in the header file /main/drv_type.h.

```
#define DRV_SUPPORT_AES  1 // AES encryption and decryption
#define DRV_SUPPORT_BCDE 1 // about 400 KB -> Barcode engine
#define DRV_SUPPORT_CJK  1 // about 150 KB (CJK to Unicode conversion)
#define DRV_SUPPORT_EMF  1 // EMF Converter
#define DRV_SUPPORT_GIF  1 // about 1 KB
#define DRV_SUPPORT_IMP  1 // about 50 KB (PDF import)
#define DRV_SUPPORT_JP2K 1 // about 180 KB
#define DRV_SUPPORT_JPEG 1 // about 90 KB (no effect if TIFF is enabled)
#define DRV_SUPPORT_PGM  1 // PBM, PGM, PNM, PPM Image formats, ~1 KB
#define DRV_SUPPORT_PNG  1 // about 100 KB
#define DRV_SUPPORT_PSD  1 // about 1 KB
#define DRV_SUPPORT_RAS  1 // about 700 KB -> Rendering engine
#define DRV_SUPPORT_RC4  1 // RC4 encryption and decryption
#define DRV_SUPPORT_SIGN 1 // Self sign signatures -> AiCrypto Library
#define DRV_SUPPORT_TIFF 1 // about 310 KB
```

To disable a specific feature set the constant to zero or comment it out. Note that the TIFF library uses also the JPEG library. Because of this, disabling the JPEG library only does not reduce the library size.

The following constants are used by WriteFText() (defined in dynapdf.h).

```
#define PDF_MAX_LIST_COUNT 6    // Maximum count of nested list levels
#define PDF_LIST_SEP_WIDTH 10.0 // Default list separator with
#define DEFAULT_LIST_CHAR  159  // Default list character
#define PDF_LIST_FONT "Wingdings-Regular" // Default list font
```

The list font must be the PostScript name of the font (see SetFont() for further information). When changing the list font you may also change the default list character. When using DynaPDF under Linux or UNIX you may define a font that is available in one of your font search directories.

Note that the list font can be overridden at runtime with the function SetListFont(). So, it is usually better to load the font at runtime with SetListFont() since you can properly handle cases in which the font cannot be found.

Main object types

In PDF, two basic object types can be created, *resource objects* such as fonts, images and so on which are used by content streams and *global objects* such as annotations, bookmarks, form fields and so on. Global objects can use resource objects but not vice versa.

In most cases both object types are defined as normal classes, which contain their own constructors and destructors to initialize and destroy allocated memory.

Global objects can be deleted or marked as deleted at runtime. Resource objects must never be deleted if the object was already used.

General design requirements

We describe here only the general rules which must be taken into account when extending DynaPDF with certain features. We do not explain how the entire library works; this would fill an entire book. To understand how an object must be written to the file we recommended that you debug especially the function CloseFile(). All objects must be prepared for writing in a two stage phase to reserve object numbers. The first stage assignes object numbers to all objects and the second run writes all objects to file. Objects must be written in the exact order in which they were previously prepared for writing.

A PDF file is described in memory as large set of classes which can be referenced or used multiple times by other classes. Due to the references which are stored in certain classes we must define a strict set of rules so that no exception occurs if an object must be deleted:

1. The owner of all resources and global objects is CPDF. No other class is permitted to destroy an object class or change its values.

2. All object classes must be derived from CBaseObject. This class holds the object number as well as several flags to determine whether the object was used, created, or already written, and whether it is part of the first page. This class contains the function CreateObject() which must be called in CPDF::PreparePageObjects() if the object is part of a page. If the object is not included in a page object then CreateObject() must be called in CPDF::PrepareObjects().

3. All classes which hold pointers to other object classes must check the "Used" flag before writing the object data to the file (GetUsed() is a member of CBaseObject and returns true if the used flag was set).

4. All classes must be well initialized so that the class can be deleted at any time without causing memory leaks or other unwanted side effects.

5. No object class is permitted to unset the "Used" flag of other object classes.

6. Page resources such as fonts, images, templates and so on must **NEVER** be deleted at runtime and their "Used" flag must **NEVER** be unset.

7. All resource classes must be derived from CBaseResource.

8. Object classes must set the "Used" flag of the resource object, if the class is used by this object.

9. The "Used" flag of non-page resources can be unset at runtime with SetUnUsed(). This will mark the class as an unused or deleted object. When writing the file, such an object must be ignored.

10. Never delete an object if you don't know exactly what you are doing. If a destroyed object is referenced in any other object, the library will crash.

11. If a **used** page resource is deleted or if the "Used" flag is unset at runtime when it was already set, the resulting PDF file will be damaged.

12. Make sure that the function FreePDF() can be called at any time without causing memory leaks or other unwanted side effects.

Requirements to add your own code to DynaPDF

If you want to make your own features permanent, you are welcome to send us your source codes including a description what kind of feature it enables.

However, we cannot accept any old code; your code must be properly written in C++ and tested with certain operating systems and compilers. It must not produce warnings of any kind and it must not use external libraries, except those are already used by DynaPDF (see drv_type.h for a full list).

Basic C functions such as strcpy, strcat, memcpy and so on, as well as templates from the STL are **NOT** allowed to use. Take a look into drv_base_func.h, drv_templ.h, or pdf_utils.h before using an external function or template. The implementations used by DynaPDF are faster and work on all operating systems in the same way.

Windows GDI functions are normally not permitted too. However, under certain circumstances GDI functions are permitted. DynaPDF uses already a few GDI functions to convert WMF files to EMF or to raster EMF files. Windows specific code must be encapsulated into a #ifdef _WINDOWS section.

If your code handles strings, it must **NOT** use an external string class which is available in any standard C++ library. Use the DynaPDF class CString or CPDFString() instead. These classes support many PDF specific functions.

Your code should be tested with Microsoft Visual C++ 6.0 **AND** Visual Studio 2005 or higher. If possible, you should test your code also with GCC 4.0 or higher under Linux or Mac OS X.

The following important defines are available depending on the operating system and characteristics of the target CPU:

Constant	Test code	Comment
_WINDOWS	`#ifdef _WINDOWS`	Set on Windows.
MAC_OS_X	`#ifdef MAC_OS_X`	Set on Mac OS X.
DRV_BIG_ENDIAN	`#if (DRV_BIG_ENDIAN == 1)`	Endian configuration.
VS_2005_OR_HIGHER	`#if (VS_2005_OR_HIGHER == 1)`	Functions which are considered as unsafe in Visual Studio 2005 or higher must be replaced with their safe versions. Use this macro to test the compiler version.

To get the full list of available defines take a look into the header file /main/drv_type.h. On non Windows operating systems the configure script creates also the header file /main/drv_conf.h which contains many OS specific defines. Note that this header file can only be included on non Windows operating systems.

Note also that the class structure in DynaPDF is not fixed. Changes can be made without further notice at any time. So, you cannot assume that a class or member of a class that exists today exists in a newer version too. Although most classes and templates do seldom change, changes can always occur!

To avoid dependencies to a specific source code version it is usually best to ask us for the best strategy if you want to add certain features to the library.

DPI Aware .Net applications

The difference of dpi aware applications in comparison to regular applications is that the latter one do not properly scale on high resolution displays. Blurry text is an indication that an application is not dpi aware.

C# PDF viewer example with disabled and enabled high dpi support:
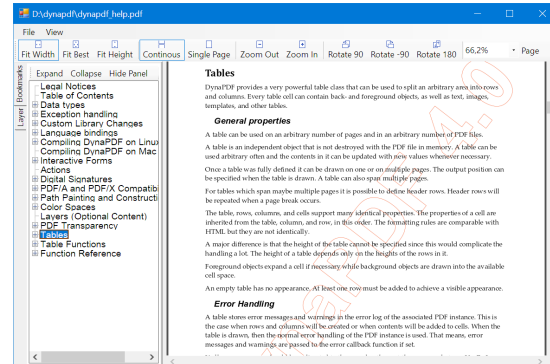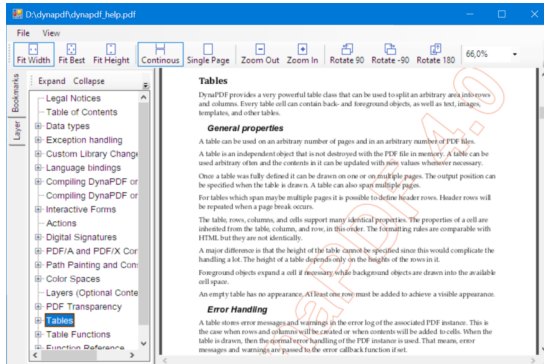
As you can see above the treeview control does not properly scale. The same control looks much better when rendered in a lower resolution. Such issues require some fine tuning for optimal results.

A user can of course right click on the executable and change the high dpi settings in the compatibility section of the application properties. This works pretty well with Windows 10 Version 2004 but every major version of Windows 10 behaves differently and we have currently 11 major versions of Windows 10!

A bit annoying is also the fact that these settings do generally not work if the application was copied to a RAM disk on Windows 10 Version 2004. This was no issue in earlier versions.

So, it would be much better to enable high dpi support without additional user interaction. In earlier days high dpi support needed to be enabled with a dpi aware manifest. This worked for a while until the way was changed how high dpi support must be enabled.

The new way of enabling high dpi support is to add a section to the app.config file.

Enabling High DPI support in Windows Forms Apps

To get high dpi support enabled in Windows Forms Apps you must first change the version of the used .Net framework to 4.7 or higher.

If you do this in the settings of the PDF viewer example, for example, then you'll notice that the required app.config fille will be added automatically to the solution. Double click on it and copy the code marked in yellow into the configuration tag:

```xml
<?xml version="1.0" encoding="utf-8"?><configuration>
<startup><supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2"/></startup>
<System.Windows.Forms.ApplicationConfigurationSection>
    <add key="DpiAwareness" value="PerMonitorV2" />
</System.Windows.Forms.ApplicationConfigurationSection>
</configuration>
```

The above settings are for Windows Forms Apps. Note that this is how it works today, tomorrow you must maybe change other settings.

WPF and UWP apps come out of the box with high dpi support but using native code for UI elements in WPF or UWP is difficult and not recommended. WPF and UWP apps function more or less like a web site. When speed is no concern it is better to use pure .Net code in these environments.

Scrolling issues on high dpi devices

Scrolling issues are one of the major issues on high dpi devices. Almost every application scrolls differently on Windows 10. Some scroll way to fast, others scroll too slow, and again others are stuttering or have weird scroll behavior.

To avoid scrolling issues is probably best to get users a chance to adjust the scrolling speed in your application since the settings of the system are often not helpful.

If you use the rendering control of DynaPDF or the page cache then you can adjust the scroll line delta with SetScrollLineDelta(). Smooth scrolling is not yet supported.

Static contents in dpi aware applications

Something that must be considered when turning on high dpi support is that static contents like the application icon or cursors must be delivered in different sizes since no scaling occurs.

Especially cursors are critical because an unscaled cursor made for Windows 7, for example, becomes very small. Microsoft recommends to add at least icons and cursors for the resolutions 96, 120, 144, and 196 DPI for desktop apps. You'll find many tools in the internet which help to create icons and curors in different sizes.

Language bindings

Differences between DynaPDF interfaces

DynaPDF can be used with most programming languages which support standard DLLs. The usage of DynaPDF is nearly identical in all programming languages. However, this help file describes all DynaPDF functions in C syntax.

All DynaPDF functions contain an instance pointer of the active PDF instance (`const PPDF* IPDF`) as first parameter. This pointer is hidden for the user in the programming languages Visual Basic, Visual Basic .Net, Visual C#, and Delphi. We deliver native wrapper classes for these programming languages which handle the PDF instances automatically.

However, this help file describes the raw API of DynaPDF that is used by the programming languages C and C++. The C/C++ interface is a direct interface that does not encapsulate the DynaPDF API into a class or other structures to improve processing speed.

The usage of DynaPDF is almost identical in all programming languages; you must only consider that the instance pointer IPDF is contained in the C/C++ interface only.

Visual Basic, Visual C#, or Delphi users must create an instance of the wrapper class CPDF or TPDF in Delphi before a DynaPDF function can be executed. The instance of the wrapper class must also be deleted by calling the destructor of the class.

C or C++ programmers must create a PDF instance with the function pdfNewPDF() before a DynaPDF function can be executed. This instance must be deleted with the function pdfDeletePDF() when it is no longer needed.

We tried also to consider the specific requirements for each programming language so that DynaPDF can always be used without limitations. This causes slightly differences because of the differences between programming languages. For instance, the Visual Basic interfaces uses events instead of callback functions because the usage of callback functions is more complicated in VB as in other programming languages.

Embarcadero C++ Builder

To use DynaPDF with Embarcadero's C++ Builder, proceed as follows:

1. Open a new project or your favourite project in C++ Builder.

2. Include the header file */include/C_CPP/dynapdf.h* into the units which will use DynaPDF functions.

3. Add the import library */borland_lib/dynapdf.lib* to your project.

4. Copy the *dynapdf.dll* into a Windows search path (e.g. Windows/System32) or into the output directory.

5. Finished!

If you want to use DynaPDF with a C++ project, add the line "using namespace DynaPDF;" after including the header file dynapdf.h.

The usage in C is essentially the same as in C++, with the exception that the namespace DynaPDF must not be declared.

A PDF instance can be used to create an arbitrary count of PDF files. All used resources are automatically freed when the PDF file is closed (except when the PDF file was created in memory). To improve processing speed, use one instance as long as possible.

If you want to use DynaPDF with an older version of C++ Builder, you may rebuild the lib file by using implib.exe. Implib is delivered with C++ Builder; you find it in the *Bin* directory of your C++ Builder installation directory.

To create a new import library run implib from the command line as follows:

```
implib dynapdf.lib dynapdf.dll
```

DynaPDF uses standard call as calling convention. C++ Builder requires for that calling convention no underscores before function names. Because of this the option -a must not be used to build the import library (see your C++ Builder help for further information).

Copy *implib* and the dynapdf.dll into the same directory and execute the command above.

Example (C++ Builder):

```cpp
// Standard includes by all C++ Builder projects
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
```

```cpp
// Include the DynaPDF header file (change the path if necessary)
#include "dynapdf.h"
// All data types and functions are declared in the namespace DynaPDF.
using namespace DynaPDF;
// First, we declare an error callback function that is called by
// DynaPDF if an error occurred.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   char errMsg[PDF_MAX_ERR_LEN + 30];
   // An error message returned by DynaPDF is a pointer to a null-
   // terminated static string.
   sprintf(errMsg, "%s\n\nAbort processing?\n", ErrMessage);
   if (MessageDlg(errMsg, mtError,
   TMsgDlgButtons() << mbYes << mbNo, 0) == mrYes)
      return -1; // break processing
   else
      return 0;  // ignore the error
}

// Place a button on the form and double click on it. Add the
// following code to the function.
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   void* pdf = pdfNewPDF(); // Create a new PDF instance
   if (!pdf) return; // Out of memory?
   // Set the error callback first
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfCreateNewPDF(pdf, "c:/myfirst.pdf");
   pdfSetDocInfo(pdf, diSubject, "My first C++ output");
   pdfSetDocInfo(pdf, diProducer, "C++ Builder test application");
   pdfSetDocInfo(pdf, diTitle, "My first C++ output");
   // We want to use top-down coordinates
   pdfSetPageCoords(pdf, pcTopDown);

   pdfAppend(pdf);
   pdfSetFont(pdf, "Arial", DynaPDF::fsItalic, 40.0, true, cp1252);
   pdfWriteFText(pdf, DynaPDF::taCenter, "My first C++ output!");
   pdfEndPage(pdf);

   pdfCloseFile(pdf); // Close the file and free all used resources
   pdfDeletePDF(pdf); // Do not forget to delete the PDF instance
}
```

Microsoft Visual C++

To use DynaPDF with Microsoft's Visual C++, proceed as follows:

1. Open a new project, or your favourite project, in Visual Studio.

2. Include the header file */include/C_CPP/dynapdf.h* into the units which will use DynaPDF functions.

3. VC++ 6.0 only: Add the file */include/C_CPP/dynapdf.cpp* to your project (see comment below).

4. Add the import library */mvs_lib/dynapdf.lib* or */mvs_lib/dynapdfm.lib* to your project.

5. Copy the *dynapdf.dll* or *dynapdfm.dll* into a Windows search path (e.g. Windows/System32).

6. Finished!

The file *dynapdf.cpp* contains dummy declarations of all exported DynaPDF functions; because the functions are not exported, they do not conflict with the original declarations. These declarations enable the usage of Microsoft's Intellisense (code completion) in Visual C++ 6.0. Due to implementation limits of Microsoft's Intellisense, code completion does maybe not work without this file. However, it seems that this issue was partially solved with Update Pack 6 of Microsoft's Visual Studio. If code completion does not work in your environment then include this file.

See also bug report „190974 PRB: Function Prototypes Do Not Generate Parameter Info", (www.microsoft.com).

DynaPDF was compiled and developed with Microsoft Visual Studio 6.0 SP 6 and Visual Studio 2005. Two versions of the library are delivered:

- `dynapdf.dll  // Compiled with Multithreaded`
- `dynapdfm.dll // Compiled with Multithreaded DLL`

Both versions are fully compatible to VC++ 7.0 or higher (Visual Studio .Net). To avoid conflicts with different standard library versions choose the right DynaPDF DLL and import library depending on your project settings. A pre-compiled single threaded version of DynaPDF is not available.

Please note that the dynapdfm.dll does not support PDF files larger than 2 GB. When the library is compiled with Visual Studio 2005 or higher, then this limitation does no longer exist. However, in this case, you must deliver the Visual Studio Runtime library MSVCR80.DLL or higher with your application. This library causes often problems since it must be installed on the system and cannot be placed in the application folder.

If possible, then use the dynapdf.dll instead. This library requires no additional dependencies to enable 64 bit file support.

If you want to use DynaPDF with a C++ project, add the line "using namespace DynaPDF;" after including the header file dynapdf.h.

The usage in C is essentially the same as in C++, with the exception that the namespace DynaPDF must not be declared.

A PDF instance can be used to create an arbitrary count of PDF files. All used resources are automatically freed when the PDF file is closed (except when the PDF file was created in memory). To improve processing speed, use one instance as long as possible.

Most examples are written in C or C++ which can directly be used with a Embarcadero or Microsoft Compiler.

Example (C++):

```cpp
// Include the DynaPDF header file (change the path if necessary)
#include "dynapdf.h"
// All data types and functions are declared in the namespace DynaPDF.
using namespace DynaPDF;
// First, we declare an error callback function that is called by
// DynaPDF if an error occurred.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0; // Any other return value break processing
}
// Place a button on the form and double click on it. Add the
following
// code to the function.
int main(int argc, char* argv[])
{
   void* pdf = pdfNewPDF(); // Create a new PDF instance
   if (!pdf) return 2; // Out of memory?
   // Set the error callback first
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfCreateNewPDF(pdf, "c:/myfirst.pdf");
   pdfSetPageCoords(pdf, pcTopDown); // We use top-down coordinates
   pdfAppend(pdf);
   pdfSetFont(pdf, "Arial", fsItalic, 40.0, true, cp1252);
   pdfWriteFText(pdf, taCenter, "My first C++ output!");
   pdfEndPage(pdf);
   pdfCloseFile(pdf); // Close the file and free all used resources
   pdfDeletePDF(pdf); // Do not forget to delete the PDF instance
}
```

Microsoft Visual Basic 6.0

The usage of DynaPDF with Visual Basic is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class CPDF to make the usage easier. The instance pointer IPDF which is used by every DynaPDF function is hidden for the user in Visual Basic. The instance pointer is controlled by the wrapper class so that you don't need to create PDF instances manually.

To use DynaPDF with Visual Basic proceed as follows:

- Add the file */include/Visual_Basic/DynapPDFInt.bas* to your project (menu Project/Add Module/Existing…).

- Add the file */include/Visual_Basic/CPDF.cls* to your project (menu Project/Add Class Module/Existing…).

- Add the file */include/Visual_Basic/IPDFCallback.cls* to your project (menu Project/Add Class Module/Existing…).

- If you want to use the table class then add also the file */include/Visual_Basic/CPDFTable.cls* to your project (menu Project/Add Class Module/Existing…)*.*

- Finally, make sure that the *dynapdf.dll* can be found by Visual Basic in debug mode; just copy the DLL into Windows/System32 or into Windows/SysWow64 on a 64 bit system, finished!

Note that Visual Basic supports the 32 bit dynapdf.dll only. If you work on a 64 bit OS then copy the library into Windows/SysWow64. Yes, this is the right folder for 32 bit DLLs!

All DynaPDF functions are encapsulated in the wrapper class CPDF. This class makes sure that the library can be used without limitations and programming is more comfortable since you can work with a native VB class. You don't need to consider specific return values of the DLL, the class converts special data types automatically to VB data types.

Exception handling in Visual Basic

The standard exception handling of DynaPDF uses a callback function to pass error messages and warnings to the client application. However, in Visual Basic we use events instead. The usage of events is quite easy and frees you from dealing with DynaPDF API function calls.

To enable the event support of the wrapper class CPDF declare a local instance variable as follows:

```vb
Option Explicit
Private WithEvents FPDF As CPDF 'Enable event support
```

The instance variable FPDF is now listed in the left combo box of the VB code editor.
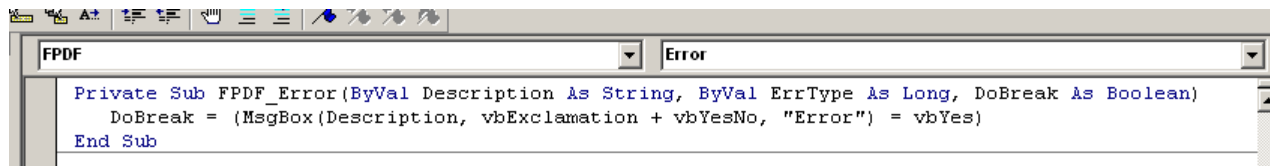
```
Private Sub FPDF_Error(ByVal Description As String, ByVal ErrType As Long, DoBreak As Boolean)
    DoBreak = (MsgBox(Description, vbExclamation + vbYesNo, "Error") = vbYes)
End Sub
```

The right combo box contains the available events, when selecting the event "Error" VB adds automatically an empty event procedure to your source code:

```
Private Sub FPDF_Error(ByVal Description As String, ByVal ErrType As Long, DoBreak As Boolean)
    ' Add your code here
    ' To break processing after an error occurred set the variable
    ' DoBreak to True
End Sub
```

Now you can enter some code that should be executed when the event is fired. Note that you must still create an instance of the class CPDF before a DynaPDF function can be executed (see the example below).

Note also that VB exceptions are <u>not</u> used so that an error block will normally never be executed. The following declaration has effect in the means of the DynaPDF exception handling:

```
On Error GoTo errPDF
...
' Call some DynaPDF functions...
: errPDF
If Err.Number <> 0 Then
    MsgBox Err.Description
    Exit Sub
End If
```

The error block cannot be executed because DynaPDF does never raise an exception (except the class constructor). If an error occurred, the event "Error" is raised instead. However, there are still cases in which a VB exception can occur, e.g. when passing an empty array to a function that requires some values in it. So, the code should still be encapsulated in an error block. All you need to know that this error block does not affect the normal exception handling of DynaPDF.

The DoEvents problem

The usage of events in Visual Basic is quite easy but there is a special behaviour that must be taken into account when developing VB applications. When using the DoEvents procedure in a VB function you must make sure that the function cannot be executed again while a previous call of the function is still running.

DoEvents enables the asynchronous processing of the message loop so that the user interface can be updated and the user can execute something while a function is still running (e.g. press a

break button). DoEvents is often used because it is an easy way to avoid blocking of an application without using of threads.

However, when using DoEvents it is possible that a user clicks on the button again that executes DynaPDF functions while a previous call is still running. This is normally no problem but it is impossible to execute an event functions inside of a cloned function. When DynaPDF tries to raise an event inside the cloned function an access violation occurs and VB crashes.

To avoid such problems check whether the function is still running:

```
Option Explicit
Private WithEvents FPDF As CPDF 'Enable event support
Private FRunning As Boolean

Private Sub Command1_Click()
   If FRunning Then Exit Sub 'Check whether a previous call is running
   FRunning = True
   'Call some DynaPDF functions here...
   DoEvents 'Process messages
   FRunning = False
End Sub
```

The code above checks whether a previous call of the function is running before the function can be executed again; a quite simple but effective solution that makes your application stabile.

Example:

```
Option Explicit
Private WithEvents FPDF As CPDF ' Enable event support
Private Sub Form_Load()
   ' We hold one instance of the class CPDF in memory
   Set FPDF = new CPDF
   If FPDF Is Nothing Then
      MsgBox "Out of memory!", vbCritical, "Fatal error"
   End If
End Sub
Private Sub Form_Terminate()
   ' Delete the class instance
   Set FPDF = Nothing
End Sub
Private Sub Command1_Click()
   FPDF.CreateNewPDFA "c:/vbout.pdf"
   FPDF.SetDocInfoA diAuthor, "Jens Boschulte"
   FPDF.SetDocInfoA diSubject, "My first VB output"
   FPDF.SetDocInfoA diTitle, "My first VB output"
   FPDF.Append
   FPDF.SetFont "Arial", fsItalic, 30#, True, cp1252
   FPDF.WriteFTextA taCenter, "My first VB output"
   FPDF.EndPage
```

```vb
    FPDF.CloseFile
End Sub
' Error event procedure
Private Sub FPDF_Error(ByVal Description As String, ByVal ErrType As
Long, DoBreak As Boolean)
    DoBreak = (MsgBox(Description, vbExclamation Or vbYesNo,
            "Error") = vbYes)
End Sub
```

Visual Basic .Net

The usage of DynaPDF with Visual Basic .Net is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class CPDF to make the usage easier. The instance pointer IPDF that is used by every DynaPDF function is hidden for the user in VB .Net. The instance pointer is controlled by the wrapper class so that you don't need to create PDF instances manually.

To use DynaPDF with VB .Net proceed as follows:

- Add the file */include/Visual_Basic_Net/CPDF.vb* to your project (menu Project/Add Existing Element…).

- Finally, make sure that the *dynapdf.dll* can be found; just copy the DLL into Windows/System32, finished!

64 Bit Applications

With VB .Net you can develop 32 bit and 64 bit applications. One thing that must be considered is that the target CPU type in Visual Studio must **not** be set to *UseAny*. This is impossible since you can either link the 32 bit dynapdf.dll or the 64 bit version but not both.

So, a 32 bit and 64 bit version must be compiled separately. Another thing that is often misunderstood is the right system directory for the dynapdf.dll. If you develop a 32 bit application on a 64 bit Windows version then copy the 32 bit version of the dynapdf.dll into Windows/SysWow64 and the 64 bit version into Windows/System32. Yes, this is correct!

Both versions can be used simultaneously. Windows loads automatically the right version if you have copied the DLLs into the right directories.

Note that the DLL should be copied into the system folder on your development machine only so that Visual Studio is able to load it. The installer of your application should copy the DLL into the application directory instead.

General Note:

Visual Studio .Net copies the interface files into your project directory if the option "Link file" is not selected when adding the files to your project. Make sure that you always link the files to your project. Otherwise you must update the interface files manually whenever you install a newer version of DynaPDF.

All DynaPDF functions are encapsulated in the wrapper class CPDF. This class makes sure that the DynaPDF functions can be used without limitations and programming with DynaPDF becomes more comfortable. You don't need to consider specific return values of the DLL; the class converts API data types automatically to VB .Net data types.

VB .Net supports more data types than VB 6.0 but the usage of the new data types is complicated. For instance, a signed integer cannot be passed to an unsigned integer variable without explicit conversion. The same behaviour is required for nearly all other new data types. This makes the usage of the new types practically impossible.

However, to make the usage of DynaPDF less complicated the VB .Net interface uses only base data types which are already available since VB 6.0. An exception is the definition of pointers. VB .Net supports the new Data IntPtr that is a variable pointer type with a length of 32 bit on a 32 bit Windows machine and 64 bit on a 64 bit Windows machine. This data type is used for all pointers so that the .Net interface can also be used on a 64 bit Windows machine. Note that the 64 bit version of DynaPDF must be used in this case.

Data types used by DynaPDF

DynaPDF uses a large set of enums and other data types which are mostly declared within the class CPDF. However, a few data types are declared in the file DynaPDFInt.vb, this must be taken into account when developing VB .Net applications. If you cannot find a data type in the class CPDF then take a look into the file DynaPDFInt.vb.

Exception handling in VB .Net

The exception handling in Visual Basic .Net is the same as in Visual Basic 6.0. The class CPDF uses per default events instead of callback functions. This makes the usage easier and frees you from handling with unmanaged data types. Please take a look into Visual Basic Exception handling for a detailed description.

It is also possible to declare callback functions instead of events but the use of callback functions is rather complicated in VB .Net. While C# handles callback functions correctly without further considerations, VB .Net users must test their callback function properly. It seems that VB .Net invalidates sometimes the pointers of callback functions if memory must be reallocated. This error occurs mostly if DynaPDF functions are executed in different classes. If you get a NullReference exception check whether the same error occurs if no error callback function is set. If the exception does no longer occur use the error event handling of DynaPDF instead.

The DoEvents problem

The usage of events in VB .Net is quite easy; however, there is a special behaviour that must be taken into account when developing .Net applications. When using the DoEvents procedure in a function you must make sure that the function cannot be executed again while a previous call of the function is still running.

DoEvents enables the asynchronous processing of the message loop so that the user interface can be updated and the user can execute something while a function is still running (e.g. press a break button). DoEvents is often used because it is an easy way to avoid blocking of an application without using of threads.

However, when using DoEvents it is possible that a user clicks on the button again that executes DynaPDF functions while a previous call is still running. This is normally no problem but when using events the event functions become invalid. This is the same behaviour as in Visual Basic 6.0 with the exception that .Net does not crash, a System.NullReferenceException is raised instead.

It is not clear why this exception occurs, it seems that this is a general bug in the event handling of Visual Basic 6.0 and VB .Net. A native programming language like C/C++ or Delphi would never cause an access violation or exception here.

However, to avoid such problems check whether the function is still running:

```vb
Private WithEvents FPDF As CPDF 'Enable event support
Private FRunning As Boolean

Private Sub Command1_Click(ByVal eventSender As System.Object, ByVal
eventArgs As System.EventArgs) Handles Command1.Click
   If FRunning Then Exit Sub 'Check whether a previous call is running
   FRunning = True
   'Call some DynaPDF functions here...
   DoEvents 'Process messages
   FRunning = False
End Sub
```

The code above checks whether a previous call of the function is running before the function can be executed again. A quite simple but effective solution that makes your application stabile.

Example (Visual Basic .Net):

```vb
Private WithEvents FPDF As CPDF 'Enable event support

Private Sub Form1_Load(ByVal eventSender As System.Object, ByVal
eventArgs As System.EventArgs) Handles MyBase.Load
   ' We hold one instance of the class CPDF in memory
   FPDF = New CPDF()
End Sub

Private Sub Command1_Click(ByVal eventSender As System.Object, ByVal
eventArgs As System.EventArgs) Handles Command1.Click
   FPDF.CreateNewPDFA("c:/vbout.pdf")
   FPDF.SetDocInfoA(CPDF.TDocumentInfo.diAuthor, "Jens Boschulte")
   FPDF.SetDocInfoA(CPDF.TDocumentInfo.diSubject, "My first VB output")
   FPDF.SetDocInfoA(CPDF.TDocumentInfo.diTitle, "My first VB output")

   FPDF.Append()
   ' The data type TFStyle is defined in DynaPDFInt.vb
   FPDF.SetFontA("Arial", TFStyle.fsItalic, 30.0, True, CPDF.TCodepage.cp1252)
   FPDF.WriteFTextA(CPDF.TTextAlign.taCenter, "My first VB .Net output")
   FPDF.EndPage()
   FPDF.CloseFile()
```

```vbnet
End Sub

' Error event procedure
Private Sub FPDF_PDFError(ByVal Description As String, ByVal ErrType
As Integer, ByRef DoBreak As Boolean) Handles FPDF.PDFError
    DoBreak = (MsgBox(Description, _
               MsgBoxStyle.Exclamation Or _
               MsgBoxStyle.YesNo, _
              "Error") = MsgBoxResult.Yes)
End Sub
```

Visual C#

The usage of DynaPDF with Visual C# is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class CPDF to make the usage easier. The instance pointer IPDF that is used by every DynaPDF function is hidden for the user in C#. The instance pointer is controlled by the wrapper class so that you must not create PDF instances manually.

.Net Core compatibility

.Net Core applications can be compiled with Visual Studio for Windows, Linux, and macOS. In order to use DynaPDF on these operating systems, copy the DynaPDF DLL or shared library into the output folder. The library can be found in the dynapdf directory on Windows or download the corresponding Linux or macOS archive respectively:

- `Windows 32 bit: /dynapdf/win32/dynapdf.dll`
- `Windows 64 bit: /dynapdf/win64/dynapdf.dll`
- `Linux 64 bit:  /dynapdf/dynapdf/libdynapdf.so`
- `macOS 64 bit:  /dynapdf/dynapdf/libdynapdf.dylib`

Using DynaPDF with Visual Studio

To use DynaPDF with Visual C# proceed as follows:

- Add the file */include/Visual_C#/CPDF.cs* to your project (menu Project/Add Existing Element…).

- Make sure that the *dynapdf.dll* can be found. On the development machine execute the msi installer (dynapdf.msi). The installer copies the the 32 and 64 bit library into Windows/System32 and Windows/SysWow64. On Windows 7 or earlier it is also possible to copy the library manually into the system folders but this is not possible on Windows 10.

- Note that an installation into the system folders is recommended on a development machine only. On the target system the DLL should be copied into the application folder to avoid issues with different DynaPDF versions which might be installed on the system.

64 Bit Applications

With C# you can develop 32 bit and 64 bit applications. One thing that must be considered is that the target CPU type in Visual Studio must **not** be set to *UseAny*. This is impossible since you can either link the 32 bit dynapdf.dll or the 64 bit version but not both.

So, a 32 bit and 64 bit version must be compiled separately. Another thing that is often misunderstood is the right system directory for the dynapdf.dll. If you develop a 32 bit

application on a 64 bit Windows version then copy the 32 bit version of the dynapdf.dll into Windows/SysWow64 and the 64 bit version into Windows/System32. Yes, this is correct!

Both versions can be used simultaneously. Windows loads automatically the right version if you have copied the DLLs into the right directories.

Note that the DLL should be copied into the system folder on your development machine only so that Visual Studio is able to load it. The installer of your application should copy the DLL into the application directory instead.

General Note:

Visual Studio .Net copies the interface file CPDF.cs into your project directory if the option "Link file" is not selected when adding the file to your project. Make sure that you always link the files to your project. Otherwise you must update the interface manually whenever you install a newer version of DynaPDF.

All DynaPDF functions are encapsulated in the wrapper class CPDF. This class makes sure that the DynaPDF functions can be used without limitations and programming with DynaPDF becomes more comfortable. You don't need to consider specific return values of the DLL; the class converts API data types automatically to C# data types.

However, C# uses a very restrictive data type handling that causes that already signed integers cannot be passed to unsigned integer variables of the same type without explicit conversion.

To make the usage of DynaPDF less complicated most function parameters which would normally declared as unsigned integer, e.g. PDF object handles, are declared as signed integer to get rid of permanent explicit data type conversions.

The usage of DynaPDF with C# is nearly identical in comparison to C++. The interface does generally not use events like in VB .Net because callback functions work very well in C#.

Data types in C#

All structures and enums used by DynaPDF are declared in the namespace DynaPDF. Because it is not possible to declare constants in a namespace, such constants are declared in the class CPDF. All data types, structures, and constants are defined in the file CPDF.cs. No further files are required to use DynaPDF.

Example (Visual C#):

```csharp
using System;
using DynaPDF;
using System.Runtime.InteropServices.Marshal;

namespace hello_world
{
    class Hello_World
    {
        // Error callback function.
        static int PDFError(IntPtr Data, int ErrCode, IntPtr ErrMessage,
```

```csharp
        int ErrType)
      {
         // The error type is a bitmask.
         Console.Write("{0}\n", PtrToStringAnsi(ErrMessage));
         Console.Write("\n");
         return 0; // We try to continue if an error occurs.
      }

      [STAThread]
      static void Main(string[] args)
      {
         try
         {
            String outFile = "c:/c#out.pdf";
            CPDF pdf = new CPDF();
            // Error messages are passed to the callback function.
            pdf.SetOnErrorProc(IntPtr.Zero, new DynaPDF.TErrorProc(PDFError));
            // We open the output file later if no error occurs.
            pdf.CreateNewPDF(null);
            // We use top down coordinates in this example
            pdf.SetPageCoords(DynaPDF.TPageCoord.pcTopDown);
            // Add an empty page to the file
            pdf.Append();
            // Before printing text you must set a font
            pdf.SetFont("Arial", DynaPDF.TFStyle.fsItalic, 20, true, DynaPDF.TCodepage.cp1252);
            pdf.WriteText(50.0, 50.0, "My first PDF output...");
            pdf.WriteText(50.0, 80.0, "File created: " + DateTime.Now.ToString());
            pdf.EndPage(); // Close the open page

            // No fatal error occurred?
            if (pdf.HaveOpenDoc())
            {
               // OK, now we can open the output file.
               if (!pdf.OpenOutputFile(outFile))
               {
                  // An error message was already passed to the error callback function
                  Console.Read();
                  return;
               }
               if (pdf.CloseFile())
               {
                  Console.Write("PDF file \"{0}\" successfully created!\n", outFile);
               }
            }
            pdf = null;
         }catch(Exception e)
         {
            Console.Write(e.Message);
         }
         Console.Read();
      }
   }
}
```

Embarcadero Delphi

The usage of DynaPDF with Embarcadero's Delphi is essentially the same as with C or C++ except that the exported DLL functions are encapsulated in the wrapper class TPDF to make the usage easier. The instance pointer IPDF that is used by every DynaPDF function is hidden for the user in Delphi. The instance pointer is controlled by the wrapper class so that you don't need to create PDF instances manually.

To use DynaPDF with Delphi, proceed as follows:

- Add the interface file */include/Delphi/dynapdf.pas* to your project.

- Add the unit dynapdf in the uses section in every source file where you want to use DynaPDF.

- Copy the *dynapdf.dll* into a Windows search path (e.g. Windows/System32) or into your application directory, finished!

64 Bit Applications

Since Rad Studio XE2 you can develop 32 bit and 64 bit applications with Delphi. One thing that is often misunderstood is where the 32 bit and 64 bit versions of dynapdf.dll must be stored. If you develop a 32 bit application on a 64 bit Windows version then copy the 32 bit version of the dynapdf.dll into Windows/SysWow64 and the 64 bit version into Windows/System32. Yes, this is correct!

Both versions can be used simultaneously. Windows loads automatically the right version if you have copied the DLLs into the right directories.

Note that the DLL should be copied into the system folder on your development machine only so that Delphi is able to load it. The installer of your application should copy the DLL into the application directory instead.

General Usage

The Delphi interface encapsulates all DLL functions in the wrapper class TPDF. This class can be used like any other VCL class. The class is thread-safe and can be used without synchronization in multithreading applications.

However, some details must be known about the class. When the first instance is created, the constructor loads the dynapdf.dll with the API function LoadLibrary(). When creating a further instance of the wrapper class TPDF, also a new PDF instance is created inside the DLL. Each instance of the wrapper class uses its own DLL instance.

If an instance of the wrapper class TPDF is destroyed, the destructor deletes the used PDF instance; if no other instance uses the library then it will be unloaded with the API function FreeLibrary().

However, the DLL is unloaded each time if the reference count of the DLL is zero. In most cases it makes sense to hold one instance of the wrapper class in memory to avoid unloading the library. The internal resources used by DynaPDF are always freed when CloseFile() is called (except when the file is created in memory), so that there is no need to destroy the main instance of TPDF.

Exception handling in Delphi

DynaPDF itself uses no native Delphi exception handling. Error messages and warnings are passed to an error callback function if any (see SetOnErrorProc()). If no callback function is used, then use the function GetErrorMessage() to get information about the last error.

However, the wrapper class TPDF uses native exceptions in the following cases:

- When creating a new instance of the wrapper class TPDF.
- When loading a DLL function with the API function GetProcAddress() (**all functions**).

If a function listed above fails, then an exception is raised by the class TPDF. Always encapsulate all function calls into a try / except block. Only a few exceptions can occur but these exceptions must not be ignored. Especially when using DynaPDF in multi-threading applications it is highly recommended to use try / except or try / finally blocks. A thread must always catch all exceptions inside the thread.

Using DynaPDF in Multithreading Applications

The usage of DynaPDF inside a thread is the same as in single-threaded applications.

However, if a callback function should be used, you must make sure that the callback function is declared in the same thread or that each thread uses its own copy of the callback function. In addition, it is highly recommended that only thread-safe functions are called inside the callback function. If any unsafe function must be executed the function that causes the execution of the callback function must be synchronized because it is impossible to synchronize a callback function itself.

Threads should be used completely isolated from the main-thread of the application. Function calls to and from the main-thread must be synchronized. The entire PDF file should be created inside the thread including the instance of the wrapper class TPDF. The class instance must also be deleted before the thread is terminated.

A running thread can be terminated at any time but it is highly recommended to wait for any running functions to end before a thread will be terminated. This can be done easily by checking the property Terminated within the thread before a new function is executed.

After a running function returns, the class instance can be destroyed by using the Free() method for that instance. This will clean up the used resources and the thread can be terminated. The instance of the wrapper class TPDF can be safely destroyed at any time after a running function

returned. All internal used resources will be freed, there is no need to call FreePDF() manually beforehand.

Example (Single threaded):

In the following example we use a simple message box inside the error callback function. However, in a larger project it makes sense to output error messages into an error log or list box. DynaPDF ignores non-fatal errors by default so that it is possible to continue, but you can protocol each warning and errors during PDF creation.

```pascal
unit Unit1;

interface

uses Windows, Messages, SysUtils, Classes, Controls, Forms, Dialogs,
StdCtrls, dynapdf; // Include the file dynapdf.pas to the unit
type
  TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
 private
 public
end;
var Form1: TForm1;
implementation
// First, we define our callback function that is called if an
// error occurred. Note: The calling convention is stdcall!
function ErrProc(const Data: Pointer; ErrCode: Integer; const
ErrMessage: PAnsiChar; ErrType: Integer): Integer; stdcall;
var s: String;
begin
   s := Format('%s'#13'Abort processing?', [ErrMessage]);
   if MessageDlg(s, mtError, [mbYes, mbNo], 0) = mrYes then
      Result := -1 // break processing
   else
      Result := 0; // try to continue
end;
procedure TForm1.Button1Click(Sender: TObject);
var pdf: TPDF;
begin
   pdf := nil;
 try
   pdf := TPDF.Create;
   // set the error callback function first
   pdf.SetOnErrorProc(nil, @ErrProc);
   pdf.SetDocInfoA(diAuthor, 'Jens Boschulte');
   pdf.SetDocInfoA(diCreator, 'Delphi sample project');
   pdf.SetDocInfoA(diSubject, 'My first PDF file...');
```

```
    pdf.SetDocInfoA(diTitle, 'My first Delphi PDF output');
    pdf.SetViewerPreferences(vpDisplayDocTitle, avNone);

    pdf.CreateNewPDFA('c:\dout.pdf');
    pdf.Append;
    pdf.SetFontA('Arial', fsItalic, 40, true, cp1252);
    pdf.WriteFTextA(taCenter, 'My first Delphi output!!!');
    pdf.EndPage;
    pdf.CloseFile;
  except
    on E: Exception do MessageDlg(E.Message, mtError, [mbOK], 0);
  end;
  if pdf <> nil then pdf.Free;
end;
```

Compiling DynaPDF on Linux / UNIX

The build process of DynaPDF was designed to enable the compilation on most Linux or UNIX operating systems as easy as possible. All Linux and UNIX versions of DynaPDF can be compiled with GCC 4.2 or higher. GCC is freely available for most machine types.

> The pre-compiled Unix libraries are usually compiled with the default compiler that is common for the specific system, e.g. Solaris Studio on Solaris or Visual Age on AIX.

The configuration files are mainly designed for use with GCC and autoconf. If you want to use another compiler you must maybe change the compiler flags in the file configure.in and rebuild the configure script with autoconf. This can be done on an arbitrary Linux machine if no working autoconf is installed on the Unix machine.

If you get compilation issues just ask for a solution! We have a lot of experience with Unix systems, maybe we can help.

System requirements:

1. Properly installed GCC (4.2 or higher) C and C++ compiler. We strongly recommended GCC 4.2 or higher!
2. GNU make
3. To create a static library of DynaPDF you need also ar and ranlib

Build process

1. Copy first the entire directory dynapdf_ent to your Linux or UNIX machine.
2. Change the access permissions of the following files as follows (subdirectory /source):
   a. chmod 777 config.guess
   b. chmod 777 config.sub
   c. chmod 777 confrel
   d. chmod 777 install-sh

3. Type "./confrel" and press enter. This command creates the make files for your machine and starts the compilation.
4. Clean up the directory with "make clean", finished!

Make install creates a static and shared library of DynaPDF and copies the libraries and header files, which are required to bind DynaPDF, into the subdirectory /source.

You find the following files in the subdirectory /source after compiling DynaPDF:

- `dynapdf.h`   `// Main header file of DynaPDF`
- `drv_conf.h`   `// Required configuration file`
- `libdynapdf.a` `// Static library`
- `libdynapdf.so` `// Extension ".sl" on HP-UX or ".dylib" on Mac OS X`

Changing the configuration scripts

DynaPDF uses the freely available tool autoconf to create the main configuration script configure. Autoconf requires the file configure.in as input file which is located in the subdirectory /source. The final configure script can be executed without changes on all supported Linux and UNIX operating systems as well as on macOS. It creates the make files from the input files makefile.in which are located in all library directories; these files can normally be left unchanged. The top level makefile.in, which is stored in the subdirectory /source too, can be modified if further installation scripts should be executed after the library was successfully compiled. Note that the makefile.in files can be modified without rebuilding the configure script.

If you want to change certain compiler settings, or the compiler itself, modify the file configure.in and execute autoconf without parameters. Autoconf will rebuild the configure script with the new settings.

Linker flags

The used linker flags are designed to create a library with minimal dependencies so that DynaPDF can be delivered without other OS specific libraries. Depending on the target OS the linker flags can be changed so that OS specific libraries can be bind dynamically. This results in a smaller library but with more dependencies. To change the linker flags, modify the variable LD_LIBS in the file configure.in and rebuild the configure script with autoconf. The library can also be build LSB 3.0 compatible on Linux.

Compiler flags

When compiling DynaPDF on HP-UX the flag -fPIC (Position Independent Code) must be set at the minimum to enable the usage as shared library.

Optimization Level

DynaPDF is compiled with optimization level 3. This level is a good compromise between stable and fast code. The highest optimization level 4 causes a very long compilation time and it is possible that the resulting code is less stabile. Test the library properly before using this optimization level by default.

However, a release build should use the optimization level 3 or 4 because certain dependencies to internal GCC specific libraries are only removed if the optimization level is higher than 2.

Recommended compiler version

All Linux and Unix versions can be compiled with GCC 4.2. Older versions are not tested and maybe don't work. The GCC compiler should be configured with enabled POSIX compatible thread handling if possible, although the library does not depend on it.

Compiling DynaPDF on macOS

Compiling with XCode

DynaPDF Enterprise contains a pre-configured XCode workspace with targets for macOS and iOS. tvOS would probably work too but this was never tested.

The default target is macOS. An important thing to note is that the FreeType workspace has no separate target for iOS. Therefore, the target must be changed to iOS before the library can be compiled for this target. Changing the target is very easy, just click on the blue FreeType workspace, activate the tab "Build Settings", and select iOS as new base SDK:
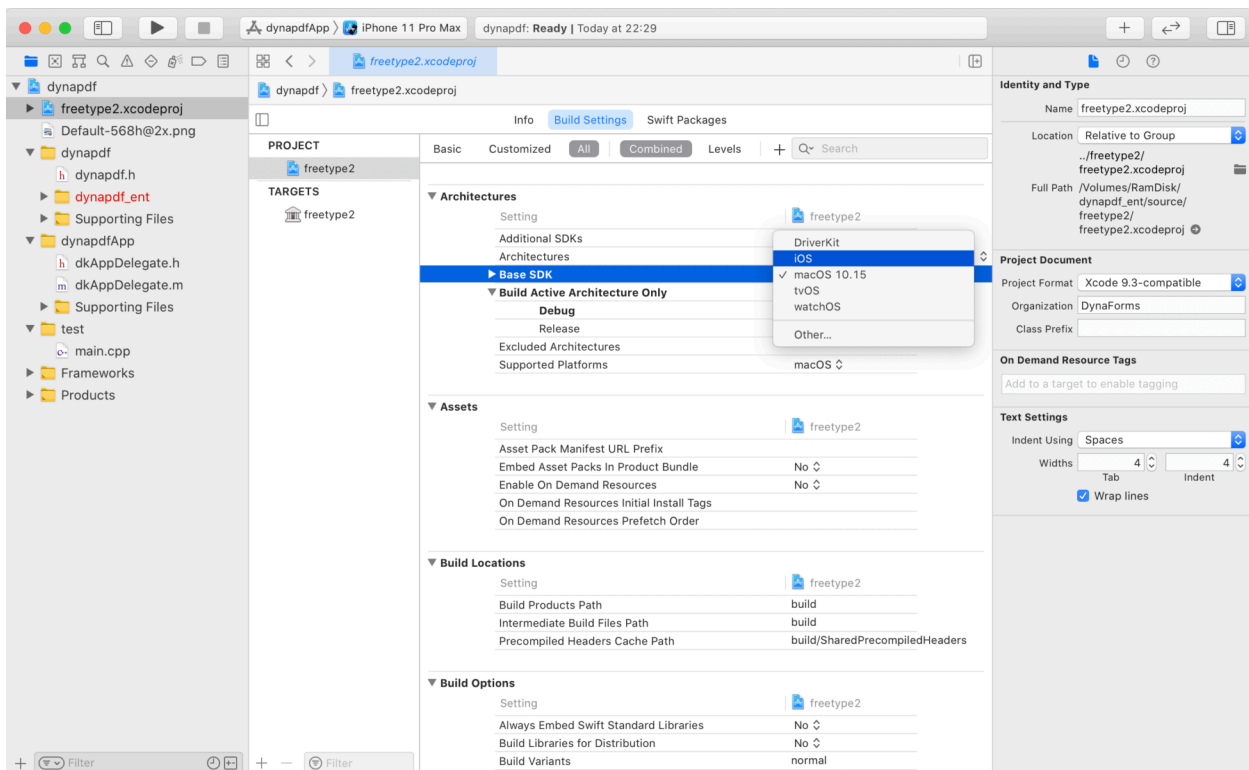
That's it, click on compile & run and the library will be compiled. When the scheme dynapdfApp is selected the emulator will start and a simple test app will be executed. This app calls just the DynaPDF function SelfText() to check whether the library works well. The output of SelfText() can be seen in XCode console.

Compiling on the Command Line

The build process via the command line is very similar in comparison to Linux and UNIX operating systems. On macOS we compile the library usually with CLang or with GCC if you work with an older Mac OS X version. All you need is a properly installed XCode including the

command line tools. XCode is freely available in the app store if not already installed on your system. The current configuration creates a Universal Binary for the targets x86_64 and ARM64.

Which targets are supported can be checked with the command *file*.

Example:

```
file libdynapdf.a
   or
file libdynapdf.dylib
```

If you don't want to create Universal binaries or if you need another CPU target then open the file configure.in in a text editor and modify the variable ARCH as needed.

Example (add the target Power PC 64):

ARCH="-arch x86_64 -arm64 -arch ppc64"

When finish save the file and execute autoconf. This command rebuilds the configure script. Finally type ./confrel and press enter. The configure script is now executed to build the make files and finally the script compiles the library. When finish execute make clean to clean up the build directory. The finish libraries are stored in the subdirectory /source.

Content parsing & editing

DynaPDF contains a very powerful content parser that is already used since several years by Optimize() and CheckConformance(). The very same parser can also be used to extract text, find and replace text, delete arbitrary operators, convert text to outline, convert colors, or downscaling and recompressing images, for example.

ParsePage() parses a page and creates a C object structure from it. This structure can be edited with various functions like Find- / ReplaceText(), or by DeleteOperator(), for example. Note that changes made on the C object structure do not take effect until the structure was written back with WriteToPage().

Include files

The parser API is defined in the file *parser_interface.h* (C/C++) or *CPDFContentParser.cls* for VB 6. Other programming languages do not need to include a separate file but the API is encapsulated in the class CPDFContentParser(), or TPDFContentParser() in Delphi respectively to make the usage easy as possible.

Abort

Syntax:

```
void psrAbort(
   const IPSR* Ctx) // PDF instance pointer
```

The function can be used to abort parsing whenever needed. However, parsing can be aborted only if ParsePage() was called in a separate thread. Note that it is not allowed to execute different functions of the same PDF instance in different threads. Every thread requires either its own PDF instance or function calls must be synchronized.

ChangeAltFont

Syntax:

```
LBOOL psrChangeAltFont(
   const PPDF* IPDF, // PDF instance pointer
   const IPSR* Ctx,  // Parser instance pointer
   UI32 Handle)      // Font handle
```

The function changes the active alternate font that is used by ReplaceSelText() when the original font is not available. The font handle must be a handle that was returned by SetFont(), SetFontEx(), SetCIDFont(), or SetAltFont().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ClearSelection

Syntax:

```
void psrClearSelection(
    const IPSR* Ctx) // Parser instance pointer
```

The function clears the selection. This function is declared for future use. FindText() clears the text selection automatically before it is executed. There is currently no need to clear the selection manually.

CreateParserContext

Syntax:

```
IPSR* psrCreateParserContext(
    const PPDF* IPDF,             // Instance pointer
    TOptimizeFlags Flags,         // Optimization flags
    struct TOptimizeParams* Parms) // Optional parameters
```

The function creates a parser context that can be used to edit and extract text, or do delete arbitrary operators of a page.

The content parser is used by Optimize() too. Therefore, the same flags and optimization parameters are supported. Please have a look at Optimize() for a description of the available flags and parameters. The parameter *Parms* can be set to NULL and *Flags* to *ofDefault* if nothing special should be achieved. *Parms* is considered only if the member *StructSize* is set to sizeof(TOptimizeParams).

The function makes a copy of the structure if set. This makes sure that no error occurs when the structure becomes out of scope.

Remarks:

To determine how the parser can be used, have a look at the example edit_text. The example is available for C/C++, C#, Delphi, VB .Net, and VB 6.

The parser context must be deleted with DeleteParserContext() when no longer needed.

Return values:

If the function succeeds the return value is a pointer of the parser context. If the function fails the return value is NULL.

DeleteOperator

Syntax:

```
LBOOL psrDeleteOperator(
   const IPSR* Ctx, // Parser instance pointer
   UI32 Index)      // Operator index
```

The function deletes an operator from the array of operators returned by ParsePage().

Please note that there is no error checking. Deleting operators can lead to errors, e.g. an operator like *BeginText* has the counterpart *EndText*, as well as additional operators inside this block which might not be allowed to occur standalone.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteOperatorInObject

Syntax:

```
LBOOL psrDeleteOperatorInObject(
   const IPSR* Ctx, // Parser instance pointer
   IPDFObj* ObjPtr, // Object pointer
   UI32 Index)      // Operator index
```

The function deletes an operator of an object that has its own content stream like templates, transparency groups, or patterns. The object pointer is provided in the structure of the corresponding operator (see TDrawTemplateOP, TDrawGroupOP, or TSetPatternOP). This is the member *Template* or *Pattern* of these structures.

For all other operators use DeleteOperator() instead.

Please note that there is no error checking. Deleting operators can lead to errors, e.g. an operator like *BeginText* has the counterpart *EndText*, as well as additional operators inside this block which might not be allowed to occur standalone.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteParserContext

Syntax:

```
void psrDeleteParserContext(
   IPSR** Ctx) // Parser instance pointer
```

The function deletes a parser context that was created by CreateParserContext() and sets the variable Ctx to NULL. If the parameter *Ctx* is NULL, the function returns immediately without error.

DeleteText

Syntax:

```
LBOOL psrDeleteText(
    const PPDF* IPDF,         // PDF instance pointer
    const IPSR* Ctx,          // Parser instance pointer
    struct TFltRect* Area,    // Required
    TDeleteTextFlags Flags)   // See below
```

The function deletes every glyph or character that touches or lies inside the rectangle *Area*.

Area must be defined as if the page would be viewed in a PDF viewer. That means in bottom up coordinates and the orientation must be considered (see GetPageOrientation()). The width and height of a page must be calculated from the crop box if set, or from the media box otherwise (see GetPageBBox()). Note also that the width and height must be exchanged if the orientation is 90, -90, 270, or -270 degrees.

Note that this function deletes text only. Text can also occur in form of images or vector graphics. There are no functions yet to identify and delete text in such objects.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ExtractText

Syntax:

```
LBOOL psrExtractText(
    const PPDF* IPDF,              // PDF instance pointer
    const IPSR* Ctx,              // Parser instance pointer
    TTextExtractionFlags Flags,   // See below
    struct TFltRect* Area,        // Optional search area
    UI16** Text,                  // Required, address of UI16* variable
    UI32* TextLen)                // Required, address of a UI32 variable

typedef enum TTextExtractionFlags
{
    tefDefault              = 0, // Create text lines in the original order.
    tefSortTextX            = 1, // Sort text records in x-direction.
    tefSortTextY            = 2, // Sort text records in y-direction.
    tefSortTextXY           = tefSortTextX | tefSortTextY,
    tefDeleteOverlappingText = 4  // Text extraction only.
}TTextExtractionFlags;
```

The function extracts the text of a page with the same algorithm that FindText() uses to find text on a page. In order to get exactly the same result the flag *tefSortTextX* must be set.

The function ExtractText() of the PDF instance calls in fact this function internally.

The optional parameter *Area* can be set to restrict text extraction to that rectangle. The rectangle must be defined as if the page would be viewed in a PDF viewer. That means in bottom up

coordinates and the orientation must be considered. The page coordinate system is de-rotated before text extraction starts since this produces better results. The width and height must be calculated from the crop box if set, or from the media box otherwise. Note also that the width and height must be exchanged if the orientation is 90, -90, 270, or -270 degrees.

Remarks:

Note that *Text* can be NULL, and *TextLen* zero, also if the function returned with no error. The page contained no text in this case.

Return value:

If the function succeeds the return value is 1. If the function fails the return value is 0.

FindText

Syntax:

```
LBOOL psrFindText(
    const PPDF* IPDF,                // PDF instance pointer
    const IPSR* Ctx,                 // Parser instance pointer
    struct TFltRect* Area,           // Optional search area
    TSearchType SearchType,          // See below
    struct TTextSelection* Last,     // The previous selection if any
    const UI16* Text,                // Search text
    UI32 TextLen,                    // Text length in characters
    struct TTextSelection* SelText)  // Required output structure

typedef enum TSearchType
{
    stDefault       = 0, // Case sensitive search
    stWholeWord     = 1, // Only whole words
    stCaseInSensitive = 2, // Case insensitive search
    stMatchAlways   = 4  // Return on every single character. Text and TextLen are ignored when
                         // this flag is set.
}TSearchType;
```

The function searches for text and stores the result so that further editing actions can be applied. The parameter *SelText* is required. The member *StructSize* must be set to sizeof(TTextSelection) before the function can be called. The parameter *Text* is required unless the flag *stMatchAlways* is set. When this flag is set, the function returns for every single character.

The bounding box of the found text can be computed with GetSelBBox().

Optional search area

Area must be defined as if the page would be viewed in a PDF viewer. That means in bottom up coordinates and the orientation must be considered (see GetPageOrientation()). The width and height of a page must be calculated from the crop box if set, or from the media box otherwise (see GetPageBBox()). Note also that the width and height must be exchanged if the orientation is 90, -90, 270, or -270 degrees.

The page coordinate system is de-rotated since this produces better results and it is much easier to find the location of text in rotated pages.

FindText() is usually called inside a loop until no more occurrences of the search string can be found. The search result of the last call must be passed to the parameter *Last* so that the function knows where to continue. In the first call, *Last* must be set to NULL. That means start at the beginning.

SelText and *Last* can point to the same structure. The values are copied before the function is executed.

Example (C++):

```
...
TTextSelection sel, *curr;
for (SI32 i = 1; i <= pdfGetPageCount(pdf); i++)
{
   curr = NULL; // Important! Must be NULL in the first call.
   if (psrParsePage(pdf, ctx, NULL, NULL, i, cpfEnableTextSelection, 0, &out))
   {
      while (psrFindText(pdf, ctx, NULL, stDefault, curr, sText, sLen, &sel))
      {
         psrReplaceSelText(pdf, ctx, rtfDefault, &sel, rText, rLen);
         curr = &sel; // Required! Otherwise the search run starts over and over again at the
                      // beginning and this would cause an endless loop.
      }
      psrWriteToPage(pdf, ctx, ofDefault, NULL);
   }
}
```

Return values:

If text was found the return value is 1. If no text was found or if an error occurred the return value is false.

GetSelBBox

Syntax:

```
LBOOL psrGetSelBBox(
   const PPDF* IPDF,          // PDF instance pointer
   const IPSR* Ctx,           // Parser instance pointer
   TTextSelection* SelText,   // Required. Structure filled with data by
                              // FindText().
   TBBox* BBox)               // Required. Address of a TBBox structure
```

The function computes the bounding box of the current selection created by FindText().

Notice:

The bounding box is calculated in top down coordinates! This differs in comparison to almost all other functions which return PDF coordinates.

In order to place objects on this position it is usually best to enable visible coordinates with SetUseVisibleCoords() and to rotate the coordinate system with SetOrientationEx() according to the page orientation after the page was opened for editing. See example on the next page.

Example:

This example draws a rectangle over every letter to check whether the coordinates are correct.

```c
void TestContentParser(const PPDF* PDF, const wchar_t* InFile, wchar_t* OutFile, SI32 Page)
{
    IPSR* ctx;
    IPGE* page;
    SI32 i, j, n, cnt;
    TBBox b, list[8192];
    TContent out = {0, 0};
    TTextSelection sel, *curr;

    pdfCreateNewPDF(PDF, NULL);
    // No need to override the original producer
    pdfSetDocInfo(PDF, diProducer, NULL);

    // Do not convert pages to templates
    pdfSetImportFlags(PDF, ifImportAll | ifImportAsPage);
    // Reduce the memory usage
    pdfSetImportFlags2(PDF, if2UseProxy);
    pdfOpenImportFileW(PDF, InFile, ptOpen, NULL);
    if (!Page)
        pdfImportPDFFile(PDF, 1, 1.0, 1.0);
    else
    {
        pdfAppend(PDF);
            pdfImportPageEx(PDF, Page, 1.0, 1.0);
        pdfEndPage(PDF);
    }

    cnt = pdfGetPageCount(PDF);
    ctx = psrCreateParserContext(PDF, ofDefault, NULL);

    sel.StructSize = sizeof(sel);

    // Important! GetSelBBox() returns top down coordinates.
    pdfSetPageCoords(PDF, pcTopDown);

    for (i = 1; i <= cnt; i++)
    {
        curr = NULL;
        if (psrParsePage(PDF, ctx, NULL, NULL, i, cpfEnableTextSelection, 0, &out))
        {
            n = 0;
            // The function returns on every single letter if the flag stMatchAlways is set
            while (psrFindText(PDF, ctx, NULL, stMatchAlways, curr, NULL, 0, &sel))
            {
                if (psrGetSelBBox(PDF, ctx, &sel, &b) && n < 8192)
                    list[n++] = b;
                curr = &sel;
            }
            if (n > 0)
            {
                pdfEditPage(PDF, i);
                    pdfSetOrientationEx(PDF, pdfGetOrientation(PDF));
                    pdfSetLineWidth(PDF, 0.0);
                    for (j = 0; j < n; j++)
                    {
                        TBBox &r = list[j];
                        pdfRectangle(PDF, r.x1, r.y1, r.Width(), r.Height(), fmClose);
```

```
                }
                pdfClosePath(PDF, fmStroke);
            pdfEndPage(PDF);
        }
    }
}
psrDeleteParserContext(&ctx);

if (pdfOpenOutputFileW(PDF, OutFile) && pdfCloseFile(PDF))
    printf("Ok   : \"%S\"\n", OutFile);
else
    printf("Error: \"%S\"\n", OutFile);
}
```

Return value:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetSelBBox2

Syntax:

```
LBOOL psrGetSelBBox2(
    const PPDF* IPDF,                // PDF instance pointer
    struct TTextSelection* SelText, // Required. Structure filled with
                                     // data by FindText().
    struct TFltPoint QuadPoint[4])  // Required. Array of TFltPoint
```

The function returns the bounding box as quad points of a single node or glyph of the current selection. This function is useful for rotated text.

The four points are computed in top down coordinates and in the following order:
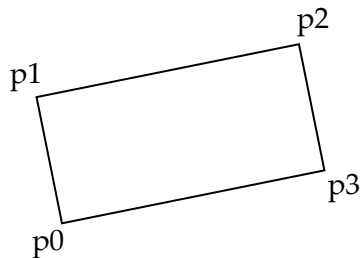
The structure *TTextSelection* is a single linked list. The next node, if any, is stored in the member *Next*.

Example (C++):

This code shows how to access all nodes of a selection but note that the contents of a page will be overriden when WriteToPage() will be called. Therefore, it is sometimes required to temporarily store the quad points in an array so that they can be drawn after WriteToPage() was called.

```
...
void DrawSelection(PPDF* PDF, TTextSelection &Curr)
{
    TFltPoint q[3];
```

```
    TTextSelection* next = &Curr;
    while (next)
    {
        if (psrGetSelBBox2(PDF, next, q))
        {
            if (fabs(q[0].y - q[3].y) < 0.0001f) // Is this a rotated node?
                pdfRectangle(PDF, q[0].x, q[0].y, q[3].x-q[0].x, q[1].y-q[0].y, fmStroke);
            else
            {
                pdfMoveTo(PDF, q[0].x, q[0].y);
                pdfLineTo(PDF, q[1].x, q[1].y);
                pdfLineTo(PDF, q[2].x, q[2].y);
                pdfLineTo(PDF, q[3].x, q[3].y);
                pdfClosePath(PDF, fmStroke);
            }
        }
        next = next->Next;
    }
}
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetSelText

Syntax:

```
LBOOL psrGetSelText(
    const PPDF* IPDF,                 // PDF instance pointer
    struct TTextSelection* SelText,   // Required. Structure filled with
                                      // data by FindText().
    UI16** Text,                      // Address of UI16* variable
    UI32* TextLen)                    // Address of UI32 variable
```

The function retrieves the text and text length of the current selection or found text.

This function is mainly a helper function for programming languages which do not support pointers like VB 6, VB .Net, C#, and so on. However, the function is also useful if FindText() was called with the search type *stMatchAlways*. In this case, and only in this case, it is not known which text FindText() has found.

In C/C++, or Delphi you can access the text directly as follows:

```
UI16* text    = selText->TextOP->Text->UniText;
UI32  textLen = selText->TextOP->Text->UniTextLength;
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ParsePage

Syntax:

```
LBOOL psrParsePage(
   const PPDF* IPDF,                      // PDF instance pointer
   const IPSR* Ctx,                       // Parser instance pointer
   const void* UserData,                  // Optional user data
   struct TPDFParseCallbacks* Funcs,      // Optional callback functions
   UI32 PageNum,                          // Page number to parse
   TContentParseFlags Flags,              // See below
   struct TContentParseParms* Parms,      // Optional parameters
   struct TContent* Out)                  // Required output variable

typedef enum TContentParseFlags
{
    cpfDefault            = 0x00000000, // Nothing special to do.
    cpfComputeBBox        = 0x00000001, // Compute bounding boxes of all objects.
    cpfFlattenLayers      = 0x00000002, // Flatten layers.
    cpfSkipInvisibleObjects = 0x00000004, // Ignore invisible objects.
    cpfFullRecursive      = 0x00000008, // Parse all objects recursively.
    cpfNoInlineTemplate   = 0x00000010, // Do not resolve templates if reference count = 1.
    cpfCalcDeviceColors   = 0x00000020, // Compute device colors of all colors which are set in
                                        // the content streams.
    cpfImidiateMode       = 0x00000040, // Internal. This flag is always set if the
                                        // TPDFParseCallbacks structure is passed to
                                        // ParsePage(). It disables certain optimisations.
    cpfNewLinkNames       = 0x00000080, // Internal. Create new link names for all objects.
                                        // Used by Optimize() and CheckConformance().
    cpfEnableTextSelection = 0x00000100, // This flag is required to enable text selection and
                                        // text extraction.
    cpfInitMatrix         = 0x00000200, // The transformation matrix must be set in the
                                        // TContentParseParms structure.
    cpfSkipClipPaths      = 0x00000400, // Useful for debugging purposes.
    cpfSkipImages         = 0x00000800, // Ignore all images. This flag is useful for text
                                        // extraction.
    cpfSkipShadings       = 0x00001000, // Useful for debugging purposes.
    cpfSkipText           = 0x00002000, // Useful for debugging purposes.
    cpfSkipVector         = 0x00004000  // Useful for debugging purposes. Exclude vector
                                        // graphics with exception of clipping paths.
}TContentParseFlags;

#pragma pack(1)
struct TContentOP
{
    BYTE  OP;    // Operator to excecute.
    void* Param; // This pointer is set for operators which have parameters.
};
#pragma pack()

struct TContent
{
    UI32              Count; // Number of available operators.
    struct TContentOP* OP;   // Array of operators.
};
```

The function parses a page and stores the page contents in a C object structure. Once a page was parsed various functions can be called, e.g. to extract the text of a page, to find and replace text, or to delete arbitrary operators.

The page that should be parsed should be closed, that means it should not be opened for editing beforehand with EditPage() or Append(). The function can parse an open page too but this can lead to errors and is not recommended.

At time of publication only the minimal functionality is documented but more and more examples and documentation will be added.

The parameters *UserData* and *Funcs* are currently available in C/C++ only. The callback API can be used for immediate mode. That means the callback functions are executed immediatly when a corresponding operator was found. This is comparable with ParseContent() but not identically because the C object structure is still build so that the page contents can be modified.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ReplaceSelText

Syntax:

```
LBOOL psrReplaceSelText(
   const PPDF* IPDF,                // PDF instance pointer
   const IPSR* Ctx,                 // Parser instance pointer
   TReplaceTextFlags Flags,         // See below
   struct TTextSelection* SelText,  // Required -> output of FindText()
   const UI16* Text,                // Optional replacment text
   UI32 TextLen)                    // Length in characters
```

The function replaces or deletes the text that was found by FindText(). Coordinates of surrounding text are not changed, this means that the new text can overlap surrounding text if the width of the new text is larger than the original text.

So, text replacement has its limitations since at some point text must might be new formatted or re-aligned. However, replacing placeholders with text is usually no problem as long as there is enough room for the new text.

Note that placeholders should not contain space characters since spaces are often not stored in PDF files and this can lead to issues finding the text.

Font substitution

Text replacement depends on the availability of the fonts which are used in a PDF file. If the original font is not available the function loads an alternate font that matches the characteristics of the original font as close as possible. However, font substitution is not perfect and a substituted font looks sometimes more different as expected.

To improve text replacement it is possible to set one or more alternate fonts which should be used if the original font cannot be found on the system. Alternate fonts can be loaded with

SetAltFont(). However, a font loaded by SetFont(), SetFontEx() or SetCIDFont() works too. In order to activate a font loaded by a regular font loading function call ChangeAltFont().

It is possible to load more than one alternate font but only the active font will be used when replacing text. If more than one font must be loaded, store the handle returned by SetAltFont() and change the font with ChangeAltFont() whenever needed.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAltFont

Syntax:

```
SI32 psrSetAltFont(
   const PPDF* IPDF, // PDF instance pointer
   const IPSR* Ctx,  // Parser instance pointer
   const char* Name, // Font name
   TFStyle Style,    // Font style
   double Size,      // Font size (must be greater zero)
   LBOOL Embed,      // If true, the font will be embedded
   TCodepage CP)     // Code page to be used
```

The function sets the font as alternate font that is used by ReplaceText() if the original font is not available.

Although the parameter *Size* must be greater zero, the value is in fact not used when replacing text. The parameter is reserved for future use.

Please note that the replacement text must be defined in the code page with which the font was loaded.

To effiently change the font whenever needed, call ChangeAltFont(). It is also possible to call SetAltFont() but this would require more processing time.

Return values:

If the function succeeds the return value is the font handle, a value greater of equal zero. If the function fails the return value is 0.

WriteToPage

Syntax:

```
LBOOL psrWriteToPage(
    const PPDF* IPDF,               // PDF instance pointer
    const IPSR* Ctx,                // Parser instance pointer
    TOptimizeFlags Flags,           // Flags to control optimization
    struct TOptimizeParams* Parms)  // Optional optimization parameters
```

The function writes the C object structure, that was created by ParsePage(), back to the page. The flags and optional parameters were taken from Optimize() because Optimize() uses the very same parser to optimize pages. Please have a look at this function to determine which flags and parameters are available.

Unchanged pages can be left unchanged or written back to the page. This is up to you. If WriteToPage() is called then the content stream will be optimized. If nothing special should be achieved set *Flags* to *ofDefault* and *Parms* to NULL.

CreateParserContext() accepts already its own flags and TOptimizeParams structure. The parameter *Parms* that was passed to this function is used for all pages unless WritePage() contains its own version.

If the parameter *Parms* is set then this structure becomes the new default until it is overriden again by a WriteToPage() call. The flags are always overriden.

The function makes a copy of the structure if set. This makes sure that no error occurs when the structure becomes out of scope.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Complex Text Layout

Support for complex text layout is required for languages like Arabic, Thai, Devanagari, Lao, and many others. At time of publication DynaPDF supports complex text layout on Windows only.

The implementation in DynaPDF based on Microsofts Uniscribe. This is the Unicode library that is used behind the scenes in almost all Windows controls and Unicode aware applications.

Complex text layout depends on the availability of OpenType fonts. Most fonts which are delivered with Windows are OpenType fonts.

How to enable Complex Text Layout?

Complex text layout can be enable with the flag *gfComplexText* with SetGStateFlags(). Please note that this feature requires DynaPDF Professional or higher.

Automatic Font Substitution

A major feature of complex text layout is called Automatic Font Substitution. That means if the current font doesn't support the requested script, it will be automatically changed if another font is available that supports it.

Although automatic font substitution produces often good results without further interaction, a human would often use another font. In order to improve automatic font substitution it is possible to create one or more alternate font lists. The active list of alternate fonts is tested before any system font. This makes it possible to improve automatic font substitution because humans have usually a better knowledge which fonts are common in their region.

In simple text layout, only one font is used to output a line of text. In complex text layout it is generally possible that more than one font is used, e.g. if a piece of text is not supported by the current font.

However, automatic font substitution can be implemented in many different ways. Some applications try to replace the font whenever a glyph cannot be found. Others change the font only for entire words or sentences. The challenge is to make all requested glyphs available without embedding a huge number of fonts and without requiring too much processing time.

DynaPDF changes a font only for entire script items. A script item is a run of characters with compatible properties and with the same direction. This is a good compromise between speed, document size, and resulting output quality.

Alternate font lists

Automatic font substitution can be improved by setting an alternate font list with fonts which support the language that must be output. It is possible to create several different lists, e.g. for different font styles or languages.

The alternate font list can be changed arbitrary often with no additional overhead.

The function to manage alternate font lists are:

- `ActivateAltFontList() // Activate or deactivate a font list`
- `CreateAltFontList()   // Create a font list`
- `DeleteAltFontList()   // Delete a font list`
- `SetAltFonts()         // Fill a list with data`

Font embedding

With exception of CJK scripts (Chinese, Japanese, Korean) complex text layout depends on font embedding. Fonts must be embedded because glyphs are usually selected with OpenType layout features. Such glyphs have mostly no Unicode code point and can therefore not be selected with simple algorithms.

Chinese, Japanese, and Korean scripts are exceptions, because the PDF format supports special features like CID Fonts in combination with external cmaps, so that fonts for these scripts must not necessarily be embedded. This is very important because Asian documents contain usually much more glyphs in comparison to other languages. Therefore, such documents are much larger. Non embedded CID fonts can be loaded with SetCIDFont().

Complex text layout and form fields

One thing that must be considered is that the support level of complex text layout in Adobes Acrobat is limited. The support level is comparable with web browsers. That means basic shaping and the bidirectional algorithm are available, but Unicode bidi control characters are not supported and mixed languages produce not always correct results.

Since the appearance of form fields should ideally not change when a user activates a field in Acrobat, DynaPDF strips out all bidi control characters when setting a field value.

This makes sure that the appearance stream can be created in a comparable way and Acrobat is able to display a field value correctly.

The very same result can be achieved with regular text functions by setting the flag *gfDisableBidiCtrls* with SetGStateFlags().

JSON Parser

Since version 4.0.74.216, DynaPDF contains a JSON parser (Javascript Object Notation) that simplifies the exchange of complex data structures a lot.

The JSON parser supports a few extensions to make the definition of additional attributes as easy as possible.

Differences in comparison to regular JSON grammar:

- A JSON string begins normally with a left brace ({) and ends with a right brace (}). This outer left / right brace pair can be omitted.

- Name or key / value pairs must normally be delimited by a colon (:). The delimiting colon can be omitted.

- Array values must normally be delimited by a comma (,). The delimiting comma can be omitted if the bounds of a value can be detected by whitespace or starting character of an object.
  Example (C/C++):
  const char attr[] = "\"TestArray\"[1/2\"3\"4 5]";

  Result:
  /TestArray[1/2(3)4 5]

  That means the first value of the array is the number 1, the second is a name object with the value "2", the third value is the string "3", the fourth is the number 4, and the fifths is the number 5.

- Extension: A Solidus (/) begins a PDF Name object. The solidus is not part of the name but is a prefix indicating what follows is a sequence of characters representing a name object. A name object supports character codes between 33..126 inclusive. Any character outside this range must be written as two digit hexadecimal code, preceded by a number sign (#), e.g. /This#20is#20a#20name#20with#20spaces. Note that a number sign in a name object must be written as hexadecimal code too, since the number sign is treated as an escape character. The de-escaped string is interpreted as UTF-8 Unicode string.

  A name object is terminated by character codes < 33, { (left brace), } (right brace), [(left bracket),] (right bracket), , (comma), : (colon), and " (double quotes). If one of these characters should be part of a name object then write it as two digit hexadecimal code preceded by a number sign (#).

  No predefined attribute key or value is defined in PDF that would require an escaping mechanism.

- A left brace ({) creates a PDF dictionary and a right brace (}) finishes or closes it.

- A left bracked ([) creates an array and a right bracked (]) finishes or closes it.

Data types supported as value of a key are: array, dictionary, name, string, boolean true or false (case sensitive), null, and number. A JSON string must be enclosed with double quotes. Example: "This is a string".

In PDF, keys are always defined as name objects. The function converts a key (or name in JSON nomenclatur) automatically to a name object. However, no such automatic conversion is possible for values since the function does not interpret the resulting PDF keys and therefore, it does not know which datatype a key accepts.

In order to output a string as name object add a slash as first charactert to the string, e.g. "Test name object value" : "/PDFNameObject".

Examples (C/C++):

The examples below show that a JSON string can contain various errors which would not be accepted by a regular JSON parser:

```
const char attr[] = "{/BBox[50, 450, 200, 500]/Placement/Inline}";   // Ok
const char attr[] = "/BBox :[50 450 200 500]/Placement/Inline";       // Ok
const char attr[] = "{{{{/BBox :[50 450 200 500]/Placement/Inline";   // Ok
const char attr[] = "/BBox :[50 450 200 500]/Placement/Inline}}}}";   // Ok
const char attr[] = "\"BBox\":[50 450 200 500]\"Placement\"/Inline";  // Ok
const char attr[] = "/BBox,[50 450 200 500]\"Placement\"/Inline";     // Ok
const char attr[] = "/BBox:[50 450 200 500]/Placement/Inline";        // Ok
```

Interactive Forms

DynaPDF supports a large set of functions to create and edit form fields incl. predefined actions and JavaScript actions. This section describes how interactive forms can be created and how certain features can be used.

Field Appearance

Interactive Form Fields support user defined background, text and border colors, as well as different border styles. These properties can be set or changed with following functions:

Global properties for new created fields:

- `Get/SetBorderStyle()`   `// Border style`
- `Get/SetFieldBackColor()`   `// Background color`
- `Get/SetFieldBorderColor()`   `// Border color`
- `Get/SetColorSpace()`   `// Color space`
- `Get/SetFieldTextColor()`   `// Text color`
- `Get/SetLineWidth()`   `// Line width of the border`

Functions to change the appearance or format of a field:

- `SetFieldBBox()`   `// Changes the field's bounding box`
- `Get/SetFieldBorderStyle()`   `// Border style`
- `Get/SetFieldBorderWidth()`   `// Line width of the border`
- `Get/SetFieldColor()`   `// Background, border, or text color`
- `SetFieldFont()`   `// Set or change the field font`
- `SetFieldFontEx()`   `// Set or change the field font`
- `SetFieldFontSize()`   `// Changes the field's font size`
- `Get/SetFieldHighlightMode()` `// Highlight mode`
- `SetDateTimeFormat()`   `// Set a date or time format`
- `SetNumberFormat()`   `// Set a number format`

Global field appearance properties:

- `Get/SetNeedAppearance()`   `// See below`

Calc order:

- `Get/SetFieldCalcOrder()`   `// Get or change the calc order`

Other:

- `DeleteAcroForm()`   `// Delete the entire AcroForm`
- `DeleteXFAForm()`   `// Delete the XFA form of a hybrid form`
- `DeleteJavaScripts()` `// Delete global Javascripts and JS Actions`
- `LoadFDFData()`   `// Load form data from a FDF file`

The global NeedAppearance flag of an Interactive Form defines whether the viewer should create the field appearances on demand when opening the file or whether the existing definitions should be taken from the PDF file. DynaPDF creates always appearance streams for

all field types with exception of barcode fields. However, in certain cases it can be useful to let the viewer render fields with their own algorithms because the exact way how Adobe's Acrobat builds the field appearances is not documented.

For example, when editing the contents of a text field in Adobe's Acrobat the viewer rebuilds first the field appearance before placing the editing cursor into the field. The new appearance created from Adobe's Acrobat can be slightly different in comparison to the one that was created by DynaPDF. The visible content, especially of text fields, is sometimes not absolutely stable.

If the NeedAppearance flag is set, the viewer uses already its own algorithms to build the field appearances when opening the file. This avoids visible changes when editing a field. However, the NeedAppearance flag must not be set to true if a form contains page templates.

Important field properties when creating new fields

The line width of the field border is derived from the current graphics state when a new field is created (see SetLineWidth()). No border will be drawn if either the line width is set to zero or if the border color is set to NO_COLOR (see SetFieldBorderColor()). The default background color for new fields is NO_COLOR; that means the background appears transparent. Form fields support the color spaces DeviceGray, DeviceRGB, and DeviceCMYK. The default background, border, and text color must be defined in the current color space. Note that DynaPDF does not convert the current color values if the color space will be changed.

Field Properties

Most field properties and values can be read and changed with DynaPDF. The following list gives an overview over the available functions and for what they can be used.

- `AddValToChoiceField()`   // List & combo boxes
- `GetFieldChoiceValue()`   // List & combo boxes, radio buttons
- `GetFieldCount()`   // Number of fields in the document
- `GetFieldEx()/GetFieldEx2()` // Most important properties
- `GetFullyQualifiedFieldName()`// Returns the fully qualified field name
- `GetPageFieldCount()`   // Number of fields of a page
- `GetFieldExpValCount()`   // Number of values/export values
- `Get/SetFieldExpValue()`   // Export value of a field
- `GetFieldExpValueEx()`   // Value and export value pair
- `GetFieldGroupType()`   // Base type of a field group
- `GetFieldType()`   // Field type
- `SetAnnotOrFieldDate()`   // Sets the modification date
- `SetCheckBoxState()`   // Changes the state of a check box
- `SetCheckBoxDefState()`   // Changes the default state of a check box
- `Get/SetFieldHighlightMode()` // Field highlight mode
- `Get/SetFieldIndex()`   // Index to change the tab order
- `Get/SetFieldFlags()`   // Field flags

```
•  Get/SetFieldMapName()       // Mapping name -> export name
•  Get/SetFieldName()          // Field name
•  Get/SetFieldOrientation()   // Field orientation
•  Get/SetFieldTextAlign()     // Text alignment of a text field
•  Get/SetFieldToolTip()       // Tool tip or field description
•  Get/SetTextFieldValue()     // Text field value or default value
```

GetFieldEx() is the most important function to retrieve field properties. Fields can be accessed via the global handle or via the index within the field array of the current page.

What is a Group Type?

The field type ftGroup is used for different purposes depending on how fields are organized. A normal group field is used to create a logical hierarchy or group of fields but group fields are also used to create a so called "Field Group".

A normal group field is a set of fields which use a parent field of type ftGroup to achieve a logical hierarchy, e.g. Address.Street, Address.Country, and so on. The field and group type of the parent group field are both set to ftGroup in this case.

One important thing must be considered when accessing children of a group field: The group field does not occur in the field array of a page; it is only available in the global AcroForm field array. So, while GetFieldEx() returns the entire field array including group fields GetPageFieldEx() returns never group fields! The parent group field can be accessed with the Parent handle of the children in this case.

The second usage of a group field is to achieve a so called "Field Group". A Field Group is an array of fields of the same type which share the same name and value. Such fields are internally organized into a special kind of group field which holds the field name and value, as well as other properties which can be shared among the group.

The children of a Field Group have no name. So, if a field contains no name then you can already assume that it is part of a Field Group because the field name is required to be present otherwise. Children of a Field Group contain always the handle to the parent group field. The field type of this group field is set to ftGroup as usual but the group type is set to a field type other than ftGroup (see GetFieldGroupType()).

The unambiguous test whether a group field is an ordinary group field or a terminal field of a Field Group is to compare the group type with the field type. If the group type is something else than ftGroup then this is a terminal field of a Field Group.

In this case the field contains the field name of the children as well as the field value, default value, and tooltip. The field flags, background, border and text color, border width, and the border style are inherited from the terminal field but can be overridden by the children.

Please note that Adobe's Designer creates mostly Field Groups also if only one child is part of the group.

When changing a value or property of a Field Group there is nothing special that must be considered. DynaPDF sets the wished value or property automatically to the right field.

How to create a Field Group?

Field groups can be created in two ways: When creating two fields with the same name and type then DynaPDF creates automatically a field group. However, it is also possible to set the handle of the parent field to indicate that the field should be added to this field as a child. The latter variant is a little bit faster.

Example:

```
...
SI32 prt = pdfCreateTextField(pdf, "Test", -1,false,-1,50, 50,150,20);
           pdfCreateTextField(pdf, "Test",prt,false,-1,50, 80,150,20);
           pdfCreateTextField(pdf, "Test",prt,false,-1,50,110,150,20);
...

or

// Same result but requires more processing time
pdfCreateTextField(pdf, "Test", -1, false, -1, 50,  50, 150, 20);
pdfCreateTextField(pdf, "Test", -1, false, -1, 50,  80, 150, 20);
pdfCreateTextField(pdf, "Test", -1, false, -1, 50, 110, 150, 20);
```

See also section "Fields with identical names".

How to change the tabulator order?

The form fields and annotations of a page are stored in an array. The order of fields in this array represents the tab order. New fields are added to the array in the order in which they were created. To enable the definition of an arbitrary tab order, each form field and annotation holds a page index variable which can be used to sort the fields.

The page indexes of new or imported fields start always at index 1000. The indexes of annotations start at 10000. Form fields are in fact Widget Annotations and all annotations of a page are stored in the same array.

Because the indexes of form fields start at index 1000 it is easier to move a field to the beginning of the array because the indexes from 0 to 999 are free when starting to change the tab order.

The page indexes can be set to any value with SetFieldIndex(), but no field or annotation of a page should use the same index. No error occurs if two fields or annotations use the same index but the order of these fields is then of course undefined.

When all indexes are set, the fields must be sorted with SortFieldsByIndex() so that the new tabulator order can be applied. Note that the function sorts the fields of the current open page. If no open page is in memory then the function will fail.

Field Names

Interactive Form Fields are identified over the field name in a viewer application. A field name is an Ansi string that should be human readable. Beginning with PDF 1.5 field names can also be defined as Unicode string. However, all functions to create new fields in DynaPDF support Ansi strings only. All characters within the Ansi character set (code page 1252) can be used with exception of the period character (.) and characters below index 32 .

A field name should also not end with a space character because Adobe's Acrobat is then sometimes unable to access such a field with a JavaScript Action or function.

The period (.) is a reserved character because it is used to build the fully qualified field name in a viewer application. The fully qualified field name is constructed from the partial field name of the field and all of its ancestors.

For a field with no parent group field, the partial and fully qualified names are the same. For a field that is the child of another field, the fully qualified name is formed by appending the child field's partial name to the parent's fully qualified name, separated by a period, e.g. Address.Street.

Fields with identical names

It is possible to create two or more fields of the same type which use all the same name. Such fields contain always the same value if the value of one field of the group is changed.

Fields with identical names are internally represented as a special type of field group, which is automatically created by DynaPDF. This makes the handling more complicated because the children of such a group do not contain a field name. The name is set to the parent's group field but not to the children of the group. This can normally be ignored but when enumerating fields with GetField() or GetPageField() you must consider that not all fields contain a name, the parent field's handle is set instead.

However, with the exception described above, field names must be unique within the hierarchy in which they appear. This is especially important when multiple Interactive Forms are imported.

When importing multiple Interactive Forms it is highly recommended to check for invalid duplicate field names. This can be done with the function CheckFieldNames(). The function returns the handle of the first field which contains a field name that is already in use. You can then change the field name with SetFieldName() and execute CheckFieldNames() again until all invalid field names are changed.

After changing a field name you must also check whether the field is used within a JavaScript Action or function. Such scripts must be changed so that they do not become invalid. Due to the possible references of fields within JavaScript functions and Actions, merging of Interactive Forms is very complicated and should be avoided whenever possible.

Actions

Annotations, form fields, bookmarks (also known as outline items), pages, and the global Catalog object may specify an action to perform, such as opening another PDF file, jumping to page, or playing a sound, for example.

Annotations (especially link annotations) and bookmarks can dirctly be associated with an action. This action will be executed when the object is activated.

Annotations, the catalog object, pages, and form fields support also additional actions which extend the set of events which trigger the execution of an action.

Actions are usually executed in viewer applications only. Otherwise it would be very difficult to understand what happens behind the scenes when editing an object.

Actions can be accessed with the following functions:

- GetObjActions() / GetObjEvent()
- GetGoToAction()
- GetGoToRAction()
- GetHideAction()
- GetImportDataAction()
- GetJavaScriptActionEx()
- GetLaunchAction()
- GetMovieAction()
- GetNamedAction()
- GetURIAction()

GetObjActions() returns the first action of an object and a pointer to the first trigger event if any.

Actions and trigger events are stored as a single linked list. That means, every action and every trigger event can reference another action or event that should be executed.

The actions which must be executed for an object should be copied to an execution list and not directly be executed. This makes sure that a duplicate check can be applied so that no endless loop occurs when an action references itself.

The first object that should be examined is the catalog object right after a PDF file was opened with OpenImportFile() or OpenImportBuffer() and after the global objects were imported with ImportCatalogObjects().

Now render the first page and add the actions of the page to the event list of this page. After this the actions of form fields should be examined because form fields support events which must be handled when a page was opened, when it become visible or invisible and when it will be closed.

After this bookmarks can be imported with ImportBookmarks(). Because bookmarks support no trigger events the application can load actions of a bookmark one demand in the OnMouseUp event of the bookmark.

PDF objects which support trigger events do not all support the entire list of available events. The following table shows which trigger events are supported by which objects. Bookmarks are not listed here because bookmarks do not support trigger events.

Event	Catalog	Annotations	Fields	Pages
oeOnOpen	•			•
oeOnClose				•
oeOnMouseUp		•	•	
oeOnMouseEnter			•	
oeOnMouseExit			•	
oeOnMouseDown			•	
oeOnFocus			•	
oeOnBlur			•	
oeOnKeyStroke			•	
oeOnFormat			•	
oeOnCalc			•	
oeOnValidate			•	
oeOnPageVisible			•	
oeOnPageInVisible			•	
oeOnPageOpen			•	
oeOnPageClose			•	
oeOnBeforeClosing	•			
oeOnBeforeSaving	•			
oeOnAfterSaving	•			
oeOnBeforePrinting	•			
oeOnAfterPrinting	•			

Form fields must be examined for actions right after a page was loaded. Otherwise it is not possible to handle the highlighted events.

Digital Signatures

A digital signature (PDF 1.3) can be used to authenticate the identity of a user and the document's contents. It stores information about the signer and the state of the document when it was signed. Once a PDF file was digitally signed it is impossible to change the file without invalidating the signature. Because of this, it is always possible to check whether a document has been changed or not.

Depending on the Acrobat version certain signature handlers are supported by Adobe's Acrobat. DynaPDF supports the PPKLite security handler which is supported since Acrobat 4.0.

Supported Certificate Formats

DynaPDF supports internal and external signature handlers. When using the internal signature handler of DynaPDF then you need a PKCS#12 certificate file. Certificates are available in different file formats and different encryption key lengths. DynaPDF supports certificates in the file format PKCS#12 with up to 4096 bits encrypted private/public key pairs on Windows.

On non-Windows operating systems the cross-platform signature library AiCrypto is used to sign PDF files. This signature handler supports 1024 bit RSA encrypted private keys only (the AiCrypto library supports almost all available key lengths but it creates indefined length encoded ASN1 objkects for strong encryption key lengths wheras Adobe's Acrobat supports defined length encoded ASN1 objects only).

The internal signature handler is mainly used with self-sign certificates but it is possible to sign a PDF file with any certificate that is installed on the system's certificate store, including hardware certificates.

External Signatures

In order to support software and hardware certificates with almost arbitrary encrpytion key lengths it is possible to sign a PDF file with an external signature handler. This makes it possible to select a certificate from the sytem's certificate store and to use system functions, for example, to sign a PDF file.

The function CloseAndSignFileExt() can be used to create detached and non-detached signatures. In case of a non-detached signature CloseAndSignFileExt() returns the SHA1 hash of the PDF file and the external signature handler signs this hash and creates a PKCS#7 signature object that must finally be written to the PDF file with FinishSignature().

A detached signature works almost identically with the exception that the signature handler creates also the hash from the PDF buffer to be signed. This variant is not recommended for programming languages which support no pointers like C# or VB .Net, for example, because an additional copy of the PDF buffer must usually be created and this doubles the memory usage

and requires additional processing time. However, detached signatures enable the usage of other hash algorithms than SHA1.

How to export a Windows Certificate?

To export a Windows certificate proceed as follows (description for Windows XP or higher): open the control panel and double click on the icon "Internet Options". Click on the tab "Contents" and then on the button "Certificates…". Select a certificate from the list and click on the button "Export…". The option "Export private key" must be selected (this option is not available if a certificate contains no private key). The private key is required; certificates without a private key cannot be used to sign PDF files. On the next dialog you must enter a password to encrypt the private key; this password must later be passed to the function CloseAndSignFile() or CloseAndSignFileEx() if the file should also be encrypted. Enter now the file name and path of the certificate file, finished! The result is a certificate file with the extension *.pfx, this file can now be used to digitally sign PDF files.

Importing signed PDF files

Signed PDF files can only be changed, without invalidating an existing signature, when changes are stored with an incremental update. An incremental update is a special way to modify a PDF file; changes are appended to the end of the file, leaving its original contents intact. This technique is required since altering any existing bytes in the file invalidates existing signatures.

However, incremental updates are not supported by DynaPDF that is the reason why only empty signature fields can be imported. Because DynaPDF creates always a completely new PDF file, it makes no sense to import existing signatures, they would always become invalid.

How to sign a PDF file?

Signing a PDF file with the internal signature handler of DynaPDF is quite easy; all you need is a PKCS#12 certificate file. Instead of calling the function CloseFile() or CloseFileEx() after the document was created, call either CloseAndSignFile() or CloseAndSignFileEx() if the file should also be encrypted, finished! A digital signature is always stored in a signature field. If no signature field was created beforehand, DynaPDF creates an invisible signature field on the first page and stores the signature in this field.

If the PDF file should be signed with an external signature handler call CloseAndSignFileExt(), sign the provided hash or byte ranges, and finally finish the signature with FinishSignature() to insert the signed PKCS#7 object into the PDF file.

How to create a signature field?

As mentioned above, the function CloseAndSignFile() or CloseAndSignFileEx() creates an invisible signature field on the first page if no signature field was already created or imported beforehand. If the signature field should be visible, just create one with the function

CreateSigField() on the page where the field should appear. If multiple signature fields exist, DynaPDF uses the last signature field to sign the PDF file.

How to modify the appearance of a signature field?

The appearance of a signature field can be fully user defined. The function CreateSigFieldAP() can be used to create a user defined signature appearance template. You can draw anything you want into this template such as images, vector graphics, text, and it is also possible to import a PDF page into or to draw an EMF file into the template.

What is stored in a signature field?

When signing a PDF file a signature handler, whether internal or external, creates a PKCS#7 signature object that contains the file's signature, optionally a time stamp, and a PKCS#1 certificate that was extracted from the PKCS#12 certificate. The difference between PKCS#1 and PKCS#12 is that a PKCS#1 certificate contains no private key.

A viewer application validates the signature by using the public key that is stored in the PKCS#1 certificate object. Because the private key is not stored in the PDF file it is impossible to sign other PDF files with the certificate that can be extracted from the PDF file.

Adobe's Acrobat supports defined length encoded ASN1 objects only for encryption key lengths stronger than 1024 bits.

How to validate a signature?

Acrobat validates signatures from unknown certificates not automatically. The certificate must first be added to the list of trusted identities. Once a certificate was added to the list of trusted identities, signatures of other documents which use the same certificate will be automatically validated.

PDF/A and PDF/X Compatibility

PDF files can be created for different purposes such as printing, publishing, or archiving which have all their own requirements. Due to these different requirements two separate PDF standards were defined by the ISO Committee, PDF/X and PDF/A.

PDF/X

The PDF/X-1a standard addresses blind exchanges where all files should be delivered in CMYK (and/or spot colors), with no RGB or device independent (color-managed) data. This is a common requirement in many areas around the world and in many print sectors – usually tied to an environment where the file supplier wants to retain maximum control of the print job.

PDF/X 3 is like PDF/X 1a an ISO standard for graphic content exchange. The main difference is that PDF/X 3 allows the use of color management and device-independent color in addition to CMYK and spot colors.

The PDF/X standard requires all fonts to be embedded, the appropriate PDF bounding boxes to be specified, and color to appear as CMYK, spot colors, or both. In addition, PDF/X compliant PDF files must contain information describing the printing condition for which they are prepared (see AddRenderingIntent()).

When creating PDF/X compliant files with DynaPDF you need to know that DynaPDF does not check whether certain features are allowed to use in the selected PDF/X standard. DynaPDF simply writes the required PDF/X key to the file which tells the viewer application that this file is compliant to a specific PDF/X version. Whether this is true or not depends on whether you used allowed features only and whether all required information were added to the file, e.g. the rendering intent (see AddRenderingIntent()), the document title (see SetDocInfo()) and the trim box for each page (see SetBBox()). It is usually best to check the resulting PDF file with a preflight tool before using certain features in a production environment.

However, it is not very difficult to create PDF/X compliant PDF files. The main recommendation is that all fonts are embedded, that at the least the trim box for all pages are set, and that colors are defined in the color space DeviceCMYK (see SetColorSpace()). In addition, an ICC profile must be embedded in the file (see AddRenderingIntent()) and images must not be compressed with JPEG2000 compression.

The wished output PDF version must be set with SetPDFVersion().

Example (C++):

```cpp
// Error callback function.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return -1; // We break processing if an error occurs
}

int main(int argc, char* argv[])
{
   void* pdf = pdfNewPDF();
   if (!pdf) return 2; // Out of memory?

   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfCreateNewPDF(pdf, "c:/cppout.pdf");
   pdfSetDocInfo(pdf, diCreator, "C++ Example project");
   pdfSetDocInfo(pdf, diTitle, "PDF/X Compatibility");

   pdfAppend(pdf);
   // Just set the trim box to the same value as the media box if no
   // better value is known.
   TPDFRect b;
   pdfGetBBox(pdf, pbMediaBox, b);
   pdfSetBBox(pdf, pbTrimBox, b.Left, b.Bottom, b.Right, b.Top);

   // The font must be embedded (this should always be the case)
   pdfSetFont(pdf, "Arial", fsItalic, 20.0, true, cp1252);
   pdfSetColorSpace(pdf, csDeviceCMYK);
   pdfSetFillColor(pdf, PDF_CMYK(0, 0, 0, 255));
   pdfWriteFText(pdf, taCenter,
      "A very simple PDF/X compliant PDF file...");
   pdfEndPage(pdf);
   // The PDF version should be set before the file is closed because
   // it can be changed when importing a PDF file.
   pdfSetPDFVersion(pdf, pvPDFX1a_2001);
   pdfAddRenderingIntent(pdf,
      "c:/WINNT/System32/spool/drivers/color/USWebCoatedSWOP.icc");
   pdfCloseFile(pdf);

   pdfDeletePDF(pdf);
   return 0;
}
```

PDF/A

PDF/A is an ISO standard for long-term preservation. These files are primarily used for archiving. PDF/A compliant files can contain text, raster images, vector graphics, as well as annotations, hyperlinks, or bookmarks.

However, PDF/A compliant files must not contain JavaScripts or an Interactive Form. In addition, all fonts must be embedded and PDF/A compliant files must contain information describing the printing condition for which they are prepared (see AddRenderingIntent()). The output intent can be either CMYK or RGB based. However, only one device color space can be used in a document with the exception that DeviceGray can be combined with RGB or CMYK color spaces.

When creating PDF/A compatible files with DynaPDF it is important to know that DynaPDF does not automatically check whether certain features are allowed to use. DynaPDF writes simply the required PDF/A key to the file which tells the viewer application that the file is compatible to a specific PDF/A standard. Whether this is true or not depends on whether prohibited features were used and whether all required information were added to the file, e.g. the rendering intent.

However, DynaPDF contains the function CheckConformance() to check and convert non-conformant PDF files to PDF/A 1b. The function was originally developed a very powerful PDF to PDF/A converter called myPDFConvert by the DETEC GmbH in Germany.

All DynaPDF versions provide a restricted version of CheckConformance() that does not convert imported PDF files to PDF/A. The conversion of imported PDF files is possible if DynaPDF was licensed with the PDF/A Extension.

CheckConformance() is a very good helper function to get your PDF file fully PDF/A 1b compatible. CheckConformance() is not a preflight function, it automatically adjusts anything that is possible to get the file PDF/A 1b compatible. The function supports a large set of flags to specify what should be done if prohibited features were found in the file. Take a look into the function description for further information.

Example (C++):

```
// Error callback function.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   // We do not break processing if an error occurs. However, if the
   // file was fully created with DynaPDF we should not receive any
   // warning or other errors.
   return 0;
}
```

```c
int main(int argc, char* argv[])
{
   SI32 retval;
   char outFile[] = "c:/cppout.pdf";
   void* pdf = pdfNewPDF();
   if (!pdf) return 2; // Out of memory?
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfCreateNewPDF(pdf, NULL); // The output file is opened later
   pdfAppend(pdf);
   // The font must be embedded (this should always be the case).
   pdfSetFont(pdf, "Arial", fsItalic, 20.0, true, cp1252);
   pdfWriteFText(pdf, taCenter, "A very simple PDF/A 1b file...");
   pdfEndPage(pdf);
   // Check whether the file is compatible to PDF/A 1b
   retval = pdfCheckConformance(pdf,ctPDFA_1b_2005,0,NULL,NULL,NULL);
   switch(retval)
   {
      case 1: pdfAddRenderingIntent(pdf,"sRGB.icc"); break; // RGB
      case 2: pdfAddRenderingIntent(pdf,"CMYK.icc"); break; // CMYK
      case 3: pdfAddRenderingIntent(pdf,"sRGB.icc"); break; // Gray
      default: break;
   }
   // Note: CheckConformance() raises a fatal exception if the file
   // cannot be converted to PDF/A 1b. We check first whether a PDF
   // file is still in memory before we try to open the output file.
   if (pdfHaveOpenDoc(pdf))
   {
      if (!pdfOpenOutputFile(pdf, outFile))
      {
         pdfFreePDF(pdf);
         return -1;
      }
      retval = pdfCloseFile(pdf);
   }
   pdfCloseFile(pdf);
   pdfDeletePDF(pdf);
   return retval;
}
```

DeviceGray is the only device color space that can be combined with RGB or CMYK color spaces. Because special DeviceGray ICC color profiles are rarely available you can safely use a RGB or CMYK ICC profile instead.

Path Painting and Construction

A vector graphics in PDF consists of paths. A path can be used to draw something on screen or to clip a specific area. A path itself is invisible until it was filled, stroked, or both. In addition, clipping and path painting operators can be combined, e.g. a clipping path can again be stroked or filled.

> **Notice:**
> Although the combination of clipping and path painting operators is supported in all PDF versions, not all viewer applications support this operator combination. To avoid unnecessary problems paths should be clipped and painted in two separate steps, also if the same path must be output twice in this case.

When working with paths two specific rendering techniques should be known to understand for what certain path painting operators are useful:

Nonzero Winding Number Rule

The *nonzero winding number rule* determines whether a given point is inside a path by conceptually drawing a ray from that point to infinity in any direction and then examining the places where a segment of the path crosses the ray. Starting with a count of 0, the rule adds 1 each time a path segment crosses the ray from left to right and subtracts 1 each time a segment crosses from right to left. After counting all the crossings, if the result is 0 then the point is outside the path; otherwise it is inside.

Note: The method just described does not specify what to do if a path segment coincides with or is tangent to the chosen ray. Since the direction of the ray is arbitrary, the rule simply chooses a ray that does not encounter such problem intersections.

For simple convex paths, the nonzero winding number rule defines the inside and outside as one would intuitively expect. The more interesting cases are those involving complex or self-intersecting paths like the ones shown in Figure below. For a path consisting of a five-pointed star, drawn with five connected straight line segments intersecting each other, the rule considers the inside to be the entire area enclosed by the star, including the pentagon in the center.
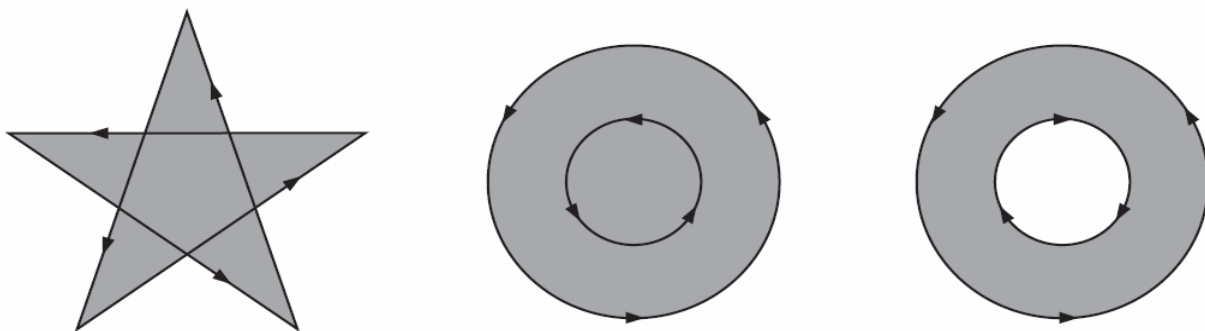
For a path composed of two concentric circles, the areas enclosed by both circles are considered to be inside, provided that both are drawn in the same direction. If the circles are drawn in opposite directions, only the "doughnut" shape between them is inside, according to the rule; the "doughnut hole" is outside (the draw direction for closed shapes can be changed with the property Get/SetDrawDirection()).

Even-Odd Rule

An alternative to the nonzero winding number rule is the even-odd rule. This rule determines the "insideness" of a point by drawing a ray from that point in any direction and simply counting the number of path segments that cross the ray. If this number is odd, the point is inside; if even, the point is outside. This yields the same results as the nonzero winding number rule for paths with simple shapes, but produces different results for more complex shapes.

The Figure below shows the effects of applying the even-odd rule to complex paths. For the five-pointed star, the rule considers the triangular points to be inside the path, but not the pentagon in the center. For the two concentric circles, only the "doughnut" shape between the two circles is considered inside, regardless of the directions in which the circles are drawn.

The rules described above are applied on vector graphics and clipping paths. DynaPDF supports three basic functions to fill, stroke, or to mark a given path as clipping path. A clipping path can in turn be filled, stroked, or both.

The basic path operating functions are:

- ClosePath()
- ClipPath()
- StrokePath()

ClosePath() and ClipPath() support a large set of path painting operators. Let us take a look at the available path painting constants:

```
typedef enum
{
    // Nonzero Winding Number Rule
    fmFillNoClose,
    fmStrokeNoClose,
```

```
    fmFillStrokeNoClose,
    fmFill,
    fmStroke,
    fmFillStroke,
    // Even-Odd Rule
    fmFillEvOdd,
    fmFillStrokeEvOdd,
    fmFillEvOddNoClose,
    fmFillStrokeEvOddNoClose,
    fmNoFill,                   // Special meaning see below
    fmClose                     // Close the path
}TPathFillMode;
```

The constant fmNoFill produces different results when using it with a function that produces a closed shape like Rectangle(), Ellipse() and so on, or when using it with ClosePath() or ClipPath().

Within ClipPath() the constant has the meaning, do neither fill or stroke the clipping path, just create the clipping area.

Within ClosePath() the constant can be used to discard the previously drawn path. The path in memory will be deleted and nothing will be written to the PDF file.

Within Rectangle(), Ellipse() and so on, the constant closes the path, the result is exactly the same as if fmClose would be used.

Let us now take a look on the different path painting rules.

If we draw two rectangles in the same draw direction the result is different depending on the used filling rule:

```
...
pdfSetFillColor(pdf, PDF_RGB(120, 120, 220));
pdfRectangle(pdf, 50.0, 50.0, 250.0, 100.0, fmNoFill);
pdfRectangle(pdf, 70.0, 70.0, 210.0, 60.0, fmNoFill);
pdfClosePath(pdf, fmFill); // Non-Zero Winding Number
...
```

Output:

Winding (fmFill):   Even Odd (fmFillEvOdd):

If we change the draw direction of the second rectangle to ddClockwise (default is ddCounterClockwise) then we get the same result as with Even Odd:

```
...
pdfSetFillColor(pdf, PDF_RGB(120, 120, 220));
pdfRectangle(pdf, 50.0, 50.0, 250.0, 100.0, fmNoFill);
pdfSetDrawDirection(pdf, ddClockwise); // Change the draw direction
pdfRectangle(pdf, 70.0, 70.0, 210.0, 60.0, fmNoFill);
pdfClosePath(pdf, fmFill); // Non-Zero Winding Number
...
```

Output:

```
Winding (fmFill):                    Even Odd (fmFillEvOdd):
```


The filling rules work also with clipping paths and more complex paths:

```
#include "dynapdf.h"
// First we declare our error callback function
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage); return 0;
}

void DrawPath(const void* PDF, double x, double y, TClippingMode Mode,
TDrawDirection Direction, UI32 Color)
{
   pdfSaveGraphicState(PDF);
   pdfSetFillColor(PDF, Color);
   pdfTriangle(PDF, x+100, y+50, x+150, y+150, x+50, y+150, fmNoFill);
   pdfSetDrawDirection(PDF, Direction);
   pdfTriangle(PDF, x+100, y+180, x+50, y+80, x+150, y+80, fmNoFill);
   pdfClipPath(PDF, Mode, fmFillStroke);
   pdfSetFont(PDF, "Arial", fsBold | fsItalic, 40, true, cp1252);
   pdfSetFillColor(PDF, 0);
   pdfWriteText(PDF, x+50, y+90, "CLIPPING");
   pdfRestoreGraphicState(PDF);
}

int main(int argc, char* argv[])
{
   void* pdf = pdfNewPDF();
   if (!pdf) return 2;
```

```
    pdfSetOnErrorProc(pdf, NULL, PDFError);
    pdfSetDocInfo(pdf, diAuthor, "Jens Boschulte");
    pdfSetDocInfo(pdf, diCreator, "C sample project");
    pdfSetDocInfo(pdf, diSubject, "How to use clipping paths");
    pdfSetDocInfo(pdf, diTitle, "Clipping paths");
    pdfSetViewerPreferences(pdf, vpDisplayDocTitle, avNone);
    pdfCreateNewPDF(pdf, "c:/cppout.pdf");
    pdfSetPageCoords(pdf, pcTopDown);
    pdfAppend(pdf);
    // Nonzero Winding Number Rule
    DrawPath(pdf, 0, 0, cmWinding, ddCounterClockwise, PDF_SKYBLUE);
    DrawPath(pdf, 110, 0, cmWinding, ddClockwise, PDF_SKYBLUE);
    // Even Odd Rule
    DrawPath(pdf, 220, 0, cmEvenOdd, ddCounterClockwise, PDF_YELLOW);
    DrawPath(pdf, 330, 0, cmEvenOdd, ddClockwise, PDF_YELLOW);
    pdfEndPage(pdf);
    pdfCloseFile(pdf);
    pdfDeletePDF(pdf);
}
```


Winding   *Even-Odd*
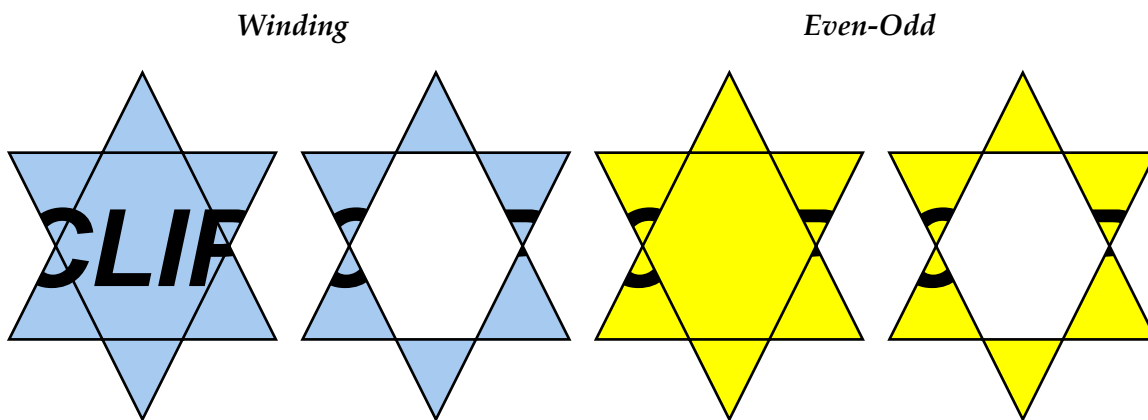
A clipping path is part of the graphics state. The only way to disable a clipping path is to restore the graphics state. If the graphics state was not saved beforehand it is impossible to deactivate the clipping region.

If a second clipping path is drawn while another one is still active then it will be intersected with the current clipping path. The clipping area of the intersected clipping paths is never larger than the initial clipping path.

Color Spaces

DynaPDF supports all color spaces defined in PDF 1.7. This section describes how color spaces can be created and used with DynaPDF.

Color spaces in PDF can be divided into 3 types:

- Device color spaces such as DeviceRGB, DeviceCMYK, and DeviceGray.

- Device independent color spaces such as CalGray, CalRGB, Lab, and ICCBased. All these color spaces are CIE based color spaces. The ICCBased color space is a special case of a CIE based color space; it uses a color profile to convert colors into the destination space.

- Special color spaces like Indexed, Separation, DeviceN, or NChannel. Pattern is also a special color space but not listed here because DynaPDF uses separate functions to create and apply pattern color spaces.

Device Color Spaces

The device color spaces enable a page description to specify color values that are directly related to their representation on an output device. Color values in these spaces map directly (or by simple conversions) to the application of device colorants, such as quantities of ink or intensities of display phosphors. This enables a PDF document to control colors precisely for a particular device, but the results may not be consistent from one device to another.

Device color spaces are supported within page descriptions and for interactive objects like annotations and form fields. While form fields support all three available device color spaces, annotations support DeviceRGB only. This behavior must be taken into account when creating such objects. The use of an unsupported color space results in an error and the wished object is not created.

The use of device color spaces is very easy because no creation of a color space object is required before such a color space can be used. All three device color spaces can be set directly with the functions SetColorSpace(), SetFillColorSpace(), and SetStrokeColorSpace().

Once the color space was set the functions SetFillColor(), SetStrokeColor(), or SetColors() accept color values of the particular color space:

- DeviceGray controls the intensity of achromatic light, on a scale from black to white. It ranges from 0 to 255 inclusive.

- DeviceRGB controls the intensities of red, green, and blue light, the three additive primary colors used in displays. RGB colors can be created with the macro PDF_RGB() or with comparable functions like rgb() which are available in most programming languages. See also Color Values.

- DeviceCMYK controls the concentrations of cyan, magenta, yellow, and black inks, the four subtractive process colors used in printing. CMYK colors can be created with the macro

PDF_CMYK() or with the function CMYK() defined in the DynaPDF interfaces for the used programming language. See also Color Values.

Device color spaces can be set directly with the functions SetColorSpace(), SetFillColorSpace(), and SetStrokeColorSpace().

Device Independent Color Spaces

Calibrated color in PDF is defined in terms of an international standard used in the graphic arts, television, and printing industries. CIE-based color spaces enable a page description to specify color values in a way that is related to human visual perception. The goal is for the same color specification to produce consistent results on different output devices, within the limitations of each device. PDF 1.1 supports three CIE-based color space families, named CalGray, CalRGB, and Lab; PDF 1.3 adds a fourth, named ICCBased.

While CalGray, CalRGB, or Lab use relatively simple formulas to convert colors into a device color space, an ICCBased color space uses a color profile instead. A color profile offers much more control over the color conversion process to match the characteristics of an output device precisely.

Device independent color spaces require a color space object that must be created before the color space can be used:

- CreateCIEColorSpace() creates a CalGray, CalRGB, or Lab color space.
- CreateICCBasedColorSpace() creates an ICCBased color space.

The functions above return a color space handle on success which is required to set the color space with SetExtColorSpace(). Color values for CIE based color spaces can be created in the very same way as for device color spaces (see previous section) with the exception of Lab colors which require a special treatment depending on the used programming language (see CreateCIEColorSpace() for further information).

Special Color Spaces

Special color spaces add features or properties to an underlying color space. There are four special color space families: Pattern, Indexed, Separation, and DeviceN. The Pattern color space is not described here since DynaPDF provides separate functions to create and apply pattern color spaces (see BeginPattern() for further information).

Indexed Color Space

An Indexed color space allows a PDF content stream to use small integers as indices into a color map or color table of arbitrary colors in some other space. A PDF consumer application treats each sample value as an index into the color table and uses the color value it finds there. Indexed color spaces are mostly used to reduce the amount of data for sampled images. The usage within a page description is possible but there is normally no advantage to do so. DynaPDF creates indexed color spaces for sampled images automatically if the original data was organized in this way.

However, it is also possible to create an Indexed color space manually and to apply it to an image. This can be useful if a DeviceN color space should be used for an image, e.g. to create a duotone image. An Indexed color space can be used for images with a color depth of 1, 2, 4, or 8 bits with or without a color table. An already existing color table stored in the image file will be replaced with the one of the Indexed color space.

DynaPDF assigns an indexed color space to an image if the graphics state flag gfUseImageColorSpace is not set (see SetGStateFlags()) and if an indexed color space is active when inserting the image.

The color table of the Indexed color space should contain 2 ^BitDepth color values. It is allowed to use a color table with less colors but this is bad practice and should be avoided.

An indexed color space can be created with the function CreateIndexedColorSpace() and must be set with SetExtColorSpace(). If you want to use the color space for vector graphics or texts pass the wished index to SetFillColor(), SetStrokeColor(), or SetColors(). The color table is zero based; index 0 refers to the first color; NumColors -1 refers to the last color in the table. An Indexed color space can contain up to 256 color values.

Separation Color Space

Color output devices produce full color by combining primary or process colorants in varying amounts. On an additive color device such as a display, the primary colorants consist of red, green, and blue phosphors; on a subtractive device such as a printer, they typically consist of cyan, magenta, yellow, and sometimes black inks. In addition, some devices can apply special colorants, often called spot colorants, to produce effects that cannot be achieved with the standard process colorants alone. Examples include metallic and fluorescent colors and special textures.

When printing a page, most devices produce a single composite page on which all process colorants (and spot colorants, if any) are combined. However, some devices, such as imagesetters, produce a separate, monochromatic rendition of the page, called a separation, for each colorant. When the separations are later combined - on a printing press, for example - and the proper inks or other colorants are applied to them, the result is a full-color page.

A Separation color space (PDF 1.2) provides a means for specifying the use of additional colorants or for isolating the control of individual color components of a device color space for a subtractive device. When such a space is the current color space, the current color is a single-component value, called a tint, that controls the application of the given colorant or color components only.

A color value in a Separation color space consists of a single tint component in the range 0 to 255. The value 0 represents the minimum amount of colorant that can be applied; 255 represents the maximum. Tints are always treated as subtractive colors, even if the device produces output for the designated component by an additive method. Thus, a tint value of 0 denotes the lightest color that can be achieved with the given colorant, and 255 is the darkest. This convention is the same as for DeviceCMYK color components but opposite to the one for DeviceGray and DeviceRGB.

If the colorant name associated with a Separation color space does not correspond to a colorant available on the device, the application arranges for subsequent painting operations to be performed in an alternate color space. The intended colors can be approximated by colors in a device or CIE-based color space, which are then rendered with the usual primary or process colorants.

The alternate color space is a required parameter of the function CreateSeparationCS(). Therefore, if a CIE based color space should be used as alternate color space then this color space must be created before the Separation color space.

Once the color space was created it can be set with SetExtColorSpace().

A Separation color space can be applied to vector graphics and images with a color depth of 1, 2, 4, or 8 bits. See also Color Spaces and Images.

DeviceN Color Space

DeviceN color spaces (PDF 1.3) can contain up to 32 color components. They provide greater flexibility than is possible with standard device color spaces such as DeviceCMYK or with individual Separation color spaces. For example, it is possible to create a DeviceN color space consisting of only the cyan, magenta, and yellow color components, with the black component excluded.

DeviceN color spaces are used in applications such as these:

- High-fidelity color is the use of more than the standard CMYK process colorants to produce an extended gamut, or range of colors. A popular example is the PANTONE Hexachrome system, which uses six colorants: the usual cyan, magenta, yellow, and black, plus orange and green.

- Multitone color systems use a single-component image to specify multiple color components. In a duotone, for example, a single-component image can be used to specify both the black component and a spot color component. The tone reproduction is generally different for the different components. For example, the black component might be painted with the exact sample data from the single-component image; the spot color component might be generated as a nonlinear function of the image data in a manner that emphasizes the shadows.

DeviceN was designed to represent color spaces containing multiple components that correspond to colorants of some target device. As with Separation color spaces, PDF consumer applications must be able to approximate the colorants if they are not available on the current output device, such as a display. To accomplish this, the color space definition provides a tint transformation function that can be used to convert all the components to an alternate color space.

While the transformation function is automatically created for a Separation color space this is not possible for DeviceN color spaces. The transformation function consists always of a Postscript Calculator Function and it is a required parameter of the function CreateDeviceNColorSpace().

Note that the alternate color space is a required parameter too. Therefore, if a CIE based color space should be used as an alternate color space then this color space must be created before the DeviceN color space can be created.

Once the color space was created it can be set with SetExtColorSpace(). Colors of a DeviceN color space should be set with SetFillColorEx() or SetStrokeColorEx() because these functions accept up to 32 color channels.

A DeviceN color space can be applied to text, vector graphics and images with a color depth of 1, 2, 4, or 8 bits, and in addition to images with up to 32 color channels using 8 bits per component. See Color Spaces and Images below and CreateDeviceNColorSpace().

Color Spaces and Images

All color spaces can be used with text, vector graphics, and images. While the usage with text and vector graphics requires no further considerations the usage with images is more complicated.

Prior versions of DynaPDF supported device color spaces only and due to this limitation images were always converted to the nearest device color space that was active when inserting an image. This default conversion is not optimal if the original image is defined in a device independent color space. However, in certain circumstances it's a nice to have feature, especially when creating PDF/X-1a compatible PDF files (this standard prohibits the usage of device independent color spaces).

To preserve backward compatibility the following default rules apply depending on the active device color space (the color conversion rules can be modified, see below):

Active color space	Description
DeviceGray	Images defined in another color space are converted to DeviceGray. 1 bit images are always left unchanged.
DeviceRGB	Images defined in a color space with 3 or more color components are converted to DeviceRGB. 1 channel images are converted to DeviceGray. 1 bit images are left unchanged.
DeviceCMYK	Images defined in a 3 channel color space are converted to DeviceCMYK. This conversion is applied to preserve backward compatibility. 1 channel images are converted to DeviceGray. 1 bit images are left unchanged.

Handling of non-device color spaces

Non-device color spaces can be used to replace an image color space with the active color space. The active color space and the image color space must be compatible. For example, a Lab color space cannot be replaced with a CalRGB color space because this would require a physical conversion of the pixel data. The source and destination color space must contain the same number of color components. If the active color space is not compatible to the image color space DynaPDF reports an error and the image is not inserted.

The following rules apply only if the flag *gfUseImageColorSpace* is **not** set (default). See SetGStateFlags() for further information.

Active color space	Description
CalGray	The color space can be set to 1 channel images without a color table. The image data should be defined in a color depth of 1 or 8 bits per pixel. Color depths of 2 or 4 bits are unusual and maybe not supported in all versions of Adobe's Acrobat or on the output device. Consider the special treatment of 1 bit images.
CalRGB	The image color space can be any 3 channel color space with exception of Lab.
ICCBased	The image color space and the input color space of the corresponding ICC profile must be compatible.
Indexed	The color space can be set to 1 channel images with or without a color table. An already existing color table in the image file will be replaced. Consider the special treatment of 1 bit images.
Separation	The color space can be set to 1 channel images without a color table. The image data should be defined in a color depth of 1 or 8 bits per pixel. Color depths of 2 or 4 bits are unusual and maybe not supported in all versions of Adobe's Acrobat or on the output device. 1 channel images without a color table are normally defined in DeviceGray or CalGray but these color spaces are additive color spaces while Separation is a subtractive color space. To get an image in a compatible range DynaPDF inverts the pixel data. This behaviour must be taken into account when creating images for a Separation color space. Consider also the special treatment of 1 bit images.
DeviceN	The supported bit depths and image color spaces depend on the number of color components defined in the DeviceN color space and which function is used to insert the image. The default function to insert images defined in a DevicN color space is InsertRawImage(). This function supports images with up to 32 color channels.

However, it is also possible to exchange the color space of arbitrary images with a DeviceN color space. The following sub clauses describe the handling if the image is loaded from an arbitrary image file with InsertImage(), InsertImageEx(), or InsertImageFromBuffer().

Images with less than 4 color channels will be inverted because DeviceN is a subtractive color space and images with less than 4 color components are defined in an additive color space in normal image files.

Num Comp.	Supported image formats
1	The image can be any 1 channel image without a color table. However, the image data should be defined in a color depth of 1 or 8 bits per pixel. Color depths of 2 or 4 bits are unusual and maybe not supported in all versions of Adobe's Acrobat or on the output device.

	2	DeviceN color spaces with 2 color components can be used with 1 bit images without a color table.
	3	The image color space can be any 3 channel color space with exception of Lab.
	4	The image data must be defined in DeviceCMYK.
	up to 32	Images with more than 4 color components are supported by InsertRawImage() only.

Special treatment of 1 bit images

1 bit images without a color table are treated as an image mask if image transparency is enabled (see SetUseTransparency()). In this case black pixels are drawn in the current fill color while white pixels produce no visible output. The fill color can be defined in an arbitrary color space.

If image transparency is disabled the active color space will be assigned to the image as long as it is compatible to the image color space which is usually DeviceGray. Note that a 1 bit image with a color table is treated as a color image. However, this format is unusual and not supported by all devices and Acrobat versions. The usage of this image format should be avoided if possible.

How to preserve the image color space?

The default conversion rules can be changed if necessary with SetGStateFlags(). The most important flag is *gfUseImageColorSpace*. If set, the color space is taken from the image file and the active color space is ignored.

Another important flag is *gfIgnoreICCProfiles*. If set, an ICC profile available in the image file is not used to create an ICCBased color space for the image. The image is inserted in the base color space in this case. This flag is not meaningful is the flag *gfUseImageColorSpace* is absent.

The creation of ICCBased color spaces should only be enabled if it is known that the image contains an ICC profile that is compatible to the output device.

Note that many PDF workflows are not able to replace the ICC profile of an ICCBased color space and this can cause big problems if an image contains a profile that is not compatible to the output device.

It is usually better to add a global rendering intent to the document (see AddRenderingIntent()) instead of using automatically created ICCBased color spaces on a per image basis.

A rendering intent is essentially the same as an ICCBased color space but it is used for all images in the file, as well as for text and vector graphics, which are defined in the underlying base color space. The advantage is that the ICC profile of a rendering intent can be easily replaced with another one and this change affects the entire document. The entire document should of course be created in the same base color space, e.g. DeviceCMYK, DeviceRGB, DeviceGray, or Lab.

Image Resolution

DynaPDF supports a high quality scaling algorithm to enable downscaling of high resolution images to a specified output resolution (see SetResolution()). This algorithm is applied if an image is provided in a higher resolution than specified and if image scaling is enabled (see SetSaveNewImageFormat()).

However, the scaling algorithm supports device color spaces as well as CalGray, CalRGB, ICCBased, Separation, and DeviceN (one channel only) color spaces as long as the underlying base or alternate color space is a device space. Images defined in an incompatible color space are inserted unchanged.

Layers (Optional Content)

Layers in PDF are defined with so called Optional Content Groups (OCGs). When we talk about a layer then we mean a piece of content that belongs to an OCG.

Layers provides the ability to dynamically hide or unhide graphical contents. The available API functions to create and manipulate the visibility state of layers with DynaPDF are:

- `AddLayerToDisplTree()`   `// Builds the layer tree for the GUI`
- `AddObjectToLayer()`   `// Adds an object to a layer`
- `AddOCGToAppEvent()`   `// Adds a layer to an application event`
- `DeleteOCGFromAppEvent()`   `// Deletes a layer from an application event`
- `DeleteOCGFromDisplayTree()` `// Deletes a layer from the display tree`
- `DeleteOCUINode()`   `// Deletes an OCUINode from the display tree`
- `BeginLayer()`   `// Opens a layer to add contents to it`
- `EndLayer()`   `// Closes the last open layer`
- `ChangeOCGName()`   `// Changes an OCG or Layer name`
- `CreateOCG()`   `// Optional Content Group (OCG)`
- `CreateOCMD()`   `// Optional Content Membership Dictionary (OCMD)`
- `CreateSetOCGStateAction()` `// Changes the visibility of a layer`
- `GetOCG()`   `// Returns the properties of a layer`
- `GetOCGContUsage()`   `// Returns a Content Usage dictionary (CUD)`
- `GetOCUINode()`   `// Returns the properties of an OCUINode`
- `GetOCGUsageUserName()`   `// Returns a user name of a CUD`
- `GetOCGCount()`   `// Returns the number of OCGs or layer`
- `SetOCGContUsage()`   `// Creates, changes, or deletes a CUD`
- `SetOCGState()`   `// Changes the visibility state of a layer`
- `LockLayer()`   `// Locks a layer in the GUI`
- `UnLockLayer()`   `// Unlocks a layer`

The creation of a layer begins always with the creation of an OCG. If we talk about a "layer" then we mean in fact an OCG. An OCG specifies the layer name, the initial visibility state, and whether the layer should be displayed in the user interface.

Note that the visibility of a layer in the user interface has nothing to do with the visibility of the contents that belongs to it. Invisible layers can be used to avoid interaction with the user interface. This can be useful if the document contains SetOCGState or JavaScript actions, or application events which change the visibility states depending on certain criteria, e.g. when printing the document, or if the user should not be able to change the visibility.

Normal page contents, such as text, images, and vector graphics can be connected with a layer with BeginLayer() / EndLayer(). Anything that is drawn between these function calls becomes part of the layer. BeginLayer() calls can be nested to express dependencies to more than one layer. The visibility of an inner layer depends then also on the visibility of the outer layer(s).

Annotations and Form Fields are **not** automatically added to a layer when created between BeginLayer() / EndLayer() calls. These objects must be added to a layer with AddObjectToLayer() since interactive objects are not part of the page contents.

Annotations and Form Fields can be connected with exactly one OCG or OCMD (Optional Content Membership Dictionary). OCMDs are used if the visibility should depend on more than one layer. An OCMD accepts an array of OCG handles and a visibility expression that specifies when the associated contents should become visibile or invisible. OCMDs can also be used with BeginLayer() to create more complex visibility expressions.

Images and templates can be explicitely connected with an OCG or OCMD too. This can be useful in situations where too many nested BeginLayer() calls would be required to create the same visibility expression or if these object types belong to a specific layer. For example, if a document contains a layer "Images" then it is easier and much more efficient to connect all images directly with that layer in comparison to BeginLayer() / EndLayer() calls.

Viewer applications display layers in a tree structure like bookmarks. Which layers should appear in the tree can be configured with AddLayerToDisplTree() as well as with CreateOCG(). It is not required to add any layer to the layer tree. The visibility state of a layer that is not included in the tree cannot be changed interactively in a viewer application. This can be useful in various situations, e.g. to hide helper layers or simply to avoid changes by the user.

The visibility of layers can be changed dynamically with SetOCGState actions, Javascript, and with application events. SetOCGState actions can be used if a user should be able to change the visibility of a layer. Javascripts are useful if the visibility depends on calculations or other more complex criteria.

The last way to change the visibility state dynamically is the usage of application events. PDF defines events for exporting, printing, and viewing the document. The visibility of certain layers can be changed in each of these events. See AddOCGToAppEvent() for further information.

PDF Transparency

The PDF format supports different types of transparency for images, vector graphics, and text. The simplest type of transparency is called Color Key Masking. This kind of transparency is used to mask areas of a specific color in an image so that they become transparent. Color Key Masking works with a transparent color that must be specified beforehand.

Color Key Masking can produce unpredictable results with JPEG or JPEG2000 compressed images because these filters are lossy filter. The use of a lossy filter means that the output is only an approximation of the original data. Therefore, these filters may lead to slight changes in the color values of image samples, possibly causing samples that were intended to be masked to be unexpectedly painted instead, in colors slightly different from the mask color.

Another technique to mask images is the use of an explicit image mask (see AddMaskImage()). This type of masking does not depend on the source color and produces predictable results with all kinds of compression filters. The PDF format supports 1 bit image masks (PDF 1.3) as well as soft masks (PDF 1.4) which can be defined as 1 bit or 8 bit grayscale image. The latter format represents essentially the alpha channel of an image.

When inserting an image with an alpha channel DynaPDF converts the alpha channel to a soft mask to preserve the transparency effect. The current transparency color or the value of SetUseTransparency() does not affect such images. If an image with an alpha channel should appear opaque then draw a rectangle in the wished background color behind the image.

It is also possible to add a separate image or soft mask to a base image with AddMaskImage(). This is useful if the mask is stored in a separate image or if the mask uses another resolution as the base image.

Alpha Blending

PDF 1.4 introduced a new kind of transparency that based on alpha blending. Alpha blending combines the backdrop and shape colors with an arbitrary opacity factor (alpha) which can be in the range 0.0..1.0. The alpha constant can be defined separately for fillings and strokes. Note that the fill alpha value is considered for images and text if the text draw mode is dmNormal.

The wished alpha values can be defined in an extended graphics state object that must be created with CreateExtGState() and applied with SetExtGState(). An extended graphics state is valid until the modified values are restored with another extended graphics state that sets the values back to their defaults.

Note that the functions Save/RestoreGraphicState() do not affect members of an extended graphics state. This circumstance must be considered when creating transparent objects.

Example (C++):

```cpp
int main(int argc, char* argv[])
{
   pdfCreateNewPDF(PDF, "c:/cppout.pdf");
```

```
    pdfSetPageCoords(PDF, pcTopDown);

    TPDFExtGState gs;
    pdfInitExtGState(&gs);
    gs.FillAlpha = 0.5f;
    SI32 gs1 = pdfCreateExtGState(PDF, &gs);

    gs.FillAlpha = 1.0f;
    SI32 gs2 = pdfCreateExtGState(PDF, &gs);

    double x = 150.0;
    double y = 150.0;

    pdfAppend(PDF);

    pdfSetExtGState(PDF, gs1); // Apply the extended graphics state
    pdfSetFillColor(PDF, PDF_RGB(255, 0, 0));
    pdfDrawCircle(PDF, x, y, 50.0, fmFill);

    pdfSetFillColor(PDF, PDF_RGB(0, 255, 0));
    pdfDrawCircle(PDF, x + 75.0, y, 50.0, fmFill);

    pdfSetFillColor(PDF, PDF_RGB(0, 0, 255));
    pdfDrawCircle(PDF, x + 37.5, y + 50.0, 50.0, fmFill);

    // Restore the fill alpha to fully opaque
    pdfSetExtGState(PDF, gs2);

    x += 200.0;
    pdfSetFillColor(PDF, PDF_RGB(255, 0, 0));
    pdfDrawCircle(PDF, x, y, 50.0, fmFill);

    pdfSetFillColor(PDF, PDF_RGB(0, 255, 0));
    pdfDrawCircle(PDF, x + 75.0, y, 50.0, fmFill);

    pdfSetFillColor(PDF, PDF_RGB(0, 0, 255));
    pdfDrawCircle(PDF, x + 37.5, y + 50.0, 50.0, fmFill);

    pdfEndPage(PDF);
    return pdfCloseFile(PDF);
}
```

Output:
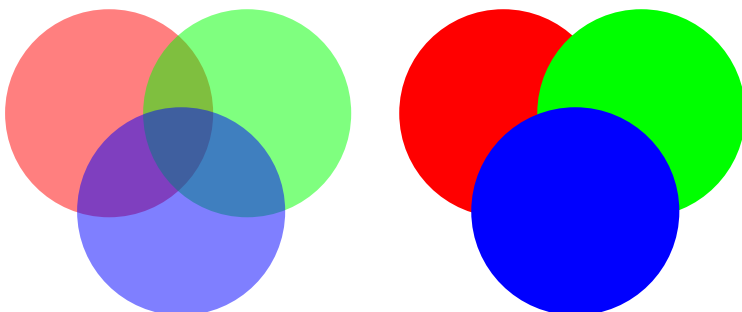
Transparency Groups / Soft Masks

To achieve more complex transparency effects PDF supports transparency groups and soft masks. A transparency group can be created with BeginTransparencyGroup(), it is an extended template that supports the same contents as regular templates, such as vector graphics, text, shadings, or images (see BeginTemplate() for further information).

The difference between a normal template and a transparency group is that a transparency group is composited with the backdrop as hole. This property is very important when working with soft masks.

A soft mask consists in turn of a transparency group and a corresponding dictionary that describes the properties of the mask. The rendered result is then used as an alpha channel to mask arbitrary objects with it. A soft mask can be created with CreateSoftMask().

Once a soft mask was created and activated in the graphics state anything that will be drawn will now will be masked.

However, a soft mask is designed to mask exactly one object at time since the effect on overlapping objects would be as if the mask is applied twice. To enable the usage on arbitrary artwork, the contents can be drawn into a transparency group since a transparency group is composited with the backdrop as a hole. This makes sure that the soft mask causes no side effects when overlapping objects must be masked.

Transparency groups can also be used stand alone, that means outside an active soft mask or other transparency group, but most effects which can be achieved in this way can also be achieved with regular drawing techniques. When a specific effect can be achieved without a transparency group then this way is preferred!

Transparency groups are expensive in terms of memory usage and processing time because a viewer must mostly render the group into an intermediate image buffer so that it can be composited with backdrop as a hole.

Because transparency groups cause already a considerable overhead, a document should be more carefully created in regard to efficiency and rendering speed. Especially the usage of useless clipping paths and JPEG 2000 compressed images should be avoided.

Blend Modes

In addition to alpha blending the PDF standard defines several predefined blend modes (PDF 1.4) to achieve certain special effects. Like the alpha constants the blend mode is part of an extended graphics state object. The usage is identically in comparison to alpha transparency; that means the wished blend mode must be applied with an extended graphics state object and the only way to disable it is to apply another extended graphics state that sets the blend mode back to bmNormal.

Blend functions are applied in a blend color space which is normally the current color space. Valid blend color spaces are DeviceGray, DeviceRGB, DeviceCMYK, CalGray, CalRGB, and ICCBased color spaces which are equivalent to the other ones.

Lab and corresponding ICCBased color spaces are no valid blend color spaces because the blend functions produce no meaningful results with these color spaces.

However, the ICC profile of an ICCBased color space must contain both AToB and BToA transformations. This condition can be problematic because there is no easy way to determine whether this is the case. DynaPDF provides no function to determine whether an ICC profile contains both transformations.

Notice:

Note that Adobe's Acrobat uses always color management to render device colors when a blend mode or alpha transparency is used on a page. The default output color space is always a CMYK color space in PDF! That means, Acrobat or Reader applies a soft-proof on the default CMYK ICC profile if the file contains no Output Intent. Therefore, it is strongly recommended to attach an ICC profile with AddOutputIntent() to achieve predictable results.

Note also that the result of most blend modes depend strongly on the used color space. Due to the way how blend modes compute color values the results for additive and subtractive color spaces are very different.

Example (C++):

```cpp
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode,
const char* ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}

void AddCircles(const void* PDF, SI32 Templ, TPDFExtGState &ExtGS,
SI32 DefGS, TBlendMode Mode, double &x, double &y)
{
   ExtGS.BlendMode = Mode;

   pdfSetExtGState(PDF, DefGS);
   pdfSetFillColor(PDF, PDF_BLACK);
   x += 30.0;
   y -= 20.0;
```

```c
   switch(Mode)
   {
      case bmNormal:     pdfWriteText(PDF, x, y, "  Normal");    break;
      case bmColor:      pdfWriteText(PDF, x, y, "  Color");     break;
      case bmColorBurn:  pdfWriteText(PDF, x, y, "ColorBurn");   break;
      case bmColorDodge: pdfWriteText(PDF, x, y, "ColorDodge");  break;
      case bmDarken:     pdfWriteText(PDF, x, y, "  Darken");    break;
      case bmDifference: pdfWriteText(PDF, x, y, "Difference");  break;
      case bmExclusion:  pdfWriteText(PDF, x, y, "Exclusion");   break;
      case bmHardLight:  pdfWriteText(PDF, x, y, "HardLight");   break;
      case bmHue:        pdfWriteText(PDF, x, y, "     Hue");    break;
      case bmLighten:    pdfWriteText(PDF, x, y, " Lighten");    break;
      case bmLuminosity: pdfWriteText(PDF, x, y, "Luminosity");  break;
      case bmMultiply:   pdfWriteText(PDF, x, y, " Multiply");   break;
      case bmOverlay:    pdfWriteText(PDF, x, y, " Overlay");    break;
      case bmSaturation: pdfWriteText(PDF, x, y, "Saturation");  break;
      case bmScreen:     pdfWriteText(PDF, x, y, " Screen");     break;
      case bmSoftLight:  pdfWriteText(PDF, x, y, "SoftLight");   break;
   }

   x -= 30.0;
   y += 20.0;

   SI32 gs = pdfCreateExtGState(PDF, &ExtGS);
   pdfSetExtGState(PDF, gs);

   pdfPlaceTemplate(PDF, Templ, x, y, 0.0, 0.0);

   x += 120.0f;
   if (x > 500.0)
   {
      x  = 70.0;
      y += 120.0;
   }
}

void TestBlendModes(void)
{
   void* pdf = pdfNewPDF();
   if (!pdf) return -1; // Out of memory
   pdfSetOnErrorProc(pdf, NULL, PDFError);

   pdfCreateNewPDF(pdf, "test.pdf");
   pdfSetPageCoords(pdf, pcTopDown);
   pdfSetColorSpace(pdf, csDeviceRGB);

   TPDFExtGState gs, def;
   pdfInitExtGState(&gs);
   pdfInitExtGState(&def);
```

```
    def.BlendMode = bmNormal;

    SI32 defGS = pdfCreateExtGState(pdf, &def);

    double x = 70.0;
    double y = 110.0;

    SI32 tmpl = pdfBeginTemplate(pdf, 105.0, 90.0);

    pdfSetFillColor(pdf, pdf_RGB(255, 0, 0));
    pdfDrawCircle(pdf, 30.0, 30.0, 30.0, fmFill);
    pdfSetFillColor(pdf, pdf_RGB(0, 255, 0));
    pdfDrawCircle(pdf, 75.0, 30.0, 30.0, fmFill);
    pdfSetFillColor(pdf, pdf_RGB(0, 0, 255));
    pdfDrawCircle(pdf, 52.5, 60.0, 30.0, fmFill);

    pdfEndTemplate(pdf);

    pdfAppend(pdf);

    pdfSetFont(PDF, "Helvetica", fsBold, 20.0, false, cp1252);
    pdfWriteFTextEx(PDF, 50.0, 50.0, pdfGetPageWidth(PDF)-100.0, -1.0,
                    taCenter, "Blend Modes");

    pdfSetFont(PDF, "Helvetica", fsRegular, 10.0, false, cp1252);
    AddCircles(pdf, tmpl, gs, defGS, bmNormal, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmColorBurn, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmColorDodge, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmDarken, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmDifference, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmExclusion, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmHardLight, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmHue, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmLighten, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmLuminosity, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmMultiply, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmOverlay, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmSaturation, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmScreen, x, y);
    AddCircles(pdf, tmpl, gs, defGS, bmSoftLight, x, y);
    pdfEndPage(pdf);

    pdfCloseFile(pdf);
    pdfDeletePDF(pdf);
}
```

Blend Modes

Normal	Color	ColorBurn	ColorDodge

Darken	Difference	Exclusion	HardLight

Hue	Lighten	Luminosity	Multiply

Overlay	Saturation	Screen	SoftLight
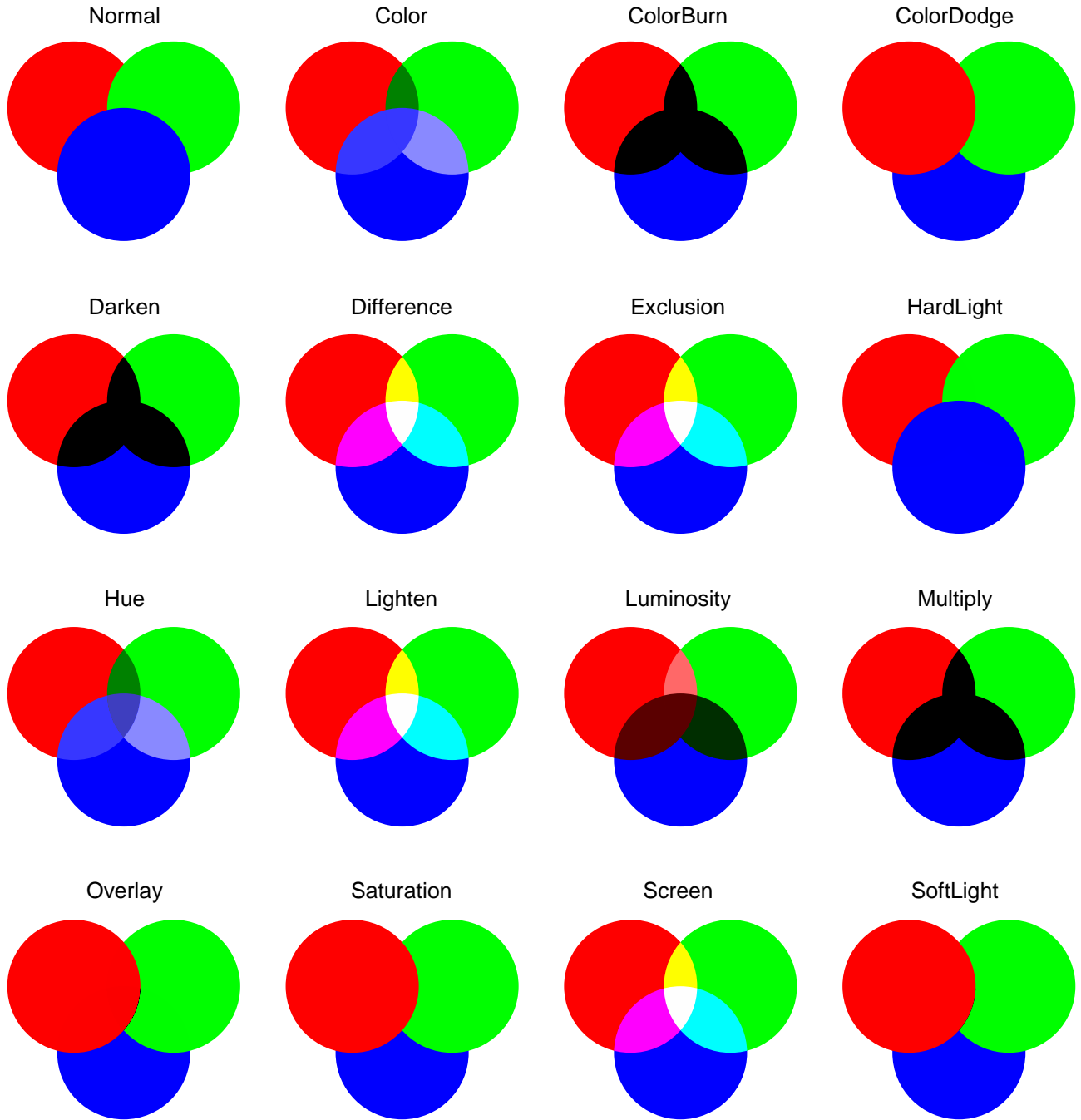
Tables

DynaPDF provides a very powerful table class that can be used to split an arbitrary area into rows and columns. Every table cell can contain back- and foreground objects, as well as text, images, templates, and other tables.

General properties

A table can be used on an arbitrary number of pages and in an arbitrary number of PDF files.

A table is an independent object that is not destroyed with the PDF file in memory. A table can be used arbitrary often and the contents in it can be updated with new values whenever necessary.

Once a table was fully defined it can be drawn on one or on multiple pages. The output position can be specified when the table is drawn. A table can also span multiple pages.

For tables which span maybe multiple pages it is possible to define header rows. Header rows will be repeated when a page break occurs.

The table, rows, columns, and cells support many identical properties. The properties of a cell are inherited from the table, column, and row, in this order. The formatting rules are comparable with HTML but they are not identically.

A major difference is that the height of the table cannot be specified since this would complicate the handling a lot. The height of a table depends only on the heights of the rows in it.

Foreground objects expand a cell if necessary while background objects are drawn into the available cell space.

An empty table has no appearance. At least one row must be added to achieve a visible appearance.

Error Handling

A table stores error messages and warnings in the error log of the associated PDF instance. This is the case when rows and columns will be created or when contents will be added to cells. When the table is drawn, then the normal error handling of the PDF instance is used. That means, error messages and warnings are passed to the error callback function if set.

If all error messages should be redirected to the error log then set the error mode to emUseErrLog. See SetErrorMode() for further information.

Non-fatal errors do not break processing. If contents cannot be drawn, e.g. when an image is not available, then the cell stays empty and the next cell will drawn.

Borders, Cell Spacing, Cell Padding

It is important to understand how borders are drawn, and how cell spacing and cell padding is applied. A table supports three different kinds of borders: the table border, cell borders, and a special kind of border: the cell grid.

Table and cell borders appear fully inside the table or cell boundary. The cell grid is no real border; it consists of horizontal and vertical lines which are drawn in the middle of two rows or columns. The grid lines reduce the width of columns but increase the height of rows.
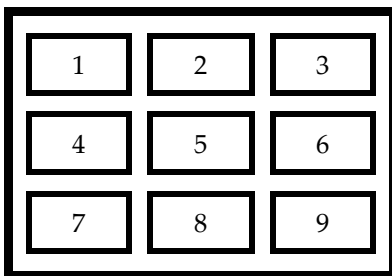
The width and colors of the grid lines can be set for the entire table and for individual rows and columns but not for cells. A grid line covers always the entire width or height of the table.

Cell spacing controls the space between cells (outer margin) and cell padding controls the space between the cell contents and the cell boundary including borders (inner margin).

Cell spacing and cell padding can be defined for the entire table, as well as for individual columns, rows, and cells. The properties are inherited from the table, columns, rows, and cells, in this order. If a value is defined for the table, for a column and for a row, for example, then the value of the row takes precedence.

Cell spacing and cell padding decrease the widths of columns but increase the height of rows. If the available width for the contents becomes too small then the column will be expanded to the minimum width: that is the sum of the border widths + cell spacing + cell padding + minimum width of the cell contents. This adjustment is done for all columns if necessary. If the sum of the column widths becomes larger as the table width then the table width is adjusted too. DrawTable() outputs a warning if the width of one or more columns was changed.

```
Table border = 3 Units, Cell Border  = 2 Units
Cell Spacing = 6 Units, Cell Padding = 3 Units
```


Background Objects

A background color can be assigned to the table, columns, rows, and cells. The table has its own background color while the background color of a cell is inherited from the column and row, in this order.

The table and cells support also background images and templates. Background objects are not inherited. The table and cells have its own background.

If a background color and an additional background object is defined then the table or cell is filled with the background color before other objects are drawn.

The background area is the area inside the cell or table border. In contrast, the content area of foreground objects is the area inside the border minus cell padding. Background objects do never change the width or height of a cell.

Foreground Objects

Foreground objects have a strong width and height; that means if the cell is not large enough then it will be expanded. Text and sub tables are always foreground objects.

Text changes the width of a column, if the column width is smaller as the font size, if the flag tfNoLineBreak is set and if the cell uses a portrait orientation, or if the cell uses a landscape orientation and if the flag tfNoLineBreak is absent. In all other case the height of the row will be adjusted if necessary.

Cell Alignment and Orientation

The contents in a cell can be horizontally and vertically aligned as well as rotated in 90 degrees steps. If a cell is rotated by 90 or 270 degrees then text flows from left to right or from right to left respectively. If the flag tfNoLineBreak is not set then the height of the cell or row must be set to a value greater zero. If no height was set, then the font size is used as default height. This value is usually too small and causes unpredictable results.

ColSpan, RowSpan

ColSpan or RowSpan are not directly supported but every cell can contain a sub table if necessary. This makes it possible to create almost every combination of spanned rows or columns.

If you need these properties, then create a master table with only one column and insert new tables with the required number of rows and columns into the cells of the master table.

Page breaks

The last parameter of DrawTable() specifies the maximum height of the table. If the maximum height is set to a value greater zero, then the function must be executed in a while statement as follows:

```
...
pdfAppend(pdf);
tblDrawTable(tbl, 50.0f, 50.0f, 742.0f);
while (tblHaveMore(tbl))
{
   pdfEndPage(pdf);
   pdfAppend(pdf);
   // The position and maximum height can be changed if necessary
   tblDrawTable(tbl, 50.0f, 50.0f, 742.0f);
}
pdfEndPage(pdf);
...
```

It is not required to close the page directly after DrawTable() returns. You can draw additional contents if necessary before closing the page and you can also draw additional contents before and after drawing the remaining rows.

The current implementation does not allow page breaks inside of cells. The contents in a cell is always fully drawn, also if the height of a single row exceeds the maximum output height. The function makes sure that at least one row is always drawn to avoid an endless loop.

Table and cell properties

When a table is first created, its properties are initialized to the following values:

```
BackgroundColor     = None (Transparent)
BackgroundColorSpace = None
Background          = None
BorderColor         = 0 (Black)
BorderColorSpace    = esDeviceRGB
BorderWidth         = 0 (Left, Right, Top, Bottom)
CellOrientation     = 0
CellSpacing         = 0 (Left, Right, Top, Bottom)
CellPadding         = 0 (Left, Right, Top, Bottom)
GridWidth           = 0 (horizontal and vertical)
GridHorzColor       = 0 (Black)
GridVertColor       = 0 (Black)
Font                = Helvetica
FontStyle           = fsRegular
FontSize            = 10.0
FontCodepage        = cp1252
FontEmbedding       = false
ImageColor          = 0 (Black) -> 1 bit images
ImageColorSpace     = esDeviceRGB
HAlign              = coLeft
VAlign              = coTop
TextColor           = 0 (Black)
TextColorSpace      = esDeviceRGB
```

Table color spaces

The function SetColor() sets a color and a corresponding device color space for a specific object type. It is also possible to use extended color spaces like ICCBased, Separation or DeviceN color spaces. Extended color spaces must be created in the PDF instance that is associated with the table. After the color space was created it can be used with SetColorEx().

The table stores just handles of the color space so that they can be activated when the table is drawn.

Extended color spaces will be deleted when the PDF file in memory is closed or when FreePDF() will be called. Therefore, it is not possible to draw a table into different PDF files if it uses extended color spaces and if the required color spaces will not be recreated. Since color space handles may change when recreated in another PDF file, all colors which use extended color spaces should be set again to avoid issues with invalid color space handles.

The default color space for all object types is DeviceRGB. Images should normally be inserted in the native image color space. This can be achieved by setting the flag tfUseImageCS. The flag is inherited from the table, columns, and rows as usual. See SetFlags() for further information.

The color space for images can be set with SetColor() and SetColorEx(). The color value is only used for 1 bit images. If an image color is defined and if the image is a 1 bit image then it will be output as an image mask. In this case, non-zero pixel values are drawn in the defined image color and zero pixel values produce no output.

If no image color is defined then 1 bit images are drawn as opaque black & white image.

Table Creation

A table can be created with the function CreateTable() in C or C++. All other interfaces contain the table class CPDFTable or TPDFTable in Delphi which encapsulates the table functions into a native class object. The class name is dynatable in PHP.

The constructor takes the number of rows which should be allocated, as well as the number of columns and the default row height.

> Since VB 6 does not support parameterized constructors, the function CreateTable() must be called after creating the class instance.

The number of pre-allocated rows should be large enough to avoid unnecessary memory re-allocations. Note that this is just the size of the array that holds the rows which will be added later. An unused row requires 4 or 8 bytes memory (32 or 64 bit).

The constructor does not allocate rows. The first chunk will be allocated when the first row is added.

C/C++ deveopers must delete a table object when no longer needed with DeleteTable(). The destructor of the class CPDFTable calls this function automatically in other programming languages when the class instance will be deleted.

In programming languages which use reference counting like PHP or VB 6, it is very important to delete tables and the associated PDF instance in the right order, especially if a table contains references of other tables. This is the case when a sub table was inserted into a cell.

Delete first all tables which contain references of other tables and delete finally the sub tables. This makes sure that the reference count reaches zero when trying to release the class instance.

Tables must also be deleted before the associated PDF instance will be deleted since a table contains a reference of this PDF instance! These rules must be carefully considered to avoid memory leaks.

Although it is not required to delete tables in the right order in C or C++ or Delphi, it is best practice to do so.

Table Functions

AddColumn

Syntax:

```
SI32 tblAddColumn(
   const ITBL* Table, // Table pointer
   LBOOL Left,        // Add the column on the left or right side?
   float Width)       // Column width
```

The function adds a column on the left or right side of the table. In most cases it is required to change the table width and the widths of the remaining columns after a new column was added. This can be done with SetTableWidth() and SetColWidth().

If possible, create the table directly with the required number of columns.

Return values:

If the function succeeds, the return value is new number of columns. If the function fails, the return value is a negative error code.

AddRow

Syntax:

```
SI32 tblAddRow(
   const ITBL* Table, // Table pointer
   float Height)      // Minimum height or -1 for default height
```

The functions adds a new row to the table. The parameter *Height* can be set to a negative value to indicate that the default row height should be used. The default row height was specified in CreateTable().

The specified height represents the minumum height of the row. If the height of a foreground object is larger then the height will be adjusted.

The height is sometimes required. This is the case when a cell is rotated by 90 or a multiple of 90 degrees and if the height cannot be computed from the cell contents, e.g. if a cell contains text and if the flag tfNoLineBreak is absent. See SetCellOrientation() for further information.

Return values:

If the function succeeds, the return value is the row index, a value greater or equal zero. If the function fails, the return value is a negative error code.

AddRows

Syntax:

```
SI32 tblAddRows(
   const ITBL* Table, // Table pointer
   UI32 Count,        // Number of rows to be added
   float Height)      // Minumum height or -1 for default height
```

The functions adds new rows to the table. The parameter *Height* can be set to a negative value to indicate that the default row height should be used. The default row height was specified in CreateTable().

The specified height represents the minumum height of each row. If the height of a foreground object is larger, then the row height will be adjusted.

The height is sometimes required. This is the case if a cell is rotated by 90 or a multiple of 90 degrees and if the height cannot be computed from the cell contents, e.g. if a cell contains text and if the flag tfNoLineBreak is absent. See SetCellOrientation() for further information.

Return values:

If the function succeeds, the return value is new number of rows in the table. If the function fails, the return value is a negative error code.

ClearColumn

```
void tblClearColumn(
   const ITBL* Table,     // Table pointer
   UI32 Col,              // Column index
   TDeleteContent Types)  // See ClearContent()
```

The function deletes the content in the specified column in all rows. The parameter *Types* is described in ClearContent() below.

ClearContent

Syntax:

```
void tblClearContent(
   const ITBL* Table,     // Table pointer
   TDeleteContent Types)  // See below
typedef enum
{
   dcText       = 0x00000001, // Text is always a foreground object
   dcImage      = 0x00000002, // Images
   dcTemplate   = 0x00000004, // Templates
   dcTable      = 0x00000008, // Sub tables are always foreground objects
   dcAllCont    = 0x0000001F, // Delete all content types
   // Required additional flag
   dcForeGround = 0x10000000, // Only foreground objects?
   dcBackGround = 0x20000000, // Only background objects?
```

```
   dcBoth        = 0x30000000  // Delete both fore- and background objects
}TDeleteContent;
```

The function deletes the specified object types from the table. The parameter *Types* is a bit mask. The content types and the fore- and background flags can be combined with a binary or operator. If the foreground and background flags are both absent, then nothing will be deleted!

In C/C++ the usage is as follows:

```
tblClearContent(tbl, TDeleteContent(dcText | dcForeGround));
```

Other programming languages require no type cast.

Note that it is not required to explicitly clear the contents of a table. Existing contents can be overriden at any time. However, when it is not known whether all cells get new values, then it is recommended to clear the table beforehand.

It is also possible to clear the contents of specific rows or columns only. See ClearColumn() and ClearRow() for further information.

ClearRow

```
void tblClearRow(
   const ITBL* Table,     // Table pointer
   UI32 Row,              // Row index
   TDeleteContent Types) // See ClearContent()
```

The function deletes the content in the specified row. The parameter Types is described in ClearContent().

CreateTable

Syntax:

```
ITBL* tblCreateTable(
   const PPDF* IPDF,    // PDF instance
   UI32 AllocRows,      // Must be greater zero
   UI32 NumCols,        // Must be greater zero
   float Width,         // Table width
   float DefRowHeight) // Default row height
```

The function creates a new table object. The parameter *AllocRows* specifies the number of rows which should be pre-allocated. The value should be large enough to avoid unnecessary memory re-allocations. Note that this is just the size of the array that holds the rows which will be added later. Unused rows require 4 or 8 bytes memory (32 bit / 64 bit).

The default row height is used if the parameter *Height* of AddRow() or AddRows() is set to a negative value.

The widths of the columns is set to the table width divided by the number of columns (Width / NumCols).

The table must be deleted with DeleteTable() when finish to avoid a memory leak.

Return values:

If the function succeeds the return value is a valid table pointer. If the function fails the return value is NULL.

DeleteCol

Syntax:

```
void tblDeleteCol(
   const ITBL* Table, // Table pointer
   UI32 Col)          // Column index
```

The function deletes a column from the table. The new table width is the table width minus the width of the deleted column. It is not possible to delete all columns. One column must be left in the table.

If the column index is invalid or if the table contains only one column then the function does nothing.

DeleteRow

Syntax:

```
void tblDeleteRow(
   const ITBL* Table, // Table pointer
   UI32 Row)          // Row index
```

The function deletes a row from the table. If the row index is invalid then the function does nothing.

DeleteRows

Syntax:

```
void tblDeleteRows(
   const ITBL* Table) // Table pointer
```

The function deletes all rows from the table.

DeleteTable

Syntax:

```
void tblDeleteTable(
   ITBL** Table) // Table pointer
```

The function deletes a table object. Tables and the corresponding PDF instance should be deleted in the reversed order of the creation. That means, delete first all tables and then the PDF instance if no longer needed. This makes sure that the table can be immediatly deleted in programming languages which use reference counting or garbage collection.

This function is not included in programming languages which use a wrapper class like C#, Delphi, VB. Net. or PHP. Delete the class instance as usual instead. The destructor deletes the table object in this case.

DrawTable

Syntax:

```
float tblDrawTable(
   const ITBL* Table, // Table pointer
   float x,           // x-coordinate
   float y,           // y-coordinate
   float MaxHeight)   // Maximum height or zero to avoid page breaks
```

The function draws the table on the specified position. The x/y-coordinates specify the top left corner of the table. The table flows always from top to bottom.

The function must be executed in a while statement if the parameter *MaxHeight* is set to a value greater zero:

```
...
pdfAppend(pdf);
tblDrawTable(tbl, 50.0f, 50.0f, 742.0f);
while (tblHaveMore(tbl))
{
   pdfEndPage(pdf);
   pdfAppend(pdf);
   // The position and maximum height can be changed if necessary
   tblDrawTable(tbl, 50.0f, 50.0f, 742.0f);
}
pdfEndPage(pdf);
...
```

The function draws the header rows if any and at least one row to avoid an endless loop if the maximum height would be smaller as the first row height.

Return values:

If the function succeeds the return value is the height of the table that was drawn so that additional contents can be drawn below the table if necessary. If the function fails the return value is -1.

GetFirstRow

Syntax:

```
SI32 tblGetFirstRow(
   const ITBL* Table) // Table pointer
```

The function returns the index of the first row that was drawn in the previous DrawTable() call. GetNextRow() returns the next row index that will be drawn in the next DrawTable() call. If the next row index equals the number of rows then the table was fully drawn.

Return value:

The first row index, a number greater or equal zero. This function cannot fail.

GetFlags

```
float tblGetFlags(
    const ITBL* Table, // Table pointer
    SI32 Row,          // Row index or -1
    SI32 Col)          // Column index or -1
```

The function returns the current flags of the table, row, column, or cell:

Row	Col	Applies to
-1	-1	Table
-1	>= 0	Column
>= 0	-1	Row
>= 0	>= 0	Cell

Return values:

If the function succeeds the return value is greater or equal zero. If the function fails, the return value is a negative error code.

GetNextHeight

Syntax:

```
float tblGetNextHeight(
    const ITBL* Table, // Table pointer
    float MaxHeight,   // Maximum height or zero to avoid page breaks
    SI32* NextRow)     // Can be set to NULL if not needed
```

The function calculates the height of the table. If the parameter *MaxHeight* is zero, then the full height will be returned. If *MaxHeight* is greater zero, then the height of the table is calculated that fits into the output height. This is the same height that DrawTable() would produce when drawing the table with the same parameters. This function can be useful if the height of the table must be known before it can be drawn. The function caches the result so that multiple calls with a different max height can be computed fast as possible.

The function must be called within an open page.

Return values:

If the function succeeds the return value is the table height if drawn with the same parameters. If the function fails the return value is -1.

GetNextRow

Syntax:

```
SI32 tblGetNextRow(
   const ITBL* Table) // Table pointer
```

The function returns the index of the next row that will be drawn when DrawTable() is called the next time. GetFirstRow() returns the first row index that was drawn in the previous DrawTable() call. If the next row index equals the number of rows then the table was fully drawn.

Return value:

The next row index, a number greater or equal zero. This function cannot fail.

GetNumCols

Syntax:

```
SI32 tblGetNumCols(
   const ITBL* Table) // Table pointer
```

The function returns the number of columns in the table.

GetNumRows

Syntax:

```
SI32 tblGetNumRows(
   const ITBL* Table) // Table pointer
```

The function returns the number of rows in the table.

GetPDFInstance

Syntax:

```
void* tblGetPDFInstance(
   const ITBL* Table) // Table pointer
```

The function returns the PDF instance that is associated with the table.

GetTableHeight

Syntax:

```
float tblGetTableHeight(
   const ITBL* Table) // Table pointer
```

The function returns the full height of the table. The function must be called within an open page.

Return values:

If the function succeeds the return value is the height of the table. If the function fails the return value is -1.

GetTableWidth

Syntax:

```
float tblGetTableWidth(
    const ITBL* Table) // Table pointer
```

The function returns the width of the table. The function must be called within an open page.

Return values:

If the function succeeds the return value is the width of the table. If the function fails the return value is -1.

HaveMore

Syntax:

```
LBOOL tblHaveMore(
    const ITBL* Table) // Table pointer
```

The function checks whether all rows where drawn. The return value is always false if the *MaxHeight* parameter of DrawTable() was set to zero.

SetBoxProperty

Syntax:

```
LBOOL tblSetBoxProperty(
    const ITBL* Table,        // Table pointer
    SI32 Row,                 // Row index or -1
    SI32 Col,                 // Column index or -1
    TTableBoxProperty Type,   // see below
    float Left,               // Value for the left side
    float Right,              // Value for the right side
    float Top,                // Value for the top side
    float Bottom)             // Value for the bottom side

typedef enum
{
    tbpBorderWidth = 0, // Table, Columns, Rows, Cells default (0, 0, 0, 0)
    tbpCellSpacing = 1, // Table, Columns, Rows, Cells default (0, 0, 0, 0)
    tbpCellPadding = 2  // Table, Columns, Rows, Cells default (0, 0, 0, 0)
}TTableBoxProperty;
```

The function sets or changes the border width, cell spacing, or cell padding. The properties can be set to the table, rows, columns, and cells as follows:

Row	Col	Applies to
-1	-1	Table
-1	>= 0	Column
>= 0	-1	Row
>= 0	>= 0	Cell

Cell padding and cell spacing are inherited from the table, columns, and rows, in this order.

The border with is inherited from columns and rows (in this order), but not from the table since the table has its own border.

Individual values can be set to each side of a cell. Note that the properties will be not be rotated with a cell. The left side is always left, independent of the cell orientation.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCellAction

Syntax:

```
LBOOL tblSetCellAction(
   const ITBL* Table,    // Table pointer
   SI32 Row,             // Row index or -1
   SI32 Col,             // Column index or -1
   UI32 ActionHandle,    // Action handle
   THighlightMode Mode)  // Highlight mode
```

The function creates an invisible link annotation for the specified cell or table and associates the provided action with that annotation. The result is an interactive area that executes the action if the user clicks on the cell or table.

A cell action is always a background object that is created in the size of the background area when the table is drawn.

Notice:

Unlike other cell objects like images, text, and so on, a cell link is no persistent object because the action will be deleted when then PDF file in memory will be closed or when FreePDF() will be called.

If the table should be drawn into more than one PDF file then re-create the actions and cell action before drawing the table into the next PDF file.

Remarks:

Actions can be created with functions like CreateGoToAction(), CreateGoToRAction(), and so on.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCellDashPattern

Syntax:

```
LBOOL tblSetCellDashPattern(
   const ITBL* Table,    // Table pointer
   SI32 Row,             // Row index or -1
   SI32 Col,             // Column index or -1
   TCellSide Side,       // Cell side on which the style should be applied
```

```
TLineCapStyle Style, // Line cap style
const float* Dash,    // Dash pattern
UI32 NumValues,       // Number of values in Dash
float Phase)          // Dash phase
```

The function sets or changes the line dash pattern of a cell, row, or of the table. See SetLineDashPattern2() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCellImage

Syntax:

```
LBOOL tblSetCellImage(
    const ITBL* Table, // Table pointer
    SI32 Row,            // Row index or -1
    SI32 Col,            // Column index or -1
    LBOOL ForeGround,    // If true, the cell will be expanded if necessary
    TCellAlign HAlign,   // Horizontal alignment
    TCellAlign VAlign,   // Vertical alignment
    float Width,         // Must be greater or equal zero
    float Height,        // Must be greater or equal zero
    const char* Image,   // This should be an absolute path
    UI32 Index)          // Image index (TIFF images only)
```

The function inserts an image horizontally and vertically aligned as specified. The image will be loaded when the table is drawn. Therefore, it is recommended to use absolute paths so that the image can still be loaded if the working directory was changed.

If the image is inserted as a background object, then it will be clipped if it does not fit into the cell boundary. A cell can contain a background and a foreground object.

An image can also be added to the table as background object.

Foreground objects have a strong width and height. That means the cell will be expanded if necessary.

Image formats which support no transparency are inserted opaque. Transparent GIF images are inserted transparent and alpha channels in images will be considered. Images should normally be inserted in the native image color space. To achieve this, set the flag tfUseImageCS with SetFlags().

Images can be scaled into the cell as follows:

- *Width* = 0 and *Height* = 0 -> The image is scaled to the cell width or height depending on the cell orientation. The aspect ratio will be preserved. If the flag tfScaleToRect is set, the image is scaled so that both sides fit into the cell width and height. The row height must be greater zero to achieve a meaningful result.

- *Width > 0 and Height = 0 **or** Width = 0 and Height > 0* -> The image is scaled to the given width or height. The aspect ratio will be preserved. If the flag tfScaleToRect is set, the image is scaled so that both sides fit into the given width or height and into cell width or height depending on which side is missing. If the parameter *Height* and the row height are zero then the flag tfScaleToRect has no effect.

- *Width > 0 and Height > 0* -> The image is scaled to the given width and height. The aspect ratio is **not** preserved. If the flag tfScaleToRect is set, the image is scaled so that both sides fit into the given width and height and the aspect ratio is preserved.

The parameters *Width* and *Height* are not exchanged if the cell uses a landscape orientation.

Remarks:

The function is available in an Ansi and Unicode compatible variant. Unicode paths are converted to UTF-8 on non-Windows operating systems. Ansi strings are interpreted as UTF-8 if the flag gfAnsiStringIsUTF8 is set the in the associated PDF instance. See SetGStateFlags() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCellImageEx

Syntax:

```
LBOOL tblSetCellImageEx(
   const ITBL* Table,  // Table pointer
   SI32 Row,           // Row index or -1
   SI32 Col,           // Column index or -1
   LBOOL ForeGround,   // If true, the cell will be expanded if necessary
   TCellAlign HAlign,  // Horizontal alignment
   TCellAlign VAlign,  // Vertical alignment
   float Width,        // Must be greater or equal zero
   float Height,       // Must be greater or equal zero
   const void* Buffer, // Image buffer
   UI32 BufSize,       // Buffer size
   UI32 Index)         // Image index (TIFF images only)
```

The function inserts an image in the same way as SetCellImage() but accepts a file buffer instead of a file path. See SetCellImage() for further information. The function creates a copy of the image buffer.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCellOrientation

Syntax:

```
LBOOL tblSetCellOrientation(
    const ITBL* Table, // Table pointer
    SI32 Row,          // Row index or -1
    SI32 Col,          // Column index or -1
    SI32 Orientation)  // 0, 90 or a multiple of 90 degrees
```

The function sets the cell orientation. The cell orientation is inherited from the table, column, row, and finally from the cell. The contents in a cell can be rotated in 90 degrees steps. Positive values rotate the cell content counter clockwise and negative values clockwise.

The row height should be set to value greater zero if a cell uses a landscape orientation. This is important if the height cannot be computed from the cell contents. This is the case if a cell contains text and if the flag tfNoLineBreak was not set.

The orientation can be set to the table, rows, columns, and cells as follows:

Row	Col	Applies to
-1	-1	Table
-1	>= 0	Column
>= 0	-1	Row
>= 0	>= 0	Cell

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCellTable

Syntax:

```
LBOOL tblSetCellTable(
    const ITBL* Table,     // Table pointer
    UI32 Row,              // Row index
    UI32 Col,              // Column index
    TCellAlign HAlign,     // Horizontal alignment
    TCellAlign VAlign,     // Vertical alignment
    const ITBL* SubTable)  // Pointer of the sub table
```

The function inserts a sub table into the specfied cell. A sub table is always a foreground object that has a strong width and height. That means, if the cell is not large enough it will be expanded.

Note that the function creates no copy of the table. Do not delete the sub table when it is used by another table. The C#, VB. Net, VB, and PHP interfaces make sure that a sub table cannot be deleted before the tables which contain references of it will be deleted.

In programming languages which use reference counting like VB or PHP, it is important to delete tables in the right order. Delete first the tables which contain references to sub tables. Finally, delete the sub tables.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCellTemplate

Syntax:

```
LBOOL tblSetCellTemplate(
    const ITBL* Table,   // Table pointer
    SI32 Row,            // Row index or -1
    SI32 Col,            // Column index or -1
    LBOOL ForeGround,    // If true, the cell will be expanded if necessary
    TCellAlign HAlign,   // Horizontal alignment
    TCellAlign VAlign,   // Vertical alignment
    UI32 TmplHandle,     // Template handle
    float Width,         // Must be greater or equal zero
    float Height)        // Must be greater or equal zero
```

The function inserts a template into the specified cell. A template can contain contents from external sources like PDF or EMF pages as well as anything that can be drawn on a page (with exception of interactive objects like form fields or annotations).

Please note that the table creates no copy of the template. Template handles are bound on the PDF instance in which they were created and all templates will be deleted when the PDF file in memory is closed or when FreePDF() will be called.

Therefore, it is **not** possible to draw a table with templates into different PDF files.

If the template is inserted as a background object, then it will be clipped if it does not fit into the cell boundary. A cell can contain a background and a foreground object.

A template can also be added to the table as background object.

Foreground objects have a strong width and height. That means the cell will be expanded if necessary.

Templates can be scaled into the cell as follows:

- *Width* = 0 and *Height* = 0 -> The template is scaled to the cell width or height depending on the cell orientation. The aspect ratio will be preserved. If the flag tfScaleToRect is set, the template is scaled so that both sides fit into the cell width and height. The row height must be greater zero to achieve a meaningful result.

- *Width > 0 and Height = 0* **or** *Width = 0 and Height > 0* -> The template is scaled to the given width or height. The aspect ratio will be preserved. If the flag tfScaleToRect is set, the template is scaled so that both sides fit into the given width or height and into cell width or height depending on which side is missing. If the parameter *Height* and the row height are zero then the flag tfScaleToRect has no effect.

- *Width > 0 and Height > 0 ->* The template is scaled to the given width and height. The aspect ratio is **not** preserved. If the flag tfScaleToRect is set, the template is scaled so that both sides fit into the given width and height and the aspect ratio is preserved.

The parameters *Width* and *Height* are not exchanged if the cell uses a landscape orientation.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCellText

Syntax:

```
LBOOL tblSetCellText(
   const ITBL* Table, // Table pointer
   UI32 Row,          // Row index
   UI32 Col,          // Column index
   TTextAlign HAlign, // Horizontal alignment
   TCellAlign VAlign, // Vertical alignment
   const char* Text,  // The text to be drawn
   UI32 Len)          // Length in characters
```

The function inserts the specified text into the cell. Text is always a foreground object. The cell will be expanded if necessary.

Text can be processed in different ways. By default the text is output with WriteFText(). WriteFText() supports a lot of format tags. These tags are also supported in a table with exception of a page break tag.

To avoid line breaks it is possible to set the flag tfNoLineBreak with SetFlags(). The flag is inherited from the table, column, row, and cell, in this order. If set, the text is output with WriteText() instead. Since this function doesn't support format tags, the text is processed as plain text.

The font and font size can be set with SetFont() and with SetFontSize(). Note that a table has its own versions of these functions.

If the cell uses a landscape orientation then the text flows to left or to right depending on the orientation and if the flag tfNoLineBreak is not set. The row height must be set to value greater zero in this case. If the text does not fit into the cell then the column width will be expanded. If the flag tfNoLineBreak is set, then the height of the row will be expanded if necessary.

If the cell uses a portrait orientation and if the flag tfNoLineBreak is not set, then text flows from top to bottom or bottom to top, depending on the orientation. The row height will be expanded if necessary. If the flag tfNoLineBreak is set, then the column width will be expanded if necessary.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Ansi strings are interpreted as UTF-8 if the flag gfAnsiStringIsUTF8 is set the in the associated PDF instance. See SetGStateFlags() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetColor

Syntax:

```
LBOOL tblSetColor(
   const ITBL* Table, // Table pointer
   SI32 Row,          // Row index or -1
   SI32 Col,          // Column index or -1
   TTableColor Type,  // see below
   TPDFColorSpace CS, // Color space in which the color is defined
   UI32 Color)        // Color value

typedef enum
{
   tcBackColor     = 0, // Table, Columns, Rows, Cells -> default none
   tcBorderColor   = 1, // Table, Columns, Rows, Cells -> default black
   tcGridHorzColor = 2, // Table                       -> default black
   tcGridVertColor = 3, // Table                       -> default black
   tcImageColor    = 4, // Table, Columns, Rows, Cells -> default black
   tcTextColor     = 5  // Table, Columns, Rows, Cells -> default black
}TTableColor;
```

The function sets or changes the specified color as well as the corresponding device color space. Colors of an extended color space can be set with SetColorEx().

The text color is inherited from the table, column, and row, in this order. The border color is inherited from columns and rows but not from the table since the table has its own border.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetColorEx

Syntax:

```
LBOOL tblSetColorEx(
   const ITBL* Table, // Table pointer
   SI32 Row,          // Row index or -1
   SI32 Col,          // Column index or -1
   TTableColor Type,  // see below
   float* Color,      // One value per component
   UI32 NumComps,     // Number of components
   TExtColorSpace CS, // Color space
   SI32 Handle)       // Color space handle of non-device color space

typedef enum
{
   tcBackColor     = 0, // Table, Columns, Rows, Cells -> default none
   tcBorderColor   = 1, // Table, Columns, Rows, Cells -> default black
```

```
   tcGridHorzColor = 2, // Table                             -> default black
   tcGridVertColor = 3, // Table                             -> default black
   tcTextColor     = 4  // Table, Columns, Rows, Cells -> default black
}TTableColor;
```

The function sets or changes the specified color and color space. The number of color components must match the number of components of the color space. The text color is inherited from the table, column, and row, in this order. The border color is inherited from columns and rows but not from the table since the table has its own border.

Extended color spaces will be deleted when the PDF file in memory is closed or when FreePDF() was called. The color spaces must be recreated when the table should be drawn in another PDF file. Color space handles may change when recreated in another PDF file. All such colors should be set again to avoid issues with invalid color space handles.

Width exception of Lab colors, color values are in the range 0 through 1. Lab colors support different ranges for the L*, a*, and b* components:

```
L* ->     0 .. 100
a* -> -128 .. 127
b* -> -128 .. 127
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetColWidth

Syntax:

```
LBOOL tblSetColWidth(
   const ITBL* Table, // Table pointer
   UI32 Col,          // Column index
   float Width,       // New width
   LBOOL ExtTable)    // Extend the table or adjust the other columns?
```

The function changes the width of a column. The width of a column can be changed in two different ways. If the parameter *ExtTable* is set to true, then the difference of the current and new column width is added to the table width and the widths of all other columns is left unchanged.

If *ExtTable* is set to false, the difference is subtracted from the next columns until the gap is zero. If the width of the column is larger as the table width then the table width is adjusted too. All other columns get a zero width in this case.

The column widths should be set from lower to higher indexes and no value should be set for the last column.

No column width should exceed the table width if *ExtTable* is set to false. Otherwise the widths of all other columns must be changed again. Change first the table width and then the column widths in such a case.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFlags

```
LBOOL tblSetFlags(
    const ITBL* Table, // Table pointer
    SI32 Row,          // Row index or -1
    SI32 Col,          // Column index or -1
    TTableFlags Flags) // see below
```

```
typedef enum
{
    tfDefault     = 0,    // Default
    tfStatic      = 1,    // Avoid the deletion with ClearContent()
    tfHeaderRow   = 2,    // Header rows are drawn first after a page break
    tfNoLineBreak = 4,    // Prohibit line breaks in cells whith text
    tfScaleToRect = 8,    // Applies to images and templates
    tfUseImageCS  = 16,   // Insert images in the native color space
    tfAddFlags    = 32,   // Add the new flags to the current ones?
    tfFixedTextHeight = 64 // If set, text does not increase the row height if necessary. The row
                           // height must be non-zero in this case.
}TTableFlags;
```

The function sets various flags. Flags are inherited from the table, column, and rows, in this order.

Flag	Description
tfStatic	This flag avoids the deletion of cell contents when ClearContent(), ClearColumn() or ClearRow() is called. The flag can be set to rows, columns, and cells.
tfHeaderRow	Header rows are the first rows which are drawn after a page break occurred. It is possible to mark more than one row as header row.
tfNoLineBreak	This flag avoids line breaks in cells which contain text. The flag is inherited from the table, columns, and rows. Text is output with WriteText() if this flag is set. See SetCellText() for further information.
tfScaleToRect	If set, images and templates are scaled into the cell so that both sides fit into the given width and height. The aspect ratio will be preserved. The flag is inherited from the table, columns, and rows.
tfUseImageCS	If set, images are inserted in the native image color space and embedded ICC profiles will be considered. The flag is inherited from the table, columns, and rows.
tfAddFlags	If set, the new flags are added to the current ones. If absent, the new flags override the previous value.
tfFixedTextHeight	If set, text does not increase the row height if necessary. The row height must be non-zero in this case. The flag is ignored otherwise.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFont

Syntax:

```
LBOOL tblSetFont(
    const ITBL* Table, // Table pointer
    SI32 Row,          // Row index or -1
    SI32 Col,          // Column index or -1
    const char* Name,  // Font name
    TFStyle Style,     // Font style
    LBOOL Embed,       // If true, the font will be embedded
    TCodepage CP)      // The code page that should be used
```

The function sets the font that is used to output text. The font is inherited from the table, column, and row, in this order. The font will be loaded when the table is drawn. The default font size is 10 units. It can be changed with SetFontSize(). The default font selection mode is smFamilyName. It can be changed with SetFontSelMode().

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Ansi strings are interpreted as UTF-8 if the flag gfAnsiStringIsUTF8 is set the in the associated PDF instance. See SetGStateFlags() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFontSelMode

Syntax:

```
LBOOL tblSetFontSelMode(
    const ITBL* Table, // Table pointer
    SI32 Row,          // Row index or -1
    SI32 Col,          // Column index or -1
    TFontSelMode Mode) // see below

typedef enum
{
    smFamilyName     = 0,
    smPostScriptName = 1,
    smFullName       = 2
}TFontSelMode;
```

The function sets or changes the font selection mode. The font names used in the corresponding SetFont() calls must coincide with the font selection mode. The font selection mode is inherited from the table, column, and row, in this order.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFontSize

Syntax:

```
LBOOL tblSetFontSize(
   const ITBL* Table, // Table pointer
   SI32 Row,          // Row index or -1
   SI32 Col,          // Column index or -1
   float Value)       // Font size
```

The function sets or changes the font size. The font size is inherited from the table, column, and row, in this order.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetGridWidth

Syntax:

```
LBOOL tblSetGridWidth(
   const ITBL* Table, // Table pointer
   float Horz,        // Horizontal grid width or zero
   float Vert)        // Vertical grid width or zero
```

The table grid consists of horizontal and vertical lines which are drawn in the middle of two rows or columns. The grid lines cover always the entire width or height of the table. The line widths can be set to different values for the horizontal and vertical lines. If no lines should be drawn then set the value to zero. The colors of horizontal and vertical grid lines can be set with SetColor() or SetColorEx().

Only the table supports grid lines. The parameters *Row* and *Col* of SetColor() or SetColorEx() must be set to -1.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPDFInstance

Syntax:

```
void tblSetPDFInstance(
   const ITBL* Table, // Table pointer
   const PPDF* IPDF)  // Pointer of a valid PDF Instance
```

The function changes the PDF instance that is associated with the table. The PDF instance must be changed if the table should be drawn into another PDF instance as the one for which the table was created.

Note that extended color spaces and templates depend on the PDF instance in which these objects were created. If the table contains such objects then it is normally not possible to change the PDF instance.

This function is only included in the C interface.

SetRowHeight

Syntax:

```
LBOOL tblSetRowHeight(
   const ITBL* Table, // Table pointer
   UI32 Row,          // Row index
   float Value)       // New height
```

The function changes the row height. The value must be greater or equal zero. Foreground objects expand the height if necessary. The height must be greater zero if a cell uses a landscape orientation and if the height cannot be computed from the cell content. This is the case if a cell contains text and if the flag tfScaleToRect is not set.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetTableWidth

Syntax:

```
void tblSetTableWidth(
   const ITBL* Table,        // Table pointer
   float Value,              // New width (must be greater zero)
   TColumnAdjust AdjustType, // see below
   float MinColWidth)        // Must be >= 0 (see description)

typedef enum
{
   coaUniqueWidth, // Set the column widths to TableWidth / NumColumns
   coaAdjLeft,     // Adjust the widths starting from the left side
   coaAdjRight     // Adjust the widths starting from the right side
}TColumnAdjust;
```

The function changes the width of the table. At least one column must be modified when the width will be changed. The parameter *AdjustType* specifies how the column widths should be modified. If the new width is larger, then the difference can be added to the left or right column, or all columns can get a unique width (TableWidth / NumColumns). The parameter *MinColWidth* is ignored in this case.

If the new width is smaller, then the difference must be subtracted from the columns. The parameter *MinColWidth* specifies in this case the minimum width of columns that must be adjusted. It is only used if *AdjustType* is not set to coaUniqueWidth. The column widths are adjusted starting from the left or right side as specified. If the width of the first column is not large enough to subtract the entire

difference, then the minimum width is set to the column and the adjustment continues with the next column until the gap is zero.

If the new table width is smaller as *MinColWidth* * NumColumns then the widths of the remaining columns will be set to zero. Such cases should be avoided since the column widths must be adjusted again when the table is drawn.

Function Reference

This section describes all supported functions of DynaPDF in detail. Most examples in this chapter are written in C, C++, or Delphi. However, as you can see, the usage of DynaPDF is nearly identical with all programming languages.

Abort (Rendering Engine)

Syntax:

```
void rasAbort(
   IRAS* RasPtr) // Instance pointer of the rasterizer
```

This function provides a safe way to stop the function RenderPage() very quickly when it is running in a separate thread. The function sets an internal abort flag so that the rendering engine can safely terminate the current rendering process. The flag is automatically reset when the function RenderPage() is called the next time.

ActivateAltFontList

Syntax:

```
LBOOL pdfActivateAltFontList(
   const PPDF* IPDF,       // Instance pointer
   SI32 Handle,            // -1 or a list handle -> CreateAltFontList()
   LBOOL EnableSysFonts)
```

The function activates or deactivates an alternate font list that was created by CreateAltFontList(). An alternate font list represents a list of fallback fonts which are tested if one or more required glyphs were not found in the current font.

If no compatible font was found in the list, then system fonts are tested if *EnableSysFonts* was set to true. Although *EnableSysFonts* should be set to true in most cases, it is sometimes useful to disable system fonts, e.g. to restrict the number of fonts which can be embedded.

In order to disable the current alternate font list set the parameter *Handle* to -1. The value of *EnableSysFonts* is ignored this case. To fully deactivate font substitution disable system fonts with SetUseSystemFonts() before calling a text function.

Please note that alternate font lists are used only, if complex text layout was enabled. This can be done by setting the flag *gfComplexText* with SetGStateFlags().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddActionToObj

Syntax:

```
SI32 pdfAddActionToObj(
   const PPDF* IPDF, // Instance pointer
   TObjType ObjType, // The type of object that should get the action
   TObjEvent Event,  // Which event should trigger the action
   UI32 ActHandle,   // Action Handle
   UI32 ObjHandle)   // Object Handle (see notes below)
```

```
typedef enum
{
   otAction,
   otAnnotation,
   otBookmark,
   otCatalog,          // PDF 1.4
   otField,
   otPage,
   otPageLink
}TObjType;
```

```
typedef enum
{
    oeNoEvent,          // Internal use only -> DO NOT USE THIS VALUE!!!
    oeOnOpen,           // Catalog, Pages
    oeOnClose,          // Pages only
    oeOnMouseUp,        // All fields, page link annotations, bookmarks
    oeOnMouseEnter,     // Form fields only
    oeOnMouseExit,      // Form fields only
    oeOnMouseDown,      // Form fields only
    oeOnFocus,          // Form fields only
    oeOnBlur,           // Form fields only
    oeOnKeyStroke,      // Text fields only
    oeOnFormat,         // Text fields only
    oeOnCalc,           // Text fields, combo boxes, list boxes
    oeOnValidate,       // All form fields, except buttons
    oeOnPageVisible,    // PDF 1.5 -> Form fields only
    oeOnPageInVisible,  // PDF 1.5 -> Form fields only
    oeOnPageOpen,       // PDF 1.5 -> Form fields only
    oeOnPageClose,      // PDF 1.5 -> Form fields only
    oeOnBeforeClosing,  // PDF 1.4 -> Catalog, must be a JavaScript Action
    oeOnBeforeSaving,   // PDF 1.4 -> Catalog, must be a JavaScript Action
    oeOnAfterSaving,    // PDF 1.4 -> Catalog, must be a JavaScript Action
    oeOnBeforePrinting, // PDF 1.4 -> Catalog, must be a JavaScript Action
    oeOnAfterPrinting   // PDF 1.4 -> Catalog, must be a JavaScript Action
}TObjEvent;
```

This function adds an action to a PDF object. The parameter ActHandle requires a handle that was returned by a function that creates an action object such as CreateGoToAction(), etc. The object handle is also a return value of its creation function. If the object type is a page, then use the page number as handle.

It is possible to add multiple actions to one object, but note that DynaPDF does not check whether all actions are valid. For example, it is possible to add more than one action to a text field to the OnFormat event, but Acrobat executes only the first one. Always test your actions with different Acrobat versions because each version handles actions in a different manner.

The OnMouseUp event is the standard event supported by all objects. This event type supports also all action types. The other events are handled differently depending on the Acrobat version. Acrobat

4 supports JavaScript actions only in non-OnMouseUp events. Newer versions support also predefined action types. However, test your forms with the Acrobat versions that should be supported - this is the only way to find incompatibilities.

Return values:

If the function succeeds the return value is 1, if the function fails the return value is 0.

AddAnnotToPage

Syntax:

```
LBOOL pdfAddAnnotToPage(
    const PPDF* IPDF, // Instance pointer
    UI32 PageNum,     // Page number
    UI32 Handle)      // Annotation handle
```

The function adds an annotation to a page. With exception of PopUp annotations all annotation types can be drawn on multiple pages. Watermark or Stamp annotations are typical annotations which can be placed on multiple pages.

Return values:

If the function succeeds the return value is 1, if the function fails the return value is 0.

AddArticle

Syntax:

```
SI32 pdfAddArticle(
    const PPDF* IPDF, // Instance pointer
    double PosX,      // X-Coordinate of the Article
    double PosY,      // Y-Coordinate of the Article
    double Width,     // Width in unscaled units
    double Height)    // Height in unscaled units
```

This function creates a new article and adds it to the currently open article thread that must be created with CreateArticleThread() beforehand.

Some types of document may contain sequences of content items that are logically connected but not physically sequential. For example, a news story may begin on the first page of a newsletter and run over onto one or more non-consecutive interior pages. To represent such sequences of physically discontinuous but logically related items, a PDF document may define one or more articles.

The sequential flow of an article is defined by an article thread; the individual content items that make up the article are called beads on the thread. PDF viewer applications such as Adobe's Acrobat provide navigation facilities to allow the user to follow a thread from one bead to the next.

Remarks:

Due to a bug in Acrobat 6 articles can cause a zoom out. The pages of a document appear then as small thumb nails.

Return values:

If the function succeeds the return value is the handle of the article, a value greater or equal zero. If the function fails the return value is a negative error code.

AddBookmark

Syntax:

```
SI32 pdfAddBookmark(
   const PPDF* IPDF,  // Instance pointer
   const char* Title, // Title of the bookmark
   SI32 Parent,       // Parent bookmark or -1, see description
   UI32 DestPage,     // Destination page
   LBOOL Open)        // Open or close the node when it contains children
```

This function adds a bookmark to the global outline tree of the document. It sets also the page mode to pmUseOutlines (see SetPageMode()). If the outline tree should not be shown when opening the document, then set the page mode back to pmUseNone or any other value before closing the PDF file.

Parent can be a handle of another bookmark if it should be added as a child of this bookmark. If *Parent* is set to **-1** the bookmark is added as a root node.

DestPage specifies the page that should be opened when clicking on the bookmark. The parameter can be set to a page number that does not yet exist. If the page does not exist when closing file then it is adjusted to page number 1.

The parameter *Open* specifies whether a bookmark node should appear open or closed when it contains children. When a bookmark contains no children then this parameter should be set to false.

Related functions:

AddBookmarkEx()
ChangeBookmark()
InsertBookmark()
InsertBookmarkEx()
SetBookmarkDest()
SetBookmarkStyle()

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252 only. To create a bookmark in an arbitrary encoding convert the string to Unicode with the function ConvToIncode() and use the Unicode version to create the bookmark.

Return values:

If the function succeeds the return value is the handle of the bookmark, a value greater or equal zero. If the function fails the return value is a negative error code.

Example (C++):

```cpp
#include "dynapdf.h"
using namespace DynaPDF;

// first we declare an error callback function
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage); // just display the error message
   return 0;
}

int main(int argc, char* argv[])
{
   void* pdf = pdfNewPDF();
   if (!pdf) return 2;
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfCreateNewPDF(pdf, "c:\cppout.pdf");
   pdfAppend(pdf);
   // we store the font handle, so that we can use pdfChangeFont() later
   SI32 f = pdfSetFont(pdf, "Arial", fsItalic, 40, true, cp1252);
   pdfWriteFText(pdf, taCenter, "How to create an outline tree...");
   pdfEndPage(pdf);
   // we create a few dummy pages to test the bookmarks
   char s[30];
   for (i = 0; i < 7; i++)
   {
      pdfAppend(pdf);
      pdfChangeFont(pdf, f); // set the font with same parameters
      sprintf(s, "Page %d", i+2);
      pdfWriteFText(pdf, taLeft, s);
      pdfEndPage(pdf);
   }
   // The first item is always a root node, Parent must be -1.
   SI32 bmk = pdfAddBookmark(pdf, "First root node", -1, 1, true);
   pdfAddBookmark(pdf, "1. sub node", bmk, 2, false);
   pdfAddBookmark(pdf, "2. sub node", bmk, 3, false);
   pdfAddBookmark(pdf, "3. sub node", bmk, 4, false);
   bmk = pdfAddBookmark(pdf, "Closed sub node", bmk, 5, false);
   pdfAddBookmark(pdf, "1. sub node", bmk, 6, false);
   pdfAddBookmark(pdf, "2. sub node", bmk, 7, false);
   bmk = pdfAddBookmark(pdf, "Closed root node", -1, 1, false);
   pdfAddBookmark(pdf, "1. sub node", bmk, 4, false);
   pdfAddBookmark(pdf, "2. sub node", bmk, 8, false);
```

```
    pdfCloseFile(pdf);
    return 0;
}
```
Output:
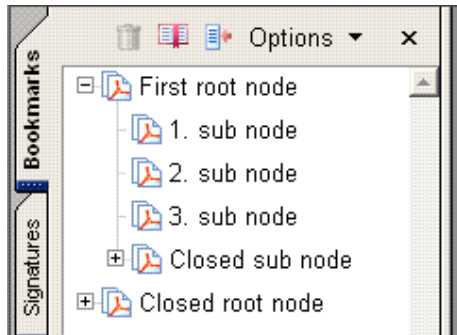
AddBookmarkEx

Syntax:

```
SI32 pdfAddBookmarkEx(
   const PPDF* IPDF,  // Instance pointer
   const char* Title, // Title of the bookmark
   SI32 Parent,       // Parent bookmark if any or -1 for a root node
   UI32 NamedDest,    // Named Destination handle
   LBOOL Open)        // Open or close the node when it has children?
```

This function adds a bookmark to the global outline tree of the document. It sets also the page mode to pmUseOutlines (see SetPageMode()). If the outline tree should not be shown when opening the document, set the page mode back to pmUseNone or any other value before closing the document.

Parent can be a handle of another bookmark if it should be added as a child of this bookmark. If *Parent* is set to **-1** the bookmark is added as a root node. When a bookmark should be inserted at a specific position then use the function InsertBookmarkEx() instead.

The parameter *NamedDest* must be a valid handle of a Named Destination, see CreateNamedDest().

If the named destination lies in another document then create a normal bookmark with AddBookmark() or InsertBookmark(), create an extended Go To Remote Action with CreateGoToRActionEx(), and add the action finally to the bookmark with AddActionToObj().

The parameter *Open* specifies whether a bookmark node should appear open or closed when it contains children. When a bookmark contains no children then this parameter should be set to false.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252 only. To create a bookmark in an arbitrary encoding convert the string to Unicode with the function ConvToIncode() and use the Unicode version to create the bookmark.

Return values:

If the function succeeds the return value is the handle of the bookmark, a value greater or equal zero. If the function fails the return value is a negative error code.

AddBookmarkEx2

Syntax:

```
SI32 pdfAddBookmarkEx2(
    const PPDF* IPDF,        // Instance pointer
    const char* Title,       // Title of the bookmark
    SI32 Parent,             // Parent bookmark if any or -1 for a root node
    const void* NamedDest,   // Name of a named destination
    LBOOL Unicode,           // NamedDest is an Unicode String?
    LBOOL Open)              // Open or close the node when it has children?
```

This function adds a bookmark to the global outline tree of the document. It sets also the page mode to pmUseOutlines (see SetPageMode()). If the outline tree should not be shown when opening the document, set the page mode back to pmUseNone or any other value before closing the document.

Parent can be a handle of another bookmark if it should be added as a child of this bookmark. If *Parent* is set to **-1** the bookmark is added as a root node.

The difference in comparison to AddBookmarkEx is that the named destination can be defined as a string. This makes it possible to create the bookmark, also if you don't have the required information to create the named destination at this point. If the named destination does not exist when the file is closed then the bookmark does nothing. See also CreateNamedDest().

The parameter Unicode specifies whether the parameter *NamedDest* is an Ansi or Unicode string (UTF-16). Note that the usage of Unicode is **not** recommended since named destinations are treated as binary strings.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252 only. To create a bookmark in an arbitrary encoding convert the string to Unicode with the function ConvToIncode() and use the Unicode version to create the bookmark.

Return values:

If the function succeeds the return value is the handle of the bookmark, a value greater or equal zero. If the function fails the return value is a negative error code.

AddButtonImage

Syntax:

```
SI32 pdfAddButtonImage(
   const PPDF* IPDF,     // Instance pointer
   UI32 BtnHandle,       // Handle of the button which should get the image
   TButtonState State,   // Button state, see below
   const char* Caption,  // Caption to be printed over the image or NULL
   const char* ImgFile)  // Image file

typedef enum
{
   bsUp,       // Normal up state of the button
   bsDown,     // Down state, button is pressed
   bsRollOver  // Rollover state, when moving the cursor over the button
}TButtonState;
```

The function adds an image to a push button to one or more of the three different states of a button. The parameter *BtnHandle* must be handle of a button field that was returned by a CreateButton() function call. The parameter *State* defines the button state in which the image should appear. Each state is optional; it is possible to add different images to all three states of the button. The rollover state should get a different image than the other states. If two states should get the same image, the image is physically inserted only once to the PDF file.

DynaPDF checks whether duplicate images are already available before inserting an image again. Depending on the parameter *UseTransparency* the image appears with a transparent background or opaque (default transparent). However, note that the background of the button must be transparent too to get a transparent button. Otherwise the button is still opaque due the opaque background of the button.

The parameter *Caption* is optional, it can be NULL. Each state can contain its own caption which is drawn horizontally and vertically centered by default. The alignment can be changed with the function SetFieldTextAlign(). Because each button state is directly drawn when calling the function, the alignment must be set before this function is called.

If a button has already an image for a specific button state, the existing image will be replaced with the new one.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddButtonImageEx

Syntax:

```
SI32 pdfAddButtonImageEx(
   const PPDF* IPDF,      // Instance pointer
   UI32 BtnHandle,        // Handle of the button which should get the image
   TButtonState State,    // Button state
   const char* Caption,   // Caption to be printed over the image or NULL
   void* hBitmap)         // HBITMAP handle of a memory bitmap
```

The function adds a memory bitmap to a push button in the same ways as AddButtonImage() but accepts a HBITMAP handle as input image. The usage of the function is described under AddButtonImage().

Remarks:

The bitmap must **not** be selected into a device context when the application calls this function. This function is not thread-safe; it must be synchronized in multi-threaded applications.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddButtonImageEx2

Syntax:

```
LBOOL pdfAddButtonImageEx2(
   const PPDF* IPDF,      // Instance pointer
   UI32 BtnHandle,        // Handle of the button which should get the image
   TButtonState State,    // Button state
   const char* Caption,   // Caption to be printed over the image or NULL
   const void* Buffer,    // Image buffer
   UI32 BufSize)          // Buffer size in bytes
```

The function adds an image to a push button in the same way as AddButtonImage(), but accepts an image buffer as input. *Buffer* must contain a valid image buffer. The function supports all image formats that InsertImageEx() supports.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddContinueText

Syntax:

```
SI32 pdfAddContinueText(
    const PPDF* IPDF,  // Instance pointer
    const char* AText) // Text to be printed
```

The function prints a single text line and moves the text cursor to the next line. Text lines created by this function are handled as a separate block. Before calling the function the first time, the usage must be prepared with BeginContinueText(). After all text lines are printed call EndContinueText() to finish the text block.

The distance between two text lines can be adjusted with SetLeading().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddDeviceNProcessColorants

Syntax:

```
LBOOL pdfAddDeviceNProcessColorants(
    const PPDF* IPDF,          // Instance pointer
    UI32 DeviceNCS,            // Handle of a DeviceN or NChannel color
space
    const char** Colorants,    // Array of colorant names
    UI32 NumColorants,         // Number of colorants in the array
    TExtColorSpace ProcessCS,  // Process color space
    SI32 Handle)               // Color space handle or -1
```

The function adds a dictionary to a DeviceN or NChannel color space that describes the process color space whose components are included in the DeviceN or NChannel color space.

The array *Colorants* holds the process colorant names which are included in the process color space. The names must appear in the normal color space order, e.g. Red, Green, and Blue, for a RGB color space and the number of colorant names must match the number of color components in the process color space. However, the names in the *Colorants* array need not match the actual color space names. For example, a Red component need not be named Red.

Definitions of process colorants must not appear in the definition of spot colorants (see AddDeviceNSeparations()).

The reserved names Cyan, Magenta, Yellow, and Black are always considered to be process colors, which do not necessarily correspond to the colorants of a specific device; they are not required to have entries in the process dictionary.

If *NumColorants* is zero the functions deletes existing definitions of process colorants in the DeviceN or NChannel color space. Otherwise, existing definitions will be overridden.

Remarks:

Process colorant names must be defined in the code page 1252 (WinAnsi).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddDeviceNSeparations

Syntax:

```
LBOOL pdfAddDeviceNSeparations(
  const PPDF* IPDF,        // Instance pointer
  UI32 DeviceNCS,          // Handle of a DeviceN or NChannel color space
  const char** Colorants,  // Array of colorant names
  UI32* SeparationCS,      // Array of Separation color space handles
  UI32 NumColorants)       // Number of colorants and handles
```

The function adds a dictionary to a DeviceN or NChannel color space that describes the spot colorants which are used in the color space.

This dictionary provides information about the individual colorants that may be useful to some applications. In particular, the alternate color space and tint transformation function of a Separation color space describe the appearance of that colorant alone, whereas those of a DeviceN color space describe only the appearance of its colorants in combination.

The array *Colorants* holds the spot colorant names and the array *SeparationCS* holds the corresponding Separation color space handles. The colorant names in the array must match the names in the corresponding Separation color spaces. In addition, the colorants must be included in the DeviceN or NChannel color space.

If *NumColorants* is zero the functions deletes existing definitions of spot colorants in the DeviceN or NChannel color space. Otherwise, existing definitions will be overridden.

Encoding of Colorant Names

Colorant names are interpreted in the code page 1252 by default. Because colorant names are stored in UTF-8 Unicode format in PDF, it is also possible to pass UTF-8 encoded Unicode strings to the function. However, the function must be able to distinguish between both string formats. To achieve this, the parameter *NumColorants* accepts the special flag 0x10000000 that specifies that the *Colorants* array contains UTF-8 encoded strings. The flag must be combined with the number of colorants with a binary or operator:

```
numColorants |= 0x10000000;                     // C/C++, C#
numColorants  = numColorants Or &H10000000 // Visual Basic
numColorants := numColorants or $10000000  // Delphi
```

Remarks:

The definition of spot colors is required to enable the output preview in Adobe's Acrobat.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddDPartNode

Syntax:

```
SI32 pdfAddDPartNode(
    const PPDF* IPDF, // Instance pointer
    SI32 Parent,      // Parent node handle or -1
    SI32 FirstPage,   // First page in range or -1
    SI32 LastPage,    // Last page in range or -1
    const char* DPM)  // Optional Document Part Metadata (JSON like string)
```

The function creates a *DPart Node*. *DPart* nodes can be used to add metadata to a single page or range of pages. In addition, *DPart* nodes can be nested to form a tree like structure.

Unlike XMP metadata streams which base on XML, *Document Part Metadata* (parameter *DPM*) is stored in PDF syntax. This type of metadata supports dictionaries, arrays, strings, name objects, numbers, and boolean values, or almost the entire range of PDF data types.

DPart nodes require a global root node that must be created with CreateDPartRoot() before this function can be called the first time.

Two different types of *DPart* nodes can be created with this function:

- If *FirstPage* and *LastPage* refer to valid page numbers (the first page is denoted by 1), the function creates a *DPart* node with optional *DPart* metadata (parameter *DPM*) that will be associated to these pages.

  The *Parent* node can be -1 for the root node, or a node handle that was returned by this function. However, nodes which reference pages, cannot be used as parent node for other *DPart* nodes.

- If *FirstPage* or *LastPage* are smaller 1, a *DPart* node will be created that can be used as parent node for other *DPart* nodes. This is useful for grouping nodes or to create a tree like structure, but in most cases such nodes are mainly used to define a base set of identical metadata for a range of *DPart* nodes. Each child node can still define additional metadata.

Document Part Metadata (DPM)

DPM is application-specific information that is generally arbitrary. However, document parts are part of the PDF/VT standard and therefore mainly used to define print specific data, e.g. job definition.

Although the metadata can contain arbitrary key / value pairs of data, there is a common naming sheme defined in ISO 21812-1:2019, for example, that describes how print production metadata can be named and organized. Keywords in this standard start with the prefix CIP4_.

How to create DPM?

Since *DPM* can contain all basic data types that PDF supports, we need an exchange format that is easy to create and that supports very similar data types. The exchange format that matches these requirements is called JSON.

The JSON parser in DynaPDF supports a few extensions to make the definition of *DPM* as easy as possible.

Differences in comparison to regular JSON grammar:

- A JSON string begins normally with a left brace ({) and ends with a right brace (}). This outer left / right brace pair can be omitted.

- Key / value pairs must normally be delimited by a colon (:). The delimiting colon can be omitted.

- Array values must normally be delimited by a comma (,). The delimiting comma can be omitted if the bounds of the value can be detected by whitespace or starting character of the next object.

  Example (C/C++):
  ```
  const char dpm[] = "\"TestArray\"[1/2\"3\"4 5]";
  ```

  Result:
  ```
      /DPM<</TestArray[1/2(3)4 5]...>>
  ```

  The first value of the array is the number 1, the second is a name object with the value "2", the third value is the string "3", the fourth is the number 4, and the fifths is the number 5.

- **Extension:** A Solidus (/) begins a PDF Name object. The solidus is not part of the name but is a prefix indicating what follows is a sequence of characters representing a name object. A name object supports character codes between 33..126 inclusive. Any character outside this range must be written as two digit hexadecimal code, preceded by a number sign (#), e.g. /This#20is#20a#20name#20with#20spaces. Note that a number sign in a name object must be written as hexadecimal code too, since the number sign is treated as an escape character. The de-escaped string is interpreted as UTF-8 Unicode string.

  A character code < 33, { (left brace), } (right brace), [(left bracket),] (right bracket), , (comma), : (colon), or " (double quotes) terminate a name object. If one of these characters should be part of a name object then write it as two digit hexadecimal code preceded by a number sign (#).

- A left brace ({) creates a PDF dictionary and a right brace (}) finishes or closes it.

- A left bracked ([) creates an array and a right bracked (]) finishes or closes it.

Data types supported as value of a key are: array, dictionary, name, string, boolean true or false (case sensitive), null, and number. A JSON string must be enclosed with double quotes. Example: "This is a string".

In PDF, keys are always defined as name objects. The function converts a key (or name in JSON nomenclatur) automatically to a name object. Values are never converted.

Example (C++):

The file test.json that is loaded in this example contains the contents below. This is basically a JSON version of the PDF/VT example provided in the PDF/VT-2 reference (ISO 16612-2), plus a few additional test key / value pairs.

Have a deeper look at "CIP4_TestArray1" and " CIP4_TestArray2". These key / value pairs contain several syntax errors which a regular JSON parser would not accept.

```
{
    "ACME_CustStatus": "Prospective"
    "CIP4_Root" :
    {
        "CIP4_Summary" :
        {
            "CIP4_PageCount" : 6
            "CIP4_Checked" true
        }
        "CIP4_Production" :
        {
            "CIP4_CopyCount" : 1
            "CIP4_Part":
            {
                "CIP4_ProductType" : "Brochure"
            }
        }
        "CIP4_Recipient" :
        {
            "CIP4_Contact" :
            {
                "CIP4_Address" :
                {
                    "CIP4_City": "Phoenix"
                    "CIP4_CivicNumber": "96"
                    "CIP4_Country" :"USA"
                    "CIP4_PostalCode" :"81215"
                    "CIP4_Region" :"AZ"
                    "CIP4_StreetName" :"South Ave"
                    "CIP4_TestArray1" [1 2 3 4 true, false]
                    "CIP4_TestArray2"
                    {
                        "Array"[5 6 7 8 "Test1"/Test2]
                    }
                }
                "CIP4_Person" :
                {
                    "CIP4_FirstName" :    "Mary"
                    "CIP4_LastName" : "Smith"
                }
            }
            "CIP4_UniqueID" : "ID_1"
        }
    }
}
```

The above file produces the output in PDF below. Note that the dictionary was formatted for better readability. No additional spaces or line separators are written to a real PDF file.

```
/DPM
<<
    /ACME_CustStatus(Prospective)
    /CIP4_Root
    <<
        /CIP4_Summary
        <<
            /CIP4_PageCount 6
            /CIP4_Checked true
        >>
        /CIP4_Production
        <<
            /CIP4_CopyCount 1
            /CIP4_Part
            <<
                /CIP4_ProductType(Brochure)
            >>
        >>
        /CIP4_Recipient
        <<
            /CIP4_Contact
            <<
                /CIP4_Address
                <<
                    /CIP4_City(Phoenix)
                    /CIP4_CivicNumber(96)
                    /CIP4_Country(USA)
                    /CIP4_PostalCode(81215)
                    /CIP4_Region(AZ)
                    /CIP4_StreetName(South Ave)
                    /CIP4_TestArray1[1 2 3 4 true false]
                    /CIP4_TestArray2
                    <<
                        /Array[5 6 7 8(Test1)/Test2]
                    >>
                >>
                /CIP4_Person
                <<
                    /CIP4_FirstName(Mary)
                    /CIP4_LastName(Smith)
                >>
            >>
            /CIP4_UniqueID(ID_1)
        >>
    >>
>>
```

```
char* GetFileBuf(const char* FileName, SI32 &BufSize)
{
    BufSize = 0;
    FILE* f = fopen(FileName, "rb");
    if (!f) return NULL;
    fseek(f, 0, SEEK_END);
    BufSize = ftell(f);
    fseek(f, 0, SEEK_SET);
    char* retval = (char*)malloc(BufSize +1);
    if (!retval)
    {
        fclose(f);
        return NULL;
    }
    fread(retval, 1, BufSize, f);
    retval[BufSize] = 0;
    fclose(f);
```

```
     return retval;
}

SI32 TestDPart(const PPDF* PDF, const wchar_t* InFileName, wchar_t* OutFileName, bool Compress)
{
     SI32 retval = 0, bufSize, last, node;
     pdfCreateNewPDF(PDF, NULL);
     pdfSetDocInfo(PDF, diProducer, NULL); // No need to override the producer
     pdfSetImportFlags(PDF, ifImportAll | ifImportAsPage);
     pdfSetImportFlags2(PDF, if2UseProxy);
     if (InFileName)
     {
         if ((retval = pdfOpenImportFileW(PDF, InFileName, ptOpen, NULL)) < 0)
         {
             if (IsWrongPwd(retval))
             {
                 // A password dialog should normally be displayed here
                 printf("File is encrypted!\n");
             }
             goto err;

         }
         if ((retval = pdfImportPDFFile(PDF, 1, 1.0, 1.0)) < 0) goto err;
     }

     // Create the required root node
     const char* nameList[] = {"Root", "Pages"};
     pdfCreateDPartRoot(PDF, nameList, 2, 3);

     // GetFileBuf() is a helper function that returns the contents of an arbitrary file. See
     // definition above.
     char* dpmbase = GetFileBuf("f:/test.json", bufSize);
     if (!dpmbase)
     {
         printf("Cannot open JSON file!\n");
         goto err;
     }
     // This node holds some DPM for two child DPart nodes.
     node = pdfAddDPartNode(PDF, -1, -1, -1, dpmbase);
     free(dpmbase);

     // We create two ranges for test purposes.
     last = pdfGetPageCount(PDF) / 2;
     const char json2[] = "\"CIP4_Production\":{\"CIP4_CopyCount\" : 1\"CIP4_User\":\"Stefan\"}";
     const char json3[] = "\"CIP4_Production\":{\"CIP4_CopyCount\" : 1\"CIP4_User\":\"Jens\"}";

     pdfAddDPartNode(PDF, node, 1, last, json2);
     pdfAddDPartNode(PDF, node, last +1, pdfGetPageCount(PDF), json3);

     // If you want to have a look into the resulting PDF file then deactivate object compression.
     if (!Compress)
         pdfSetGStateFlags(PDF, gfNoObjCompression | gfDoNotComprMetadata, false);
     else
         pdfSetGStateFlags(PDF,  gfDoNotComprMetadata, false);
     if (pdfHaveOpenDoc(PDF))
     {
         if ((retval = pdfOpenOutputFileW(PDF, OutFileName)) == TRUE)
             retval = pdfCloseFile(PDF);
     }
     return retval;
  err:
     pdfFreePDF(PDF);
     return retval;
}
```

Return values:

If the function succeeds the return value is the handle of the *DPart* node, a value greater or equal zero. If the function fails the return value is a negative error code.

AddFieldToFormAction

Syntax:

```
SI32 pdfAddFieldToFormAction(
   const PPDF* IPDF, // Instance pointer
   UI32 Action,      // Handle of a submit or reset form action
   UI32 AField,      // Handle of a field that should be added to array
   LBOOL Include)    // Include or exclude this field from submitting?
```

Instead of simply submitting all field values of an interactive form to a web server, it is also possible to submit the values of specific fields only. A submit form and reset form action can hold an optional array of fields which can be included or excluded from the action. For example, if a form contains 20 fields which all should be submitted to the web server apart from two of those fields, then add the two fields to the submit form action and set the parameter *Include* to false.

Note, that parameter Include is handled reversed if the submit flag sfExclude is set (see also CreateSubmitAction(), CreateResetAction()). The fields are then excluded if *Include* is true and vice versa.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddFieldToHideAction

Syntax:

```
SI32 pdfAddFieldToHideAction(
   const PPDF* IPDF, // Instance pointer
   UI32 HideAct,     // Handle of the hide action
   UI32 AField)      // Field handle
```

This function adds a field to a hide action. An arbitrary count of fields can be added to a hide action.

Remarks:

Acrobat 4 does not support radio buttons or group fields inside a hide action. The fields of a group field or the check boxes of a radio button field must be added to a hide action separately. Since Acrobat 5, group fields and radio buttons are supported too.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddFileComment

Syntax:

```
SI32 pdfAddFileComment(
   const PPDF* IPDF,  // Instance pointer
   const char* AText) // The comment that should be added
```

Under certain circumstances it can be useful to add one or more comments to a PDF file which are not visible on screen or elsewhere in PDF viewer applications. The function AddFileComment() can be used to add such comments to the end of the PDF file.

Comments are stored behind the %%EOF marker of the PDF file. The required space to save all comments should not exceed 1024 bytes. However, all tests with larger comments didn't produce an error in Adobe's Acrobat, but Ghostview fails if the space exceeds ~2000 bytes.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. However, Unicode strings are converted back to Ansi; unsupported characters appear as question mark (?).

File comments are prohibited in certain PDF standards like PDF/A.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddFontSearchPath

Syntax:

```
SI32 pdfAddFontSearchPath(
   const PPDF* IPDF,  // Instance pointer
   const char* APath, // Path to font file directory
   SI32 Recursive)    // Include subdirectories (search recursively)
```

This function adds a search path to the list of available font search paths. An arbitrary number of search paths can be added at runtime. Subdirectories are added recursively if the parameter *Recursive* is true.

DynaPDF requires no separate metric files for Type1 fonts.

This function is implemented in an Ansi and Unicode compatible version. On non-Windows operating systems, Unicode paths are converted to UTF-8 and passed to the Ansi version of the function.

Extensions:

Since DynaPDF version 3.0.21.56 it is also possible to pass a font file to the function. This makes it possible to load a specific font file with the same rules as other fonts in a search directory. The file extension is not used to identify the font format. Resource forks, suitcases and dfont resources are

supported on Mac OS X. Note that the font file is not loaded into memory; it is just added to the list of font files.

Remarks:

The list of search directories can be cleared at runtime with the function ClearHostFonts().

If system fonts are enabled (default) DynaPDF adds the default font directories of the operating system automatically to the list of font search paths (Windows and Mac OSX only, see SetUseSystemFonts() for further information). On Windows this is the %WINDIR%/Fonts directory as well as fonts listed in the Registry and linked fonts.

On Mac OSX the following directories are added to the list of font search paths (in this order):

- ~/Library/Fonts
- /Library/Fonts
- /System/Library/Fonts

Return values:

If the function succeeds the function returns the number of found font files (this is maybe not the number of available fonts, because the list is cleared each time all fonts in the list are processed during font selection). If the function fails a negative error code is returned.

AddHeaderFooter

Syntax:

```
LBOOL pdfAddHeaderFooter(
    const PPDF* IPDF,                 // Instance pointer
    struct TPDFHeaderFooter* Init,    // Initial header / footer settings
    struct TPDFHdrFtr* HFArray,       // Array of header / footer values
    UI32 Count)                       // Number of array values

struct TPDFHdrFtr
{
    UI32          Color;       // Text color.
    TExtColorSpace CS;         // Color space -> default = esInitSpace.
    SI32          CSHandle;    // Optional color space handle if CS is set to a non-device space.
    TCodepage     Codepage;    // The code page that should be used to load the font.
    LBOOL         Embed;       // If true, the font will be embedded.
    SI32          Reserved1;   // Placeholder to avoid alignment issues
    const char*   FontA;       // Optional font name.
    const UI16*   FontW;       // Optional font name.
    float         FontSize;    // Font size -> considered only if a font name is set.
    SI32          Reserved2;   // Placeholder to avoid alignment issues
    const char*   FontType;    // Optional font type ("TrueType", "OpenType", or "Type1").
    LBOOL         IsHeader;    // If true, the record is treated as header.
    TTextAlign    Position;    // Valid values are taLeft, taCenter, and taRight.
    float         ShiftX;      // Positive values move the text to the right, negative to the left.
    float         ShiftY;      // Positive values move the text up or down depending on the used
                               // coordinate system (top down or bottom up).
    const char*   TextA;       // Either the Ansi or Unicode string must be set.
    const UI16*   TextW;       // Either the Ansi or Unicode string must be set.
    UI32          TextLen;     // Text length in characters.
    SI32          Reserved3;   // Placeholder to avoid alignment issues
};

typedef enum TPDFHdrFtrFlags
{
    hffDefault    = 3,   // Output header / footer text on even and odd pages.
    hffEvenPages  = 1,   // Output header / footer text on even pages.
    hffOddPages   = 2,   // Output header / footer text on odd pages.
    hffFixedPrint = 4,   // Can be set by LoadHeaderFooterSettings(). Not considered yet.
    hffLoadUTF8   = 8,   // Load all strings in UTF-8 format.
    hffShrink     = 16,  // Can be set by LoadHeaderFooterSettings(). Not considered yet.
    hffLoadBatesN = 32,  // If set, LoadHeaderFooterSetting() loads bates numbers only.
    hffLoadHF     = 64,  // If set, LoadHeaderFooterSetting() loads header / footer settings only.
    hffSearchRun  = 128  // See LoadHeaderFooterSetting() for further information.
}TPDFHdrFtrFlags;

struct TPDFHeaderFooter
{
    UI32          StructSize;     // Must be set to sizeof(TPDFHeaderFooter).
    SI32          FirstPage;      // In/Out -> The first output page.
    TPDFHdrFtrFlags Flags;        // In/Out -> Additional processing flags.
    UI32          InitColor;      // In/Out -> Initial text color.
    TExtColorSpace InitCS;        // In/Out -> Initial color space -> default = esDeviceGray.
    SI32          InitCSHandle;   // In     -> Required if InitCS is set to a non-device space.
    const char*   InitDate;       // Out    -> Initial date format.
    TCodepage     InitCodepage;   // In/Out -> The code page that should be used to load the font.
    LBOOL         InitEmbed;      // In     -> If true, the font will be embedded.
    const char*   InitFontA;      // In/Out -> Optional font name.
    const UI16*   InitFontW;      // In/Out -> Optional font name.
    float         InitFontSize;   // In/Out -> Optional, considered only if a font name is set.
    SI32          Reserved;       // Placeholder to avoid alignment issues
    const char*   InitFontType;   // Out    -> "TrueType", "OpenType", or "Type1".
    const char*   InitPageFormatA; // Out   -> Initial page format string.
    const UI16*   InitPageFormatW; // Out   -> Initial page format string.
    SI32          LastPage;       // In/Out -> The last output page.
    struct TFltRect Margin;       // In/Out -> Margin to output the header/footer text.
```

```
};
```

The function adds headers / footers to the document. Headers and footers support each three independent strings for the alignments left, center, and right. The text of every header or footer supports also placeholders to add bates numbers, date formats, and page numbers.

Notice:

This function mimics the bates numbering and header / footer tools which are available in certain PDF viewers like Adobe Acrobat and others. The function creates also Acrobat compatible XML properties so that headers / footers created with this function can be edited and deleted with Adobe Acrobat.

However, not every feature that is supported by this function has a counterpart in Acrobat or other PDF viewers. To achieve maximum compatibility a few restrictions must be taken into account wich are described on the following pages.

The creation of headers / footers starts with the initialization of the structure *TPDFHeaderFooter.* Set the member *StructSize* to sizeof(TPDFHeaderFooter) and call InitHeaderFooter() afterwards.

InitHeaderFooter() initializes the structure as follows:

- *InitColor:* The color space is set to *esDeviceRGB* and the color is set to black.

- *Flags:* Flags is set to *hffDefault*.

- *InitFont:* The font is set to the standard font "Helvetica", the font size is set to 8.0, the code page is set to cp1252, and the font type is set to "Type1". Note that the font type is set for information only, it is not used for font selection.

- *HFA*rray: The member *CS* set to *esInitSpace*. This means ignore the color and color space.

- All remaining members are intitialized to zero.

After the structure was initialized it is required to set at least one header or footer string. The parameter *HFArray* can contain up to 6 records to define headers / footers.

The member *IsHeader* specifies whether the text should be considered as header or footer, and the member *Position* specifies where the text should be output. The three alignments left, center, right, combined with *IsHeader* results into 6 possible combinations.

The parameter *Count* specifies the number of records which were set. It is allowed to have empty records in the array.

Margin specifies the distance between the text and the outer contour of the page. Text is vertically placed on the baseline. That implies that at margin [0, 0, 0, 0] header text is placed outside the visible area:
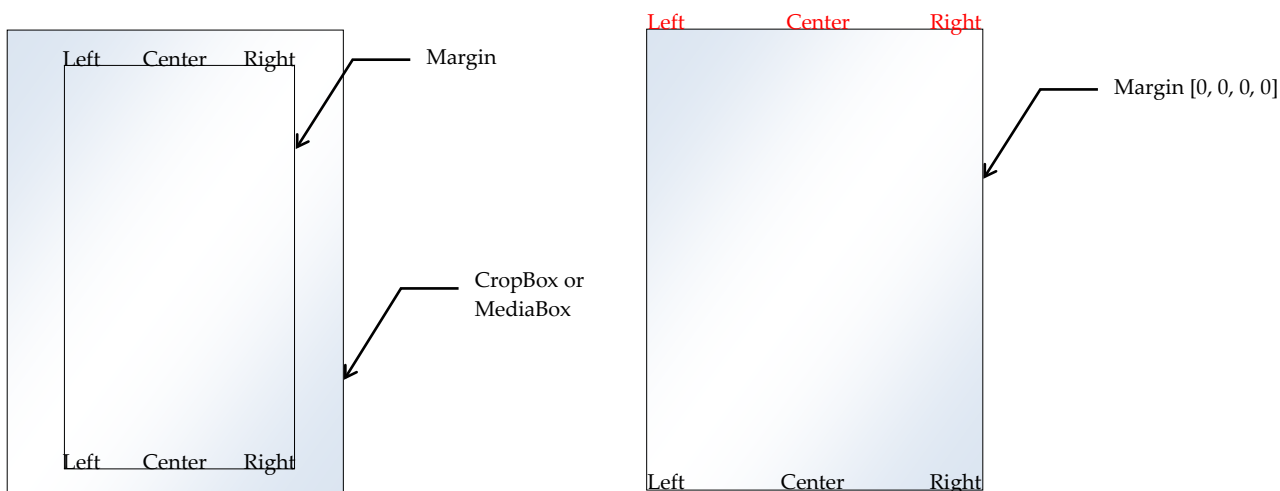
Note that Adobe Acrobat places text vertically to another position. It is not known how Acrobat calculates the y-coordinate.

Headers and footers can be defined in arbitrary order. The function sorts the records before they are processed. Double records are not allowed and records with no text will be ignored. Note that the alignment *taCenter* represents the middle of a page and not the middle of *Margin*.

Font selection

The font, font size, and so on that should be used to output headers or footers can be set globally with the parameter *Init* and individually for each record.

Before the first header or footer string is processed, the function loads the font that was specifed with the parameter *Init*. If *Init* defines no font, then the first header / footer record must specify a font. This font is then used until it is changed by another record.

If a *TPDFHdrFtr* record sets *FontA* or *FontW*, then it must also set *FontSize*, and *Codepage*. *Embed* specifies whether the font should be embedded. The function returns with an error if font selection fails.

If *Init* specifies a font, then this font is selected before a header or footer record is processed. If a header or footer record specifies another font then this font is used for this record only. It is not used for other records, as it is the case if *Init* contains no font definition.

Note that if compatiblity to Adobe Acrobat is required then *Init* must set a font.

The font name can be a Family, Postscript, or Full name. A family name can be combined with one of the style strings "*Bold*", "*Italic*", or "*BoldItalic*". Only these style strings are supported. A style string must be separated with a comma:

```
"Arial,Bold"            // Ok
"Arial, Bold"           // Wrong, no space must be present after the comma
"Courier New,BoldItalic" // Ok
```

Text color

The text color can be set globally with the parameter *Init* and individually for each record. If no color is specified then the default initialization of the graphics state is used. This is black defined as DeviceGray. Although the function supports arbitrary color spaces to output the text, the color is stored as rgb black in the XML data if a non-device color space is used. It is also not known whether the color spaces DeviceGray or DeviceCMYK are supported in Adobe Acrobat. To achieve best compatiblility use DeviceRGB only.

Bates numbering

A bates number is an integer number with a fixed length that is incremented on each page. The bates numbering syntax is as follows:

```
<<Bates#0..10#1..2147483648>> (case-sensitive)
```

The number sign (#) is part of the syntax and is required. No spaces before or after the number sign or before and after the keyword "Bates" are allowed. A valid Bates format contains no spaces and no prefix or suffix inside the format identifier <<>>. The first number specifies the number of digits, and the second where numbering starts.

Examples:

- `<<Bates#6#1>>` // Six digits, numbering starts at 1 -> 000001.
- `<<Bates#10#5>>` // Ten digits, numbering starts at 5 -> 0000000005.
- `<< Bates#6#1>>` // Wrong, no space is allowed.
- `<<bates#6#1>>` // Wrong, the correct keyword name is 'Bates'.

Date formats

The date format supports three characters: m = month, d = day, and y = year. Each character can be repeated to define the number of digits, e.g. mm, or yyyy are examples. Allowed delimiters are '.', '-', and '/'. The day and month format support one and two digits, and the year format supports two and four digits. A valid date format contains no spaces and no prefix or suffix inside the format identifier <<>>.

Examples:

- `<<yy/mm/d>>`   // Ok
- `<<mm-d/yy>>`   // Unusual but ok
- `Date: <<dd.mm.yyyy>>` // Ok
- `<y/mm/d>>`   // Wrong, missing '<' and a valid year format has 2 or 4 digits
- `<<Date: m/d>>`   // Wrong, no prefix or suffix is allowed
- `<<yyyy:mm:dd>>`   // Wrong, invalid delimiter

Page numbering

The page number format has two reserved characters:

```
•  0..2147483648 // The number is added to the current page index (the first page is 1).
•  'n'           // 'n' is replaced by the number of pages. The 'n' must be surrounded by a
                 // delimiter, otherwise it is treated as text.
•  Delimiters: ' ', ')', '(', '[', ']', '<', '>', '/', '%', '{', '}'.
```

Arbitrary text can occur before, between, and after a reserved character. Example results on the first page if the document has 57 pages:

```
•  <<28>>             // Ok, 28 is added to the page index.          -> "28"
•  << Page 4 of n>>   // Ok.                                         -> " Page 4 of 57"
•  <<1/n>>            // Ok.                                         -> "1/57"
•  <<0/n>>            // Ok, but unusual.                            -> "0/57"
•  <<-5 of n>>        // Ok, note that the minus is treated as text! -> "-5 of 57"
•  <<Page1 ofn pages>> // Ok, but the 'n' is treated as text.        -> "Page1 ofn pages"
•  <<Page 5 of n>>    // Ok, 5 is added to the page index.           -> "Page 5 of 57"
•  <<Seite 1 von n>>  // Ok.                                         -> "Seite 1 von 57"
•  <<Page>>           // Wrong, nothing to do. Treated as text.      -> "<<Page>>"
```

Although it is possible to output the page number without surrounding whitespace, the function saves the format with surrounding whitespace in the XML data to achieve better compatibility to Adobe Acrobat (except if a format like "1/n" is used).

However, the page numbering tool in Adobe Acrobat is very restricted. Therefore, it is not always guaranteed that Acrobat is able to load a page numbering format created by this function. Although if Acrobat will fail to edit a page numbering record, it is usually still possible to delete the header / footer definition and to add a new one.

Header / footer text

Every header / footer string can contain plain text and one or more format strings as described above. The maximum output length of a header / footer string is 511 characters.

Header / footer types

The function creates two different header / footer types depending on the contents in it. If a header or footer string contains a bates number, then the type *BatesN* is created, otherwise the type *Header*.

The two different types can be loaded and deleted separately. Header / footer settings can be loaded with LoadHeaderFooterSettings() and deleted with Optimize(). Optimize() supports the flags *ofRemoveBatesNumbers* and *ofRemoveHeaderFooter* in order to remove headers / footers from a PDF file.

Return value:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (C#):

```csharp
static int PDFError(IntPtr Data, int ErrCode, IntPtr ErrMessage, int ErrType)
{
    MessageBox.Show( string.Format("{0}",
                     System.Runtime.InteropServices.Marshal.PtrToStringAnsi(ErrMessage)));
    return 0; // We try to continue if an error occurrs. Any other return value breaks processing.
}

private void button1_Click(object sender, EventArgs e)
{
    CPDF pdf = new CPDF();
    pdf.SetOnErrorProc(IntPtr.Zero, new TErrorProc(PDFError));

    pdf.CreateNewPDF("e:/cppout.pdf");

    // Do not convert pages to templates.
    pdf.SetImportFlags(TImportFlags.ifImportAll | TImportFlags.ifImportAsPage);
    // This flag reduces the memory usage drastically
    pdf.SetImportFlags2(TImportFlags2.if2UseProxy);

    pdf.OpenImportFile("c:/PDFs/testfile.pdf", TPwdType.ptOpen, null);
    pdf.ImportPDFFile(1, 1.0, 1.0);

    // Three header / footer strings should be output in this example
    TPDFHdrFtr[]     hdr  = new TPDFHdrFtr[3];
    TPDFHeaderFooter init = new TPDFHeaderFooter();

    // Initialize the structures with default values
    pdf.InitHeaderFooter(ref init, ref hdr);

    init.InitColor     = 0;
    init.InitCodepage  = TCodepage.cpUnicode;
    init.InitEmbed     = true;
    init.InitFont      = "Arial";
    init.InitFontSize  = 8.0f;

    init.Margin.Left   = 50.0f;
    init.Margin.Right  = 50.0f;
    init.Margin.Top    = 30.0f;
    init.Margin.Bottom = 20.0f;

    hdr[0].IsHeader    = true;
    hdr[0].Text        = "This is a bates number: <<Bates#8#1>>";
    hdr[0].Position    = TTextAlign.taCenter;

    hdr[1].Color       = 255;
    hdr[1].CS          = TExtColorSpace.esDeviceRGB;
    hdr[1].Codepage    = TCodepage.cpUnicode;
    hdr[1].Font        = "Arial,Italic";
    hdr[1].FontSize    = 8.0f;
    hdr[1].Embed       = true;
    hdr[1].IsHeader    = true;
    hdr[1].Text        = "Page numbering: <<Page 1 of n>>";
    hdr[1].Position    = TTextAlign.taRight;

    hdr[2].IsHeader    = false;
    hdr[2].Text        = "Multiple formats: <<yyyy/mm/dd>> <<Bates#8#1>> <<Page 1 of n>>";
    hdr[2].Position    = TTextAlign.taLeft;

    pdf.AddHeaderFooter(ref init, ref hdr);

    pdf.CloseFile();
}
```

AddImage

Syntax:

```
LBOOL pdfAddImage(
const PPDF* IPDF,                  // Instance pointer
  TCompressionFilter Filter,   // See below
  TImageConversionFlags Flags, // See below
  struct TPDFImage* Image)     // Image structure

typedef enum
{
    cfFlate   = 0,   // PDF or TIFF output
    cfJPEG    = 1,   // PDF, JPEG, or TIFF output
    cfCCITT3  = 2,   // PDF or TIFF output -> B&W CCITT Fax G3 compression
    cfCCITT4  = 3,   // PDF or TIFF output -> B&W CCITT Fax G4 compression
    cfLZW     = 4,   // TIFF or GIF output -> Much faster than flate
    cfLZWBW   = 5,   // TIFF
    cfFlateBW = 6,   // TIFF, PNG, or BMP output -> Dithered black & white.
                     // The resulting image is compressed with Flate or left
                     // uncompressed if the output image format is a bitmap.
                     // If you want to use CCITT Fax 4 compression (TIFF
                     // only) then set the flag icUseCCITT4.
    cfJP2K    = 7    // PDF or JPEG2000 output
    cfNone    = 255, // TIFF output only
}TCompressionFilter;

typedef enum
{
   icNone      = 0, // Default
   icUseCCITT4 = 1  // Use CCITT Fax 4 for dithered images.
}TImageConversionFlags;
```

The function adds an image that was returned by the content parser to the current open image file. The output image must be opened with CreateImage() beforehand.

If the output format is TIFF, multiple images can be added to the file while each image can be compressed with a different compression filter. All other image formats support only one image. So, the image must be closed with CloseImage() after the image was added to the file in this case. Depending on the output format the compression filter cannot be freely chosen. For example, PNG is the native image format for Flate compression. This format supports Flate compression only. If an incompatible filter is set the function uses the default filter for the output format, e.g. Flate for PNG, JPEG for JPG, and so on.

The compression level can be adjusted with the function SetCompressionLevel(). The JPEG quality can be set with SetJPEGQuality().

The function is able to create up to 2 GB large TIFF files. As far as the images are compressed with the original filters it is usually always possible to add all images of a PDF file to one TIFF file.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddInkList

Syntax:

```
LBOOL pdfAddInkList(
   const PPDF* IPDF,          // Instance pointer
   UI32 InkAnnot,             // Handle of an ink annotation.
   struct TFltPoint* Points,  // Array of points
   UI32 NumPoints)            // Number of points in the array
```

The function adds an array of points or path to an Ink Annotation. The function must be called within an open page.

An ink annotation can contain an arbitrary number of paths. The points will be connected by curves to achieve a smooth transition between points. The coordinates are treated in current user space. Any transformation that was applied on the coordinate system will be taken into account.

If the number of points is zero then all paths will be deleted from the annotation.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddJavaScript

Syntax:

```
SI32 pdfAddJavaScript(
   const PPDF* IPDF,    // Instance pointer
   const char* Name,    // Name of the script (required)
   const char* Script)  // JavaScript (required)
```

The function adds a global JavaScript to the PDF file. The parameter *Name* must be a unique name to identify the script in the user interface:
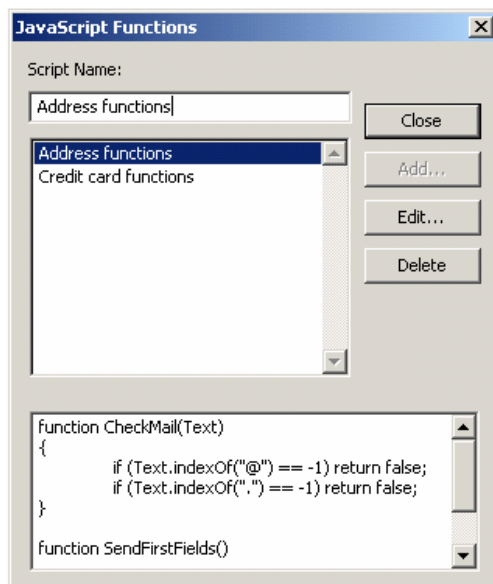
Note that the name is **NOT** used to identify a function inside the JavaScript. The parameter *Script* must be a JavaScript function or a set of JavaScript functions. DynaPDF does not check whether the script is valid. A JavaScript is always stored unchecked and unchanged to the PDF file. The JavaScript functions inside a script can be used by JavaScript actions to enable parameterized function calls (see also CreateJSAction()).

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Because JavaScript 1.2 is not Unicode compatible, Unicode encoded scripts are translated to a platform specific encoding prior to interpretation by the JavaScript engine. Note that there is no advantage when using Unicode encoded scripts in a PDF file. To avoid unnecessary problems JavaScripts should be stored in Ansi format.

Return values:

If the function succeeds the return value is the handle of the JavaScript, a value greater or equal zero. If the function fails the return value is a negative error code.

AddLayerToDisplTree

Syntax:

```
LGRP* pdfAddLayerToDisplTree(
    const PPDF* IPDF,  // Instance pointer
    LGRP* Parent,      // Parent node or NULL
    SI32 Layer,        // Layer (OCG) handle or -1
    const char* Title) // Group title or NULL
```

The function adds a layer or group to the layer display tree. The layer display tree is used in viewer applications to show the available layers in a PDF file and to interactively hide or unhide layers.

If we talk about a layer then we mean a handle of an OCG (Optional Content Group, see section Layers (Optional Content) for further information). OCMDs (Optional Content Membership Dictionaries) cannot be added to the tree, since OCMDs represent a visibility expression and no layer that can be displayed or hidden.

Note that the function CreateOCG() adds a layer automatically to the display tree if the parameter *DisplayInUI* was set to true. So, the parameter must be set to **false** if you want to build the tree manually.

The display tree consists of nested arrays. A sub array represents a new group which can optionally contain a title, one or more layers, or both. The order in which layers are added or how they are grouped can be freely choosen. However, it is usually best to create a layer tree with the same order and nesting levels as in the document. Layers which were not added to the tree will not be displayed in a viewer. This can be useful if a user should not be able to change the state of a layer interactively.

It is also possible to lock layers if necessary (see LockLayer() for further information).

As mentioned above, the parameters *Layer* and *Title* are both optional. This results in four possible combinations which produce different results.

If no layer is provided then the function creates a new group (a new sub array). However, the parent node cannot be empty (with one exception). At least one element must be added to an empty group before a new group can be added to it.

Take a look on the examples on the next page to understand how it works.

Return values:

If the function succeeds then the return value is a pointer of the newly created group. If no new group was created then the return value is the pointer of the parent group that was passed to the function. If the function fails then the return value is NULL.

Examples 1:

```
...
LGRP* root, *grp;
// Set the parameter DisplayInUI to false...
SI32 oc1 = pdfCreateOCG(PDF, "All", false, true, oiView);
SI32 oc2 = pdfCreateOCG(PDF, "Text", false, true, oiView);
SI32 oc3 = pdfCreateOCG(PDF, "Images", false, true, oiView);

root = pdfAddLayerToDisplTree(PDF, NULL, -1, NULL);
pdfAddLayerToDisplTree(PDF, root, oc1, "Just a title");
pdfAddLayerToDisplTree(PDF, root, oc2, NULL);
pdfAddLayerToDisplTree(PDF, root, oc3, NULL);
...
```

Result:
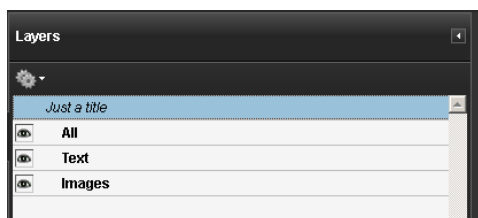
Example 2:

```
root = pdfAddLayerToDisplTree(PDF, NULL, -1, NULL);
grp  = pdfAddLayerToDisplTree(PDF, root, -1, "This is a group");
pdfAddLayerToDisplTree(PDF, grp, oc1, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc2, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc3, NULL);
```

Result:
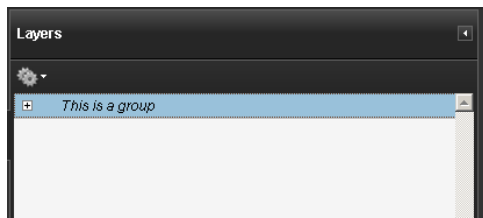
Example 3:

The following two versions are a bit confusing, I know...

```
root = pdfAddLayerToDisplTree(PDF, NULL, oc1, NULL);
grp  = pdfAddLayerToDisplTree(PDF, root, -1, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc2, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc3, NULL);
```
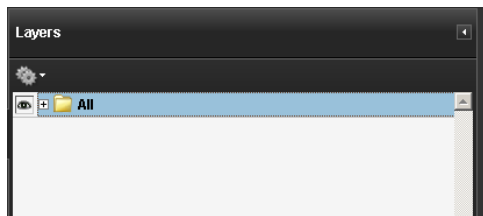
Result:

Example 4:

```
root = pdfAddLayerToDisplTree(PDF, NULL,oc1,"A layer group with a title");
grp  = pdfAddLayerToDisplTree(PDF, root, -1, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc2, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc3, NULL);
```

Result:

There is a special case that produces the same result in Acrobat but certain viewers have problems to display the layer tree depending on the used syntax. Compare the following two versions:

```
root = pdfAddLayerToDisplTree(PDF, NULL, -1, "This is a group");
grp  = pdfAddLayerToDisplTree(PDF, root, -1, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc1, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc2, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc3, NULL);

root = pdfAddLayerToDisplTree(PDF, NULL, -1, NULL);
```

```
grp  = pdfAddLayerToDisplTree(PDF, root, -1, "This is a group");
pdfAddLayerToDisplTree(PDF, grp, oc1, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc2, NULL);
pdfAddLayerToDisplTree(PDF, grp, oc3, NULL);
```

Normally, the second version shouldn't work since the parent node is empty but in practice both versions produce the same result. However, many third-party viewers have problems to display the layer tree if the title was set in the parent node (the first variant).

AddMaskImage

Syntax:

```
LBOOL pdfAddMaskImage(
   const PPDF* IPDF,    // Instance pointer
   UI32 BaseImage,      // Image to which the mask should be added
   const void* Buffer,  // Image buffer
   UI32 BufSize,        // Buffer size
   SI32 Stride,         // Scanline length in bytes
   UI32 BitsPerPixel,   // 1 and 8 bits per pixel are supported
   UI32 Width,          // Image width
   UI32 Height)         // Image height
```

The function adds an image mask to a base image. If *BitsPerPixel* is 1, the image is used as an image mask or stencil mask. The image mask specifies which areas on the page should be painted and which should be masked out (left unchanged).

If *BitsPerPixel* is 8, the image is used as a soft mask. A soft mask represents an alpha channel that ranges from 0 (transparent) through 255 (opaque).

The base and mask image need not to have the same width and height since both images are scaled into the same destination rectangle or parallelogram so that the boundaries coincide. This makes it possible to use a higher resolution mask with a lower resolution base image or vice versa.

If the image contains already a mask image then this mask will be replaced with the new one.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddObjectToLayer

Syntax:

```
LBOOL pdfAddObjectToLayer(
   const PPDF* IPDF,   // Instance pointer
   UI32 OCG,           // Must be a OCG or OCMD handle
   TOCObject ObjType,  // See below
   UI32 Handle)        // Object handle

typedef enum
{
   ooAnnotation,
   ooField,
   ooImage,
   ooTemplate
}TOCObject;
```

The function adds an object to an Optional Content Group (OCG) or Optional Content Membership dictionary (OCMD). See also CreateOCG() and CreateOCMD().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddOCGToAppEvent

Syntax:

```
LBOOL pdfAddOCGToAppEvent(
   const PPDF* IPDF,                  // Instance pointer
   UI32 Handle,                       // OCG handle
   TOCAppEvent Events,                // Bitmask, see below
   TOCGUsageCategory Categories)      // Bitmask, see below

typedef enum
{
   aeExport = 1,
   aePrint  = 2,
   aeView   = 4
}TOCAppEvent;
```

The categories refer to the members of a Content Usage dictionary. See SetOCGContUsage() for further information.

```
typedef enum
{
   oucNone     = 0,
   oucExport   = 1,
   oucLanguage = 2,
   oucPrint    = 4,
   oucUser     = 8,
   oucView     = 16,
   oucZoom     = 32
}TOCGUsageCategory;
```

The function adds an Optional Content Group (OCG) to an application event. PDF defines three events, in which the state of layers or OCGs can be dynamically changed depending on settings in

Content Usage Dictionaries, which are associated to OCGs. A Content Usage dictionary must be created or added to an OCG with SetOCGContUsage() before this function can be called.

The Content Usage dictionary can define multiple settings which control the visibility state, e.g. the wished Export and Print state. The parameter *Categories* is a bitmask, so that more than one category can be used in a specific event. The parameter *Event* is a bitmask too, so that the OCG can be added to multiple events in just one function call.

The function makes sure that only valid categories can be added to a specific event. If the Content Usage dictionary contains no settings for a specific category, then this category will be masked out from the parameter *Categories*. If no more category remains then the OCG will not be added to the event.

For example, if the Content Usage dictionary contains settings for exporting and printing, then it is possible to add the OCG to the Export and Print events in just one pass:

```
pdfAddOCGToAppEvent(pdf,
                    ocg,
                    TOCAppEvent(aeExport | aePrint),
                    TOCGUsageCategory(oucExport | oucPrint));
```

In the above example only the category oucExport will be added to the Export event, and oucPrint to the Print event. The category oucPrint is not meaningful for exporting and oucExport is not meaningful for printing. The invalid categories are automatically removed so that the above function call is allowed and valid.

Remarks:

Note that application events are not supported in all viewer applications. If a watermark should appear when printing, for example, then it is usually best to set the state of such a layer to visible, and to make it invisible in the View event. A viewer that supports application events will hide the layer at viewing time but still print it. A viewer that does not support application events will always display the layer.

If the initial state of the layer would be invisible, then it would also stay invisible at printing time if the viewer does not support application events but layers.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AddOutputIntent

Syntax:

```
SI32 pdfAddOutputIntent(
    const PPDF* IPDF,    // Instance pointer
    const char* ICCFile) // File path of the ICC profile
```

An Output Intent represents the destination color space for which a PDF file was created. Different profiles for DeviceGray, DeviceRGB and DeviceCMYK can be attached but this is not recommendet. Certain PDF standards like PDF/A or PDF/X prohibit the usage of multiple profiles since only one destination color space can exist. So, although PDF supports multiple profiles, please attach only one profile.

Depending on the application for which a PDF file was created the attached ICC profile is mostly a RGB or CMYK profile. RGB refers mainly to PDF files created for the web and CMYK profiles for printing.

BTW - You find a very detailed description about color management in PDF on our website...

ICC profiles are available for different input color spaces and for different output devices classes. PDF viewer applications support the device classes mntr, prtr, scnr, and spac.

The input color space can be DeviceGray, DeviceRGB, DeviceCMYK, or Lab. Other input color spaces are not supported. An ICC color profile is always required to create PDF/X compatible PDF files. PDF/A-1 compatible files must contain a rendering intent too if the document uses device dependent color spaces (DeviceGray, DeviceRGB, or DeviceCMYK).

ICC profiles are available in different versions. Which profile versions are allowed to use depends on the output PDF version:

PDF Version	ICC Specification Version	ICC Profile Version Number
PDF 1.3	3.3 or earlier	2.10
PDF 1.4	ICC.1:1998-09 and its addendum ICC.1A:1999-04	2.20
PDF 1.5	ICC.1:2001-12	4.00
PDF 1.6	ICC.1:2003-09	4.10
PDF 1.7	ICC.1:2004-10	4.20

Please note that neither PDF/A-1 nor PDF/X-1 and PDF/X-3 support ICC profile major versions higher than 2. The function checks the profile version only if the output version was set to a PDF/A or PDF/X compatible version.

The function scans the document always for a suitable profile which is maybe already available in an ICC-based color space. If a suitable profile can be found the function uses the already existing one so that a profile must not be embedded twice.

The profile should be embedded at the end of processing, that means directly before calling CloseFile() or CloseFileEx().

Return values:

If the function succeeds the return value is the handle of the Rendering Intent object, a value greater or equal zero. If the function fails the return value is a negative error code.

AddOutputIntentEx

Syntax:

```
SI32 pdfAddRenderingIntentEx(
   const PPDF* IPDF,    // Instance pointer
   const void* Buffer,  // File buffer of an ICC profile
   UI32 BufSize)        // Buffer size in bytes
```

The function adds an ICC profile to the PDF file like AddOutputIntent() but accepts a file buffer as input. See also AddOutputIntent().

Return values:

If the function succeeds the return value is the handle of the Rendering Intent object, a value greater or equal zero. If the function fails the return value is a negative error code.

AddPageLabel

Syntax:

```
SI32 pdfAddPageLabel(
   const PPDF* IPDF,        // Instance pointer
   UI32 StartRange,         // Page number where labelling starts
   TPageLabelFormat Format, // Label format
   const char* Prefix,      // Optional prefix
   SI32 FirstPageNum)       // Page number to be displayed in the label

typedef enum
{
   plfDecimalArabic,    // 1,2,3,4...
   plfUppercaseRoman,   // I,II,III,IV...
   plfLowercaseRoman,   // i,ii,iii,iv...
   plfUppercaseLetters, // A,B,C,D...
   plfLowercaseLetters, // a,b,c,d...
   plfNone
}TPageLabelFormat;
```

The function creates a page label object. The parameter *StartRange* must be the page number where labelling should start. The first page is denoted by 1. The parameters *Prefix* and *FirstPageNum* are both optional. *FirstPageNum* represents the numeric portion of the first page label in the range. This value is incremented on subsequent pages in the range. The value must be greater zero, otherwise it will be ignored. Numbering of the range starts at one in this case.

FirstPage should normally be set to the same value as *StartPage* to achieve a continous page numbering. Although it is possible to number each range separately, this can be very confusing for users since the logical and physical page order is then different.

A document can contain multiple page label objects. Each of them represents a labelling range which is a series of consecutive pages using the same numbering system.

Pages within a range are numbered sequentially in ascending order. A page's label consists of a numeric portion based on its position within its labelling range, optionally preceded by a label prefix denoting the range itself. For example, the pages in an appendix might be labelled with decimal numeric portions prefixed with the string A; the resulting page labels would be A-1, A-2, and so on.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is the page label handle, a value greater or equal zero. If the function fails the return value is a negative error code.

AddRasImage (Rendering Engine)

Syntax:

```
LBOOL pdfAddRasImage(
    const PPDF* IPDF,           // PDF Instance pointer
    IRAS* RasPtr,               // Instance pointer of the rasterizer
    TCompressionFilter Filter)  // Compression filter for TIFF output

typedef enum TCompressionFilter
{
    cfFlate               = 0x00000000, // TIFF output
    cfJPEG                = 0x00000001, // JPEG, or TIFF output
    cfCCITT3              = 0x00000002, // B&W CCITT Fax G3 compression
    cfCCITT4              = 0x00000003, // B&W CCITT Fax G4 compression -> higher compression ratio
    cfLZW                 = 0x00000004, // Very fast but less compression ratios than flate
    cfLZWBW               = 0x00000005, // TIFF -> Floyd Steinberg dithering
    cfFlateBW             = 0x00000006, // TIFF, PNG, or BMP output -> Floyd Steinberg dithering
    cfJP2K                = 0x00000007, // JPEG2000 output
    // These flags can be combined with the filters cfFlate, cfCCITT3, cfCCITT4, and LZW.
    cfDitherFloydSteinberg = 0x00001000, // See 1 bit image output below
    cfConvGrayToOtsu       = 0x00002000, // See 1 bit image output below
    cfOrderedDithering     = 0x00004000, // See 1 bit image output below
    cfIgnoreICCProfile     = 0x00008000  // Meaningful only if color management is enabled. If set,
                                         // the device ICC profile is not embedded in the image.
}TCompressionFilter;
```

The function adds an image of the rasterizer to the current open image file. The output image must be opened with CreateImage() beforehand.

The function performs a color conversion if the output image format requires another component order. For example, if the pixel format of the rasterizer is RGB and if the output image format is BMP then the component order must be changed to BGR because Windows Bitmaps require the

component order BGR. This conversion must be considered if the image buffer should be passed to another function.

Windows Bitmaps require also double word aligned scanlines while all other image formats support byte aligned the scanlines. AddRasImage() adds required fill bytes automatically to achieve the line alignment. However, to increase conversion speed the image buffer should consider the scaline alignment of the output image format if possible.

TIFF is the one and only multi-page image format. All other image formats support exactly one image. The image must be closed with CloseImage() when finish.

1 bit image output

If the pixel format of the rasterizer was set pxfGray, it is possible convert the image buffer to 1 bit black & white with ordered dithering, Floyd Steinberg dithering, or with the Otsu filter. The Otsu filter is a special filter that avoids anti-aliasing artifacts. It produces true black & white output and does not try to simulate gray shades by dithering areas of an image. This filter is optimal if the resulting image should be passed to an OCR engine.

There are two ways to apply the Floay Steinberg dithering algorithm: either combine the flag cfDitherFloydSteinberg with the compression filter you want to use:

```
TCompressionFilter(cfFlate | cfDitherFloydSteinberg)
```

or use one of the filters cfCCITT3/4, cfLZWBW or cfFlateBW. These constants were already defined in DynaPDF 3.0. However, in future we use separate flags for such things because it is more flexible.

The Otsu filter or ordered dithering can be enabled in the very same way:

```
    TCompressionFilter(cfFlate | cfConvGrayToOtsu)
or
    TCompressionFilter(cfFlate | cfOrderedDithering)
```

Compression filters

The compression filter is considered only for TIFF output because all other image formats support only one specific filter. The compression filter must be compatible to the pixel format. For example, CCITT Fax compression is compatible to 1 bit images, it is not supported for gray or color images.

The function is able to create up to 4 GB large TIFF image files. Depending on the used compression filter and compression ratio this is mostly enough to create TIFF images with up to 8000 RGB pages in a resolution of 300 DPI. However, the maximum number of pages depends on the resolution, compression ratio and pixel format.

The JPEG Quality and compression level can be adjusted with SetJPEGQuality() and SetCompressionLevel() before calling the function.

Embedding ICC Profiles

The function embeds by default the device profile in the image file if color management is enabled and if the image format supports embedded ICC profiles. At time of publication JPEG 2000, PNG, and the TIFF encoder support embedded ICC profiles.

However, the JPEG 2000 encoder has only limited support for embedded ICC profiles. A profile is embedded only if loss-less encoding is used. Loss-less encoding is used if the JPEG quality is set to 0 or 100 or -1000.

It is also possible to disable embedding of ICC profiles with the flag *cfIgnoreICCProfile*.

Return values:

When the function succeeds the return value is 1. When the function fails the return value is 0.

AddRenderingIntent (obsolete)

Syntax:

```
SI32 pdfAddRenderingIntent(
   const PPDF* IPDF,    // Instance pointer
   const char* ICCFile) // File path of the ICC profile
```

This function was incorrectly named. The correct name is AddOutputIntent(). Please use the correctly named version instead.

AddRenderingIntentEx (obsolete)

Syntax:

```
SI32 pdfAddRenderingIntentEx(
  const PPDF* IPDF,   // Instance pointer
  const void* Buffer, // File buffer of an ICC profile
  UI32 BufSize)       // Buffer size in bytes
```

This function was incorrectly named. The correct name is AddOutputIntentEx(). Please use the correctly named version instead.

AddValToChoiceField

Syntax:

```
SI32 pdfAddValToChoiceField(
   const PPDF* IPDF,     // Instance pointer
   UI32 Field,           // Handle of a combo box or list box
   const char* ExpValue, // Export value (can be NULL)
   const char* Value,    // Visible choice value (required)
   LBOOL Selected)       // Select this values?
```

This function adds a value to a choice field. The parameter *Field* must be a handle of a combo box or list box. The parameter *ExpValue* defines the export value that will be submitted to a web server if

the value is selected in Adobe's Acrobat. The export value can be an empty string (NULL) and it is possible to use one export value for multiple values.

The parameter *Value* is the visible text that can be selected inside the combo box or list box. Value must be a unique string inside the combo box or list box values. *Value* is required; it must not be an empty string.

If the parameter *Selected* is true the value appears as selected value in Adobe's Acrobat. List boxes support multiple selected values depending on whether the flag ffMultiSelect was set or not (see SetFieldFlags() and CreateListBox() for further information). If the flag ffMultiSelect is not set, a selected value disables all others (default).

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Because Unicode is not supported by interactive form fields, it is recommended that Unicode values contain characters of the actual used code page only. At time of publication CJK strings are not supported.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Append

Syntax:

```
SI32 pdfAppend(
    const PPDF* IPDF) // Instance pointer
```

This function appends an empty page to the document. The default page format for new created pages is DIN A4. However, when changing the page format, the next page is created in the same format as the previous one. The CropBox is also set to the values of the previous page. The other bounding boxes such as ArtBox, BleedBox and so on must be set manually for each page.

The page format can be changed with the function SetPageFormat() if a predefined format should be used, or manually with SetBBox(). SetBBox() must also be used to set a user defined bounding box (ArtBox, BleedBox, CropBox, TrimBox).

The meaning of the bounding boxes are described under the property Get/SetBBox().

The graphics state of a new page is initialzed with the following default values:

Parameter	Data type	Initial Value
CharSpacing	float	0.0f
Clipping Path	Vector array	Crop box or Media box
DashPattern	double*	NULL (Solid line)
DashPhase	UI32	0
FillColor	BYTE[32]	Black
FillColorSpace	TExtColorSpace, IColorSpace*	esDeviceRGB, NULL
FillPattern	IPattern*	NULL
Font	IFont*	NULL
Leading	float	0.0f
LineCapStyle	TLineCapStyle	csButtCap
LineJoinStyle	TLineJoinStyle	jsMiterJoin
LineWidth	float	1.0f
Matrix	TCTM	{1, 0, 0, 1, 0, 0}
MiterLimit	float	10.0f
StrokeColor	BYTE[32]	Black
StrokeColorSpace	TExtColorSpace, IColorSpace*	esDeviceRGB, NULL
StrokePattern	IPattern*	NULL
TextDrawMode	TDrawMode	dmNormal
TextScale	float	100.0f
WordSpacing	float	0.0f

Return values:

If the function succeeds the return value is 1. If the function fails the return value is zero.

ApplyAppEvent

Syntax:

```
LBOOL pdfApplyAppEvent(
    const PPDF* IPDF,   // Instance pointer
    TOCAppEvent Event,  // Event that should be applied
    LBOOL SaveResult)   // If true, the new state will be stored in the file

typedef enum
{
    aeExport = 1,
    aePrint  = 2,
    aeView   = 4
}TOCAppEvent;
```

The visibility state of optional content groups (OCGs) can be dynamically changed in application events like Export, View, or Print. DynaPDF loads the view state when rendering PDF pages by default.

The function can be used to apply the visibility state of another event if necessary. The result can optionally be stored in the PDF file but note that this has only an effect in PDF viewers which support layers but no application events since the events will not be deleted from the PDF file. It is also possible to delete the events with DeleteAppEvents().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ApplyPattern

Syntax:

```
SI32 pdfApplyPattern(
    const PPDF * IPDF,     // Instance pointer
    SI32 PattHandle,       // Handle of the pattern
    TColorMode ColorMode,  // Color mode (fill or stroke color or both)
    UI32 Color)            // uncolored patterns only

typefed enum
{
    cmFill,       // use the pattern as fill color
    cmStroke,     // use the pattern as stroke color
    cmFillStroke  // use the pattern as fill and stroke color
}TColorMode;
```

The function sets a tiling pattern as current fill, stroke or fill and stroke color. Two types of patterns are supported; colored tiling patterns which contain their own color information and uncolored tiling patterns which can be used with a user defined color.

The parameter *PattHandle* must be a valid pattern handle that was returned by BeginPattern() or CreateStdPattern(). The parameter *Color* is ignored for colored tiling patterns. Uncolored tiling

patterns are applied like a mask in the user defined color. The color must be defined in the current color space.

To disable a pattern just change the current fill or stroke color with SetFillColor(), SetStrokeColor() or SetColors().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (Delphi):

```delphi
implementation
{$R *.dfm}
uses dynapdf; // include the unit dynapdf.pas
// First we define our callback function that is called if an
// error occurred. Note: The calling convention is stdcall!
function ErrProc(const Data: Pointer; ErrCode: Integer; const ErrMessage:
PAnsiChar; ErrType: Integer): Integer; stdcall;
var s: String;
begin
   s := Format('%s'#13'Abort processing?', [ErrMessage]);
   if MessageDlg(s, mtError, [mbYes, mbNo], 0) = mrYes then
      Result := -1 // break processing
   else
      Result := 0; // try to continue
end;

procedure TForm1.Button1Click(Sender: TObject);
var pdf: TPDF; pat: Integer; cw, h: Single;
begin
  pdf := nil;
  try
    pdf := TPDF.Create;
    // Set the error callback first. No need to check return values.
    pdf.SetOnErrorProc(nil, @ErrProc);
    pdf.SetDocInfoA(diAuthor, 'Jens Boschulte');
    pdf.SetDocInfoA(diCreator, 'Delphi sample project');
    pdf.SetDocInfoA(diSubject, 'How to use patterns');
    pdf.SetDocInfoA(diTitle, 'Vector graphics');
    pdf.SetViewerPreferences(vpDisplayDocTitle, avNone);
    pdf.CreateNewPDFA('c:\dout.pdf');

    pdf.Append;
    // create a colored tiling pattern
    pat := pdf.BeginPattern(ptColored, ttFastConstSpacing, 20, 20);
    pdf.SetFontA('ZapfDingbats', fsNone, 12, false, cp1252);
    h := pdf.GetAscent;
    cw := pdf. GetTextWidthA(#171);
    pdf.SetFillColor(clRed);
    pdf.WriteTextA(0, 0, #171);
```

```
    pdf.SetFillColor(clGreen);
    pdf.WriteText(cw +2, 0, #170);
    pdf.SetFillColor(clBlue);
    pdf.WriteTextA(0, 20 - h, #169);
    pdf.SetFillColor(clBlack);
    pdf.WriteTextA(cw +2, 20 - h, #168);
    pdf.EndPattern;

    pdf.ApplyPattern(pat, cmFill, 0);
    pdf.SetPageCoords(pcTopDown);
    pdf.DrawCircle(100, 160, 50, fmFillStroke);
    pdf.DrawCircle(160, 60, 50, fmFillStroke);
    pdf.DrawCircle(220, 160, 50, fmFillStroke);
    pdf.Triangle(100, 160, 160, 60, 220, 160, fmFillStroke);
    pdf.SetPageCoords(pcBottomUp);

    // create an uncolored tiling pattern
    pat := pdf.BeginPattern(ptUnColored, ttFastConstSpacing, 20, 20);
    pdf.SetFontA('ZapfDingbats', fsNone, 12, false, cp1252);
    h := pdf.GetAscent;
    cw := pdf. GetTextWidthA(#171);
    pdf.WriteTextA(0, 0, #171);
    pdf.WriteTextA(cw +2, 0, #170);
    pdf.WriteTextA(0, 20 - h, #169);
    pdf.WriteTextA(cw +2, 20 - h, #168);
    pdf.EndPattern;

    pdf.SetPageCoords(pcTopDown);
    // use the pattern as fill color with different colors
    pdf.ApplyPattern(pat, cmFill, clGreen);
    pdf.DrawCircle(350, 160, 50, fmFillStroke);
    pdf.ApplyPattern(pat, cmFill, clBlue);
    pdf.DrawCircle(410, 60, 50, fmFillStroke);
    pdf.ApplyPattern(pat, cmFill, clRed);
    pdf.DrawCircle(470, 160, 50, fmFillStroke);
    pdf.ApplyPattern(pat, cmFill, clMaroon);
    pdf.Triangle(350, 160, 410, 60, 470, 160, fmFillStroke);
    pdf.EndPage;

    pdf.CloseFile;
  except
    on E: Exception do MessageDlg(E.Message, mtError, [mbOK], 0);
  end;
  if pdf <> nil then pdf.Free;
end;
```

Output:
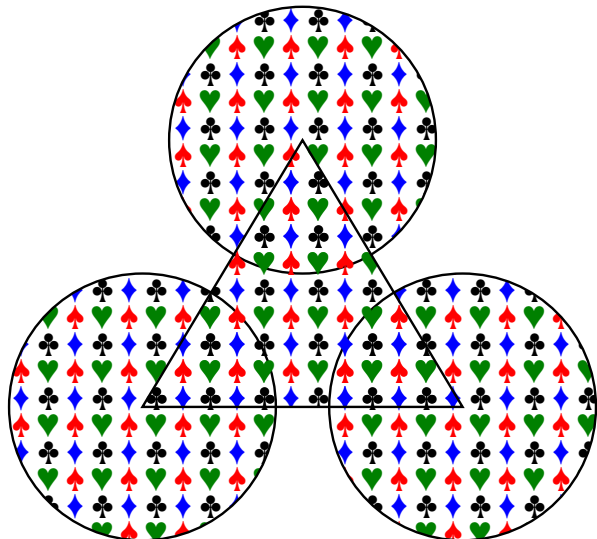
Colored Tiling Pattern   **_Uncolored Tiling Pattern_**
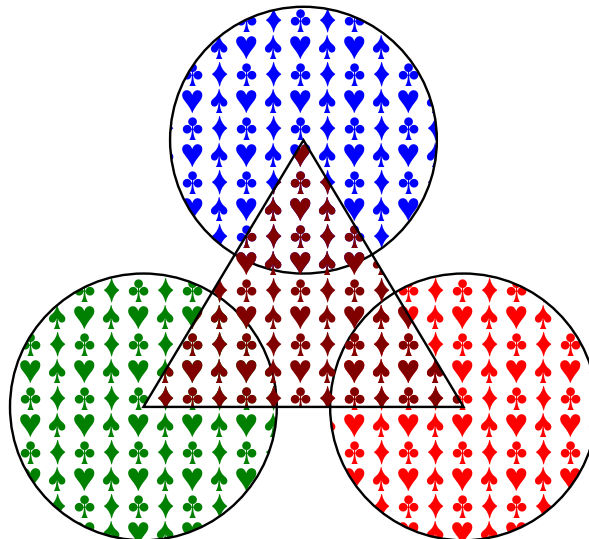
ApplyShading

Syntax:

```
SI32 pdfApplyShading(
   const PPDF* IPDF, // Instance pointer
   SI32 ShadHandle)  // Handle of the shading
```

This function applies a shading to the current clipping path, or if no clipping is active, to the entire page. See also CreateAxialShading(), CreateRadialShading().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (C):

```c
// First we declare our error callback function
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}

int main(int argc, char* argv[])
{
   SI32 sh;
   void* pdf = pdfNewPDF();
   if (!pdf) return 2; // Out of memory?

   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfSetDocInfo(pdf, diAuthor, "Jens Boschulte");
   pdfSetDocInfo(pdf, diCreator, "C sample project");
   pdfSetDocInfo(pdf, diSubject, "How to use shadings");
   pdfSetDocInfo(pdf, diTitle, "Shadings");
   // we use top-down coordinates
   pdfSetPageCoords(pdf, pcTopDown);

   pdfCreateNewPDF(pdf, "c:/cppout.pdf");

   pdfAppend(pdf);

   pdfSaveGraphicState(pdf);
   pdfBeginClipPath(pdf);
   pdfRectangle(pdf, 50, 50, 200, 30, fmNoFill);
   pdfClipPath(pdf, cmWinding, fmFill);
   sh = pdfCreateAxialShading(pdf, 50, 0, 250, 0, 1.5, PDF_WHITE,
      PDF_BLUE, false, false);
   pdfApplyShading(pdf, sh);
   pdfRestoreGraphicState(pdf);
```

```
    pdfSaveGraphicState(pdf);
    pdfBeginClipPath(pdf);
    pdfRectangle(pdf, 50, 90, 200, 30, fmNoFill);
    pdfClipPath(pdf, cmWinding, fmFill);

    sh = pdfCreateAxialShading(pdf, 0, 120, 0, 90, 1.5, PDF_WHITE,
        PDF_BLUE, false, false);
    pdfApplyShading(pdf, sh);
    pdfRestoreGraphicState(pdf);

    // The radial shading can be drawn outside of a clipping path
    sh = pdfCreateRadialShading(pdf, 165, 70, 5, 150, 85, 35, 0.7,
        PDF_BLUE, PDF_WHITE, true, false);
    pdfApplyShading(pdf, sh);

    pdfEndPage(pdf);
    pdfCloseFile(pdf);
    pdfDeletePDF(pdf);
}
```

Output:
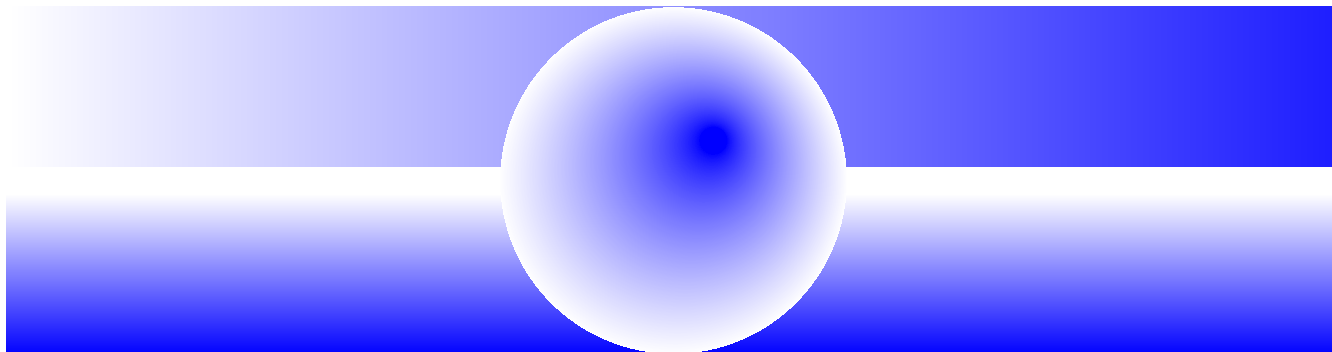
AssociateEmbFile

Syntax :

```
LBOOL pdfAssociateEmbFile(
    const PPDF* IPDF,               // Instance pointer
    TAFDestObject DestObject,       // Destination object type (see below)
    SI32 DestHandle,                // Destination object handle or -1
    TAFRelationship Relationship,   // see below
    UI32 EmbFile)                   // Handle of an embedded file

typedef enum
{
    arAssociated,
    arData,
    arSource,
    arSupplement,
    arAlternative, // This key must be used for ZUGFeRD compatible invoices
    arFormData
}TAFRelationship;

typedef enum
{
    adAnnotation,
    adCatalog,     // The documents catalog is the root object
    adField,
    adImage,
    adPage,
    adTemplate
}TAFDestObject;
```

The function associates an embedded file with a PDF object. The parameter *DestHandle* must be a valid handle of a PDF object. If destination object type is a page, then the page number must be used as handle. The first page is denoted by one. If the destination object is the documents catalog then the parameter *DestHandle* is ignored. Set the parameter to zero or -1 in this case.

The parameter EmbFile must be a valid handle of an embedded file. See AttachFile() or AttachFileEx() for further information.

Remarks:

Associated files are supported since PDF 2.0 and in PDF/A 3 files. In PDF/A 3 files all embedded files must be associated with a PDF object.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

AttachFile

Syntax:

```
SI32 pdfAttachFile(
  const PPDF* IPDF,        // Instance pointer
  const char* FilePath,    // File that should be attached
  const char* Description, // Optional description
  LBOOL Compress)          // If true, the file will be compressed
```

The function attaches a file to the document. The description is optional, the parameter can be set to NULL. The embedded file is compressed if the parameter *Compress* is set to true. Otherwise it is left uncompressed. It is not always useful to compress embedded files especially if the file is already compressed, e.g. Zip files or already compressed image formats require no further compression. In the worst case the compressed file becomes larger as the uncompressed version. However, text files and most document formats should be compressed to reduce the file size.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Unicode paths are converted to UTF-8 on non-Windows operating systems.

Please note that embedded files must be associated with a PDF object in PDF/A 3 files. See AssociateEmbFile() for further information.

Return values:

If the function succeeds the return value is the handle of the embedded file, a value greater or equal zero. If the function fails the return value is a negative error code.

AttachFileEx

Syntax:

```
SI32 pdfAttachFileEx(
  const PPDF* IPDF,        // Instance pointer
  const void* Buffer,      // File buffer
  UI32 BufSize,            // Buffer size
  const char* FileName,    // File name to display in a viewer
  const char* Description, // Optional description
  LBOOL Compress)          // If true, the file will be compressed
```

The function attaches a file to the document in the same way as AttachFile() but accepts a file buffer instead of a file path. The description is optional, the parameter can be set to NULL. The embedded file is compressed if the parameter *Compress* is set to true. Otherwise it is left uncompressed. It is not always useful to compress embedded files especially if the file is already compressed, e.g. Zip files or already compressed image formats require no further compression. In the worst case the compressed files becomes larger as the uncompressed version. However, text files and most document formats should be compressed to reduce the file size.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Please note that embedded files must be associated with a PDF object in PDF/A 3 files. See AssociateEmbFile() for further information.

Return values:

If the function succeeds the return value is the handle of the embedded file, a value greater or equal zero. If the function fails the return value is a negative error code.

AttachImageBuffer (Rendering Engine)

Syntax:

```
LBOOL rasAttachImageBuffer(
   IRAS*  RasPtr,       // Instance pointer of the rasterizer
   BYTE** Rows,         // Pointer to a row buffer or alternatively
   BYTE*  Buffer,       // Pointer to an image buffer that was allocated
                        // as one large buffer.
   UI32   Width,        // Width in Pixel
   UI32   Height,       // Height in Pixel
   SI32   ScanlineLen)  // Scanline length in bytes
```

The function attaches a new image buffer to the rasterizer. The function supports two different types or pointers to the image buffer depending on how the buffer was allocated.

When the image buffer was allocated as one large block then set the parameter *Rows* to NULL and pass the buffer to the parameter *Buffer*. If the scanlines were allocated separately or in blocks then use the parameter *Rows* instead. One of these parameters must be set but never both. The function checks first whether the parameter *Rows* is present.

The rasterizer uses internally an array of scanlines to achieve optimal processing speed. Therefore, the function must maybe reallocate the internal pointer array when the new image height is larger than the previous one. When there is no sufficient memory to allocate the row pointer array then the function will fail and returns false. The one and only reason why this function can fail is out of memory.

The return value must not be ignored since the rasterizer will cause an access violation if you try to raster into a non-existant image buffer.

The rasterizer requires no special alignment of the scanlines, the scanlines can be byte aligned.

However, the scaline length can be longer than necessary, e.g. to achieve aligment requirements of certain image formats like Bitmaps. A negative value of the scanline length mirrors the image vertically.

Remarks:

This function is not included in the Visual Basic 6 interface.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0. The one and only reason why this function can fail is out of memory.

AutoTemplate

Syntax:

```
SI32 pdfAutoTemplate(
    const PPDF* IPDF, // Instance pointer
    UI32 Templ,       // Template handle
    double PosX,      // X-Coordinate
    double PosY,      // Y-Coordinate
    double Width,     // Scaled with
    double Height)    // Scaled height
```

Templates can be used on multiple pages, e.g. as background, custom logos and so on. To insert a template automatically on newly created pages, use the function AutoTemplate().

The parameter *Templ* must be a valid template handle. The width and height can be calculated as follows:

- If *Width* or *Height* is **-1** the function uses the original width or height from the template.

- If *Width* **or** *Height* is **0**, the missing value is calculated in relation to the given value of *Height* or *Width* to preserve the template's aspect ratio. The resulting output is a template with exact proportions relative to its original size.

- If *Width* **and** *Height* is **0**, the original size is used (same effect as -1).

- A negative value of *Width* or *Height* mirrors the template on the x- and or y-axis.

Remarks:

The templates are inserted on new pages by using the function PlaceTemplate().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

BeginClipPath (Obsolete)

Syntax:

```
SI32 pdfBeginClipPath(
    const PPDF* IPDF) // Instance pointer
```

It is not longer required to call this function before a clipping path can be created. The function is included in DynaPDF for compatibility reasons only. Do not longer use it, it executes nothing. See also ClipPath(), Path Construction and Painting.

BeginContinueText

Syntax:

```
SI32 pdfBeginContinueText(
    const PPDF* IPDF, // Instance pointer
    double PosX,      // X-Coordinate of the first text line
    double PosY)      // Y-Coordinate of the first text line
```

This function prepares the output of multiple text lines with AddContinueText(). The distance between two text lines can be adjusted with SetLeading(). If no leading was defined it will be set to the font size. Note that the function changes the leading of the graphics state if it was not defined beforehand. Subsequent calls use then the leading that was set in a prior call. If the font or font size will be changed it is strongly recommended to set the leading to a correct value. The default leading is the font size. The leading, font, character spacing, word spacing, text scaling, or the text color can be changed before calling AddContinueText() the next time.

The point *PosX, PosY* defines the baseline of the first text line if the coordinate system is bottom-up, otherwise the top-left corner of the text's bounding box. The origin can be changed with the function SetFontOrigin().

If the font uses horizontal writing mode the text cursor moves from top to bottom. In vertical writing mode the text cursor moves from left to right.

Remarks:

A font must be set before this function can be used (see also SetFont(), or SetCIDFont()).

In DynaPDF versions prior 2.0 it was required to finish the text block with EndContinueText(). Because the function uses now internally WriteText() to output the text, it is no longer required to finish the text block with this function. EndContinueText() does nothing in DynaPDF 2.0 or higher.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (Delphi):

```
uses dynapdf;
// First we declare an error callback function
```

```pascal
function ErrProc(const Data: Pointer; ErrCode: Integer; const ErrMessage:
PAnsiChar; ErrType: Integer): Integer; stdcall;
var s: String;
begin
   s := Format('%s'#13'Abort processing?', [ErrMessage]);
   if MessageDlg(s, mtError, [mbYes, mbNo], 0) = mrYes then
      Result := -1 // break processing
   else Result := 0; // try to continue
end;
procedure TForm1.Button1Click(Sender: TObject);
var pdf: TPDF;
begin
  try
    pdf := TPDF.Create;
    pdf.SetOnErrorProc(nil, @ErrProc);
    pdf.SetDocInfoA(diAuthor, 'Jens Boschulte');
    pdf.SetDocInfoA(diCreator, 'Delphi sample project');
    pdf.SetDocInfoA(diSubject, 'How to use ContinueText');
    pdf.SetDocInfoA(diTitle, 'ContinueText');
    pdf.SetViewerPreferences(vpDisplayDocTitle, avNone);
    pdf.CreateNewPDFA('c:\dout.pdf');
    pdf.Append;
    pdf.SetFontA('Arial', fsNone, 12, true);
    pdf.ChangeFontStyle(fsUnderlined);
    pdf.BeginContinueText(50, 780);
    pdf.AddContinueTextA('Underlined text...');
    pdf.ChangeFontStyle(tsNone);
    pdf.AddContinueTextA('Normal text!');
    pdf.ChangeFontStyle(fsStriked);
    pdf.AddContinueTextA('Strikeout text!');
    pdf.EndContinueText;

    pdf.EndPage;
    pdf.CloseFile;
  except
    on E: Exception do MessageDlg(E.Message, mtError, [mbOK], 0);
  end;
  pdf.Free;
end;
```

Output:

Underlined text…

Normal text!

~~Strikeout text!~~

BeginLayer

Syntax:

```
LBOOL pdfBeginLayer(
  const PPDF* IPDF, // Instance pointer
  UI32 OCG)         // OCG or OCMD handle
```

The function opens a layer in the current open page or template. The layer must be closed with EndLayer() when finish. The parameter *OCG* must be valid handle of an Optional Content Group (OCG) or Optional Content Membership Dictionary (OCMD) (see CreateOCG() and CreateOCMD() for further information). The OCG or OCMD controls the visibility of the contents that is included in the layer.

BeginLayer() / EndLayer() calls connect normal page contents, such as text, images, and vector graphics with a layer but no interactive objects like annotations or form fields. Annotations and Form Fields can be added to a layer with AddObjectToLayer().

It is not required to draw the entire contents of a layer in one pass. The page contents can be drawn as usual and delimited into BeginLayer() / EndLayer() calls as necessary. Only contents that is delimited into BeginLayer() / EndLayer() calls becomes part of a layer.

Layers can be nested by calling BeginLayer() more than one time with different OCGs or OCMDs. The visibility of an inner layer depends then also on the visibility of the out ones (see example below). If an outer layer is invisible then all inner layers are invisible too, regardless of the visibility settings of the inner layers.

Remarks:

A layer is **not** part of the graphics state. Neither BeginLayer() nor EndLayer() change any parameter of the graphics state. BeginLayer() must not be called within an open path.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (C++):

```
...
pdfCreateNewPDF(pdf, "test_layer.pdf");

// We use three layers in this example
SI32 oc1 = pdfCreateOCG(pdf, "Anything", true, true, oiView);
SI32 oc2 = pdfCreateOCG(pdf, "Text and Annotations", true, true, oiView);
SI32 oc3 = pdfCreateOCG(pdf, "Images", true, true, oiView);

pdfSetPageCoords(pdf, pcTopDown);

pdfAppend(pdf);
// The main layer controls the visibility of all three layers in this
// example.
```

```
pdfBeginLayer(pdf, oc1);

   pdfBeginLayer(pdf, oc2);
      pdfSetFont(pdf, "Arial", fsRegular, 12.0, true, cp1252);
      char someText[] = "Some text with a link!!!";
      pdfWriteText(pdf, 50.0, 50.0, someText);
      double tw = pdfGetTextWidth(pdf, someText);
      // To reflect the same nesting as the text layer we must
      // use an OCMD for the annotation because the visibility of the
      // layer oc2 depends on oc1 at this position.
      pdfSetBorderStyle(pdf, bsUnderline);
      pdfSetStrokeColor(pdf, PDF_BLUE);
      SI32   annot = pdfWebLink(pdf, 50, 51, tw, 12, "www.dynaforms.com");
      UI32 ocgs[2] = {oc1, oc2};
      SI32    ocmd = pdfCreateOCMD(pdf, ovAllOn, ocgs, 2);
      pdfAddObjectToLayer(pdf, ocmd, ooAnnotation, annot);
   pdfEndLayer(pdf);

   pdfBeginLayer(pdf, oc3);
      InsertImageEx(pdf, 50.0, 70.0, 300.0, 200.0, "c:/Imgs/test.tif", 1);
   pdfEndLayer(pdf);

pdfEndLayer(pdf);

pdfWriteText(pdf, 50.0, 300.0, "This text is not part of a layer!");

pdfEndPage(pdf);

pdfCloseFile(pdf);
...
```

BeginPageTemplate

Syntax:

```
LBOOL pdfBeginPageTemplate(
    const PPDF* IPDF,        // Instance pointer
    const char* Name,        // Template name
    LBOOL UseAutoTemplates)  // See description
```

The function creates a page template. A page template is an invisible page that can be added to the document via Javascript.

Page templates are mainly used in Interactive Forms to dynamically add additional pages to the document. A page template is like a normal page. You can add form fields and annotations to it but no article beads (see AddArticle()) since a page template is not part of the document's page array.

The parameter *Name* is required, it must not be NULL or an empty string. The name can be used to reference the template via Javascript.

The parameter *UseAutoTemplates* specifies whether the list of auto templates should be drawn just as if you would call Append() (see AutoTemplate() for further information).

The page template must be closed like a normal page with EndPage() when finish.

Remarks:

Page templates cannot be added to the document with Adobe's Reader. This functionality requires a full version of Adobe's Acrobat.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

BeginPattern

Syntax:

```
SI32 pdfBeginPattern(
    const PPDF* IPDF,         // Instance pointer
    TPatternType PatternType, // Pattern type
    TTilingType TilingType,   // Tiling type
    double Width,             // Cell width
    double Height)            // Cell height
typedef enum
{
    ptColored,
    ptUnColored
}TPatternType;

typedef enum
{
    ttConstSpacing,
    ttNoDistortion,
    ttFastConstSpacing
}TTilingType;
```

This function creates a new tiling pattern. The graphics state is initialized to its default state when opening a pattern. See Append() for further information. The parameters *Width* and *Height* define the bounding box of the pattern cell.

Colored Tiling Patterns

Colored tiling patterns can be created like a normal PDF page or template. The only difference is that the pattern must be enclosed in BeginPattern() and EndPattern() calls. The objects inside the pattern are drawn in the usual way. After all objects are drawn the pattern must be finished with EndPattern().

Uncolored Tiling Patterns

Uncolored patterns are also painted in the usual way. That means all object types can be inserted such as text, vector graphics or images. However, an uncolored tiling pattern must **NOT** include any color information. Because of this inserting a color image into an uncolored tiling pattern is not

allowed, only 1 bit images can be used. This type of pattern can be painted in a user defined color that can be applied with ApplyPattern().

The color information of a colored tiling pattern is fixed, it can not be changed.

A pattern can be used like a color. It is not recommended to save the graphics state before using a pattern. A pattern is active as long as the used color is changed by SetFillColor() or SetStrokeColor(). See ApplyPattern() for an example application.

The current graphics state is saved entirely before the pattern will be entered. This graphics state is restored when the pattern is closed with EndPattern(). That means, the current font, line width, fill color and so on are all restored to its values before entering the pattern.

Remarks:

Patterns are invisible as long as they are not applied by the function ApplyPattern(). The initial graphic state defines a zero line width in Acrobat 5. If a pattern should contain any line art or stroked paths, the line width must be explicitly set before an object is drawn. If the line width is not set, the results can vary between different Acrobat versions. In general a pattern should not contain very complex objects to improve graphics speed.

Return values:

If the function succeeds, the return value is a pattern handle, a value greater or equal zero. If the function fails the return value is a negative error code.

Pattern types:
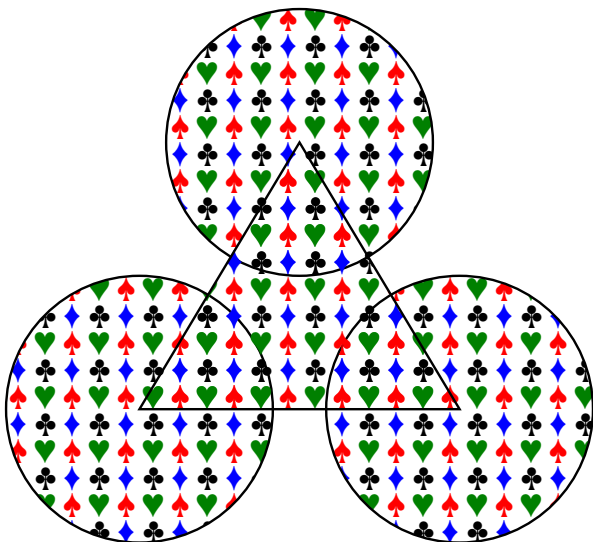
Colored Tiling Pattern   *Uncolored Tiling Pattern*
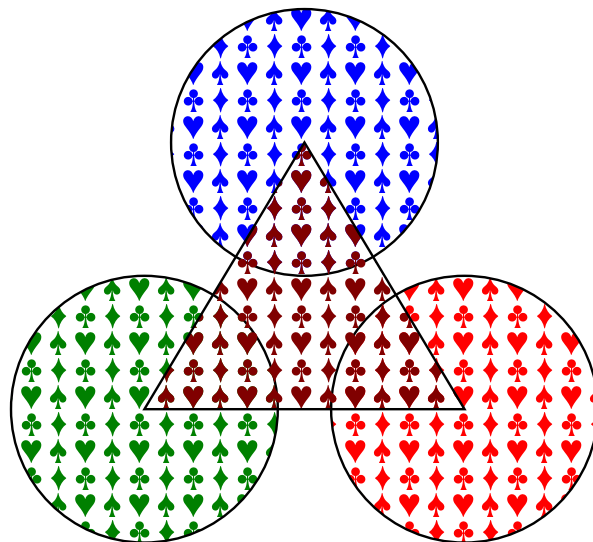
BeginTemplate

Syntax:

```
SI32 pdfBeginTemplate(
   const PPDF* IPDF, // Instance pointer
   double Width,     // Width of the template
   double Height)    // Height of the template
```

The function creates a template and opens it so that arbitrary contents can be drawn into it. After the objects of the template are drawn, it must be finished with EndTemplate(). To insert the template on a page use PlaceTemplate().

A template is a PDF object (Form XObject in PDF syntax) that can contain vector graphics, images, and text, but no interactive objects like annotations or form fields.

The difference between a normal page and a template is that a template can be used arbitrary often and scaled to different sizes without increasing the file size since a template is stored only once in the PDF file. Templates are used for repeating contents, such as static page backgrounds and so on, simply anything that must be drawn multiple times.

The initial background of a template is transparent.

When creating a template the function saves the graphics state and initializes all members of the new state to their default values.

EndTemplate() closes the template and restores the graphics state so that no changes made during creation of the template affect the parent object, if the template was created in an open page or other template.

A template can automatically be inserted to new pages. This can be achieved by adding the template to the list of auto-templates (see AutoTemplate() for further information). This can be useful if a template represents a fixed page background that must appear on all pages.

Remarks:

A template remains invisible until it was placed on a page with PlaceTemplate(). Unused templates are deleted when CloseFile() is called, they are not added to the PDF file.

Templates can be created outside of an open page and it is also possible to create a template outside of on open PDF file. But note: it is not possible to create a persistent template that can be used with multiple PDF files. All objects will be deleted when CloseFile() or FreePDF() is called.

Return values:

If the function succeeds the return value is the template handle, a value greater or equal zero. If the function fails the return value is a negative error code.

Example (C):

```c
#include "dynapdf.h"
// First we declare our error callback function
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}

int main(int argc, char* argv[])
{
   SI32 tmpl;
   void* pdf = pdfNewPDF();
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfCreateNewPDF(pdf, "c:/cppout.pdf");
   pdfSetPageCoords(pdf, pcTopDown);

   pdfAppend(pdf);

   tmpl=pdfBeginTemplate(pdf,pdfGetPageWidth(pdf),pdfGetPageHeight(pdf));
      pdfSetFont(pdf, "Arial", fsItalic, 40, true, cp1252);
      pdfWriteFText(pdf, taCenter, "This is a template!");
   pdfEndTemplate(pdf);

   pdfPlaceTemplate(pdf, tmpl, 0, 0, 0, 0);
   pdfPlaceTemplate(pdf, tmpl, 0, 50, 500, 0);
   pdfPlaceTemplate(pdf, tmpl, 0, 100, 450, 0);

   pdfEndPage(pdf);
   pdfCloseFile(pdf);
   pdfDeletePDF(pdf);
}
```

Output:

This is a template!

This is a template!

This is a template!

BeginTransparencyGroup

Syntax:

```
SI32 pdfBeginTransparencyGroup(
    const PPDF* IPDF,   // Instance pointer
    double x1,          // Lower left corner of the bounding box
    double y1,          // Lower left corner of the bounding box
    double x2,          // Upper right corner of the bounding box
    double y2,          // Upper right corner of the bounding box
    LBOOL Isolated,     // See description
    LBOOL Knockout,     // See description
    TExtColorSpace CS,  // The group's color space
    SI32 CSHandle)      // Required if a non-device space should be used
```

The function creates a transparency group and opens it so that arbitrary contents can be drawn into it. A transparency group is a special type of template (see BeginTemplate() for further information) that enables various effects in the transparent imaging model.

After the contents of the group was drawn the group must be finished with EndTemplate(). Yes, this is correct. A transparency group is just an extended template.

A transparency group can be used stand alone or as a soft mask. A soft mask consists of a transparency group that acts as an alpha channel and a corresponding soft mask dictionary that describes the characteristics of the mask. A soft mask is not directly usable, it must be activated with an extended graphics state. See CreateSoftMask() for further information.

The creation of a stand alone transparency group is essencially the same as a normal template and both objects share the same properties. A transparency group can be placed on a page with PlaceTemplate() or PlaceTemplateEx(), just like ordinary templates.

The difference between a template and a transparency group is that a transparency group is composited with the backdrop as a hole. This property is very important when a soft mask should be applied on an arbitrary artwork because a soft mask is designed to mask exactly one object at time.

The effect for overlapping objects drawn with regular techniques would be as if the soft mask is applied twice and this is mostly not what is intended.

However, transparency groups can also be used without an active soft mask. Whether this can be useful depends strongly on the effect that should be achieved. Most effects that can be achieved in this way can also be achieved with regular drawing techniques and the latter is always preferred.

The Group's Bounding Box

Due to the way how a transparency group is rended it is important to create the group in the correct size. Creating transparency groups larger than necessary can slow down rendering a lot, especially if many such groups are used on a page.

The way how the bounding box should be calculated is differently for soft masks and transparency groups which are used stand alone.

Stand alone transparency groups

A stand alone group can be placed on a page like a template. The lower left corner of the bounding box is typically set to zero, since this corner represents the coordinate origin of the group. The upper right corner is set to the wished width and height.

It is also possible to create the group larger than necessary and to modify the bounding box with SetBBox() when the real size is known. The media box represents the bounding box of the group. If the left and bottom members are set to non-zero values then use PlaceTemplateEx() to place the group on a page because this function considers the coordinate origin.

Soft masks

The calculation of the bounding box of a soft mask is slightly different because a soft mask is just a parameter of the graphics state and hence there is no direct way to specify where the mask should be rendered when it is activated.

The coordinate system in a soft mask is that of the parent object in which the mask is used. This is no problem so far but this conflicts with the coordinate logic in DynaPDF since no other object is drawn in this way.

Coordinates and bounding box of a soft mask:

Bounding box of the soft mask

```
x1 = 50
x2 = 350
y1 = 550
y2 = 650
```

Media box of the page
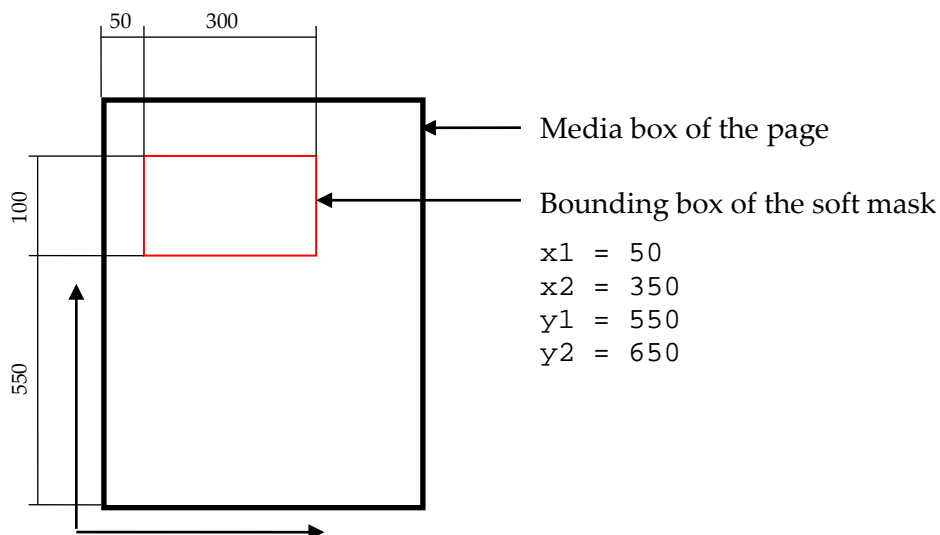
The bounding box of a soft mask can be seen like a clipping rectangle. The left and bottom members of the bounding box do not change the coordinate origin.

Before creating a soft mask it is usualy best to set SetUseVisibleCoords() to true. This property moves the coordinate origin into the visible area of the group. Now you can draw objects into the

group as if you would work with a normal template whose lower left corner is set to zero. This works with bottom-up and top-down coordinates.

Another way to avoid coordinate problems is to create the group in the size of the object in which the soft mask should be used. You can now draw into the group as if you would draw into the page or template. When finish, adjust the bounding box with SetBBox() to the required size.

Isolated and Non-Isolated Groups

The parameter *Isolated* specifies whether the initial backdrop of the group should be fully opaque or initialized to the current backdrop when the group is entered. Depending on how objects are drawn various effects can be achieved.

Knockout Groups

A knockout group is rendered as a hole, that means the top most group knockouts the groups below. The effect is as if the nested groups would be opaque, the top most wins. A knockout group is not meaningful if it does not draw other transparency groups since there is nothing that can be knocked out if no other groups are drawn. A knockout group can be useful if the entire group is composited with a soft mask.

Color Spaces

The color space in which a transparency group is rendered can be any device, ICC based, or calibrated color space, but no Lab, Indexed, DeviceN or Separation color space. If a non-device color space should be used then pass the handle of that color space to the parameter *CSHandle*. For device color spaces this parameter will be ignored.

The blending color space is important if the group should be composited with the background with a blend function. Since the result of blend functions depend strongly on the used color space, the blending color space makes sure that the same result can be achieved on different devices.

If no blend functions are used the blending color space should be set to DeviceGray or to a gray based ICC based color space.

How to use a Transparency Group?

As mentioned earlier a transparency group is just an extended template. A stand alone transparency group is placed on a page with PlaceTemplate() or PlaceTemplateEx() as usual. The only difference is that we use mostly a blend mode or a soft mask to composite the group with the backdrop.

All transparency related settings can be changed with an extended graphics state. The function CreateExtGState() creates such an extended graphics state, and SetExtGState() activates it, typically right before the group is placed on the page with PlaceTemplate():

```
...
grp = pdfBeginTransparencyGroup(pdf, ...);
...
```

```
pdfEndTemplate(pdf, ...);

pdfSetExtGState(pdf, gsHandle);  // Activate the wished settings
pdfPlaceTemplate(pdf, grp, ...); // Insert the group
pdfSetExtGState(pdf, defGState); // Restore the changes made before
...
```

When entering a transparency group a viewer initialized the fill and stroke alpha to 1.0 and sets the blend mode to bmNormal so that the transparency is not applied twice (one time on the entire group and one time on the objects in it). If the group represents a soft mask, the current soft mask is also explicitly deactivated before the group will be drawn.

If a soft mask should be applied on the group then the group should disable the current soft mask as the first command or before drawing the first object. Otherwise the mask is applied twice, one time on the entire group and another time on the objects in it.

Remarks:

It is bad practice to place a transparency group into an active clipping path or to activate a soft mask in a clipping path. Clip the contents inside the group but don't clip the group itself! A viewer must mostly adjust the clipping path so that it remains valid. This operation can be very computation intensive and should be avoided.

Return values:

If the function succeeds the return value is the template handle, a value greater or equal zero (a transparency group is an extended template). If the function fails the return value is a negative error code.

Bezier_1_2_3

Syntax:

```
SI32 pdfBezier_1_2_3(
   const PPDF* IPDF, // Instance pointer
   double x1,        // X-Coordinate of the first control point P1
   double y1,        // Y-Coordinate of the first control point P1
   double x2,        // X-Coordinate of the second control point P2
   double y2,        // Y-Coordinate of the second control point P2
   double x3,        // X-Coordinate of the end point P3
   double y3)        // Y-Coordinate of the end point P3
```

Curved path segments are specified as cubic Bézier curves. Such curves are defined by four points: the two endpoints (the current point P0 and the final point P3) and two control points P1 and P2. Given the coordinates of the four points, the curve is generated by varying the parameter *t* from 0.0 to 1.0 in the following equation:

$$R(t) = (1 - t)^3 \, P_0 + 3t(1 - t)^2 \, P_1 + 3t^2 \, (1 - t) \, P_2 + t^3 \, P_3$$

When *t* = 0.0, the value of the function R(t) coincides with the current point P0; when *t* = 1.0, R(t) coincides with the final point P3. Intermediate values of *t* generate intermediate points along the curve. The curve does not, in general, pass through the two control points P1 and P2. Cubic Bézier curves have two desirable properties:

- The curve can be very quickly split into smaller pieces for rapid rendering.

- The curve is contained within the convex hull of the four points defining the curve, most easily visualized as the polygon obtained by stretching a rubber band around the outside of the four points. This property allows rapid testing of whether the curve lies completely outside the visible region, and hence does not have to be rendered.

The most general type of curve in PDF is the cubic Bézier curve with two control points. The starting point P0 is the current point that was set before by a painting operator (MoveTo(), LineTo() and so on). PDF also supports two other Bézier curves with only one control point (see Bezier_1_3() and Bezier_2_3()). Theses two curve types are rarely used but still supported by all Acrobat versions incl. DynaPDF.

As mentioned earlier the function requires a start point that must be set with MoveTo() or another painting operator beforehand. Once the start point was set, multiple curve segments can be drawn to build circles, ellipses or other curved paths. The DynaPDF functions DrawArc(), DrawCircle(), Ellipse() and so on use all Bezier_1_2_3() to construct the path.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Bezier_1_3

Syntax:

```
SI32 pdfBezier_1_3(
   const PPDF* IPDF, // Instance pointer
   double x1,        // X-Coordinate of the first control point P1
   double y1,        // Y-Coordinate of the first control point P1
   double x3,        // X-Coordinate of the end point P3
   double y3)        // Y-Coordinate of the end point P3
```

This function paints a Bézier curve with one control point. The second control point P2 coincides with the final point P3 of the curve. The start point P0 must be set with MoveTo() or any other painting operator before.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Bezier_2_3

Syntax:

```
SI32 pdfBezier_2_3(
   const PPDF* IPDF, // Instance pointer
   double x2,        // X-Coordinate of the second control point P2
   double y2,        // Y-Coordinate of the second control point P2
   double x3,        // X-Coordinate of the end point P3
   double y3)        // Y-Coordinate of the end point P3
```

The third supported Bézier curve type paints also a curve with one control point. In this version the first control coincides with initial point P0 of the curve. The start point P0 must be set with MoveTo() or any other painting operator before.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Bézier curves:
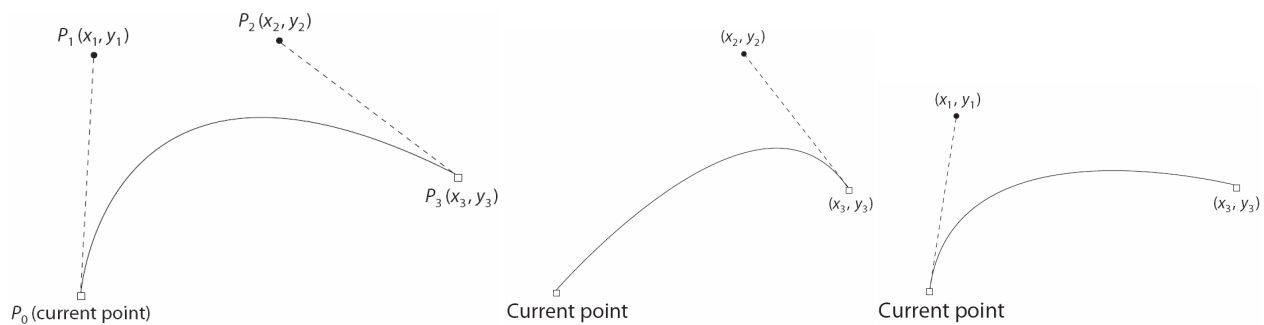
BuildFamilyNameAndStyle

Syntax:

```
LBOOL fntBuildFamilyNameAndStyle(
  const void* IFont,  // Pointer to font object
  char Name[128],     // 128 bytes long font name buffer
  TFStyle ADDR Style) // Font style
```

The function tries to extract the famaily name and font style from a font object. If the family name is not available or could not be algorithmically identified, the postscript name is copied to the parameter *Name* instead. The return value indicates whether the copied font name is probably a family name or a postscript name. The font selection mode should be set according to the return value when trying to load the font from the system (see SetFontSelMode()).

Return values:

If the family name could be extracted the return value is 1. If the font does not contain a family name or if there is no safe indication whether the font name represents a family name the return value is false.

Example (C++):

```cpp
SI32 SetPDFFont(const void* IFont)
{
   SI32 retval;
   TFStyle style;
   TFontSelMode selMode;
   char fontName[128];
   // Get the font name
   if (fntBuildFamilyNameAndStyle(IFont, fontName, fontStyle))
      selMode = smFamilyName;
   else
      selMode = smPostScriptName;
   // Set font selection mode for the first search run
   pdfSetFontSelMode(pdf, selMode);
   if ((retval = pdfSetFont(pdf, fontName, style, 12, true, cp1252)) < 0)
   {
      // If the font cannot be found then search again with reversed font
      // selection mode.
      if (selMode == smFamilyName)
         selMode = smPostScriptName;
      else
         selMode = smFamilyName;
      pdfSetFontSelMode(pdf, selMode);
      return pdfSetFont(pdf, fontName, style, 12.0, true, cp1252);
   }else
      return retval;
}
```

CalcPagePixelSize (Rendering Engine)

Sytax:

```
void rasCalcPagePixelSize(
   IPGE* PagePtr,          // Page pointer
   TPDFPageScale DefScale, // Scaling type
   float Scale,            // Scaling factor (if DefScale = psFitZoom)
   UI32 FrameWidth,        // Width of the output rectangle
   UI32 FrameHeight,       // Height of the output rectangle
   TRasterFlags Flags,     // Additional flags
   UI32* Width,            // Out -> Image width in pixels (required)
   UI32* Height)           // Out -> Image height in pixels (required)
```

The function calculates the image size of a page exactly in the way as RenderPage() does, if called with the same parameters. This makes it possible to create the image in the required size so that it can be rendered without a border.

CalcWidthHeight

Syntax:

```
double pdfCalcWidthHeight(
   const PPDF* IPDF,    // Instance pointer
   double OrgWidth,     // Original width of the image or template
   double OrgHeight,    // Original height of the image or template
   double ScaledWidth,  // Scaled Width
   double ScaledHeight) // Scaled Height
```

This function calculates the scaled width or height of a given size in the same way as InsertImage() or PlaceTemplate(). It can be used to calculate the new width or height of an image or template before it will be inserted to the page.

- If *ScaledWidth* == 0 and fabs(*ScaledHeight)* > 0 the function returns the scaled **width** with exact proportions in relation to the given values of *OrgWidth* and *OrgHeight*.

- If fabs(*ScaledWidth)* > 0 and *ScaledHeight* == 0 the function returns the scaled **height** with exact proportions in relation to the given values of *OrgWidth* and *OrgHeight*.

The function uses the absolute values so that the parameters can be negative. The parameters *OrgWidth* and *OrgHeight* must be greater zero.

Return values:

If the function succeeds the return value is unequal -1. If the function fails the return value is -1.

CaretAnnot

Syntax:

```
SI32 pdfCaretAnnot(
    const PPDF* IPDF,      // Instance pointer
    double PosX,           // X-Coordinate
    double PosY,           // Y-Coordinate
    double Width,          // Width of the annotation
    double Height,         // Height of the annotation
    UI32 Color,            // Color of the caret symbol
    TPDFColorSpace CS,     // Color space in which the color is defined
    const char* Author,    // Optional author
    const char* Subject,   // Optional subject
    const char* Content)   // Optional content
```

The function creates a caret annotation. This type of annotation is typically used to mark a position on a page where a user should add or edit text.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the bounding rectangle. If the coordinate system is top-down it defines the upper left corner.

This annotation type has an associated PopUp annotation that displays the string *Content* in a floating window. The initial window state of the associated PopUp annotation is closed by default but the state can be changed with SetAnnotOpenState() if necessary.

The caret annotation is the only annotation type that can be rotated in 90 degrees steps like form fields. This is not documented in the PDF specs but it works in all Acrobat and Reader versions.

To rotate the annotation call SetFieldOrientation(). Since this function requires normally a field handle, the annotation handle must be marked as annotation handle with the constant PDF_ANNOT_INDEX. The annotation handle must be added to the constant with a binary or operator.

Example (C++):

```
...
SI32 a = pdfCaretAnnot(pdf, 50.0, 50.0, 20.0, 20.0, PDF_BLUE,
                       csDeviceRGB, "Jim Tester", "Test", "Test");
// Rotate the annotation 90 degrees
pdfSetFieldOrientation(pdf, a | PDF_ANNOT_INDEX, 90);
...
```

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

ChangeAnnotName

Syntax:

```
SI32 pdfChangeAnnotName(
    const PPDF* IPDF,    // Instance pointer
    UI32 Handle,         // Annotation handle
    const char* Name)    // Annotation name or NULL
```

This function changes or deletes the optional unique name of an annotation. The parameter *Handle* must be a valid annotation handle. If the parameter *Name* contains an empty string or if it is set to NULL, the name will be deleted. The annotation name must be a unique string so that the annotation can be identified in JavaScript actions. The string could be a GUIID, a hash, or any other string that uniquely identifies the annotation.

Remarks:

This function is available in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeAnnotPos

Syntax:

```
SI32 pdfChangeAnnotPos(
    const PPDF* IPDF,    // Instance pointer
    UI32 Handle,         // Annotation handle
    double PosX,         // X-Coordinate of the annotation
    double PosY,         // Y-Coordinate of the annotation
    double Width,        // Width in unscaled units
    double Height)       // Height in unscaled units
```

This function changes the position and size of an annotation. The bounding box of annotations is defined in bottom up coordinates. The reference point is *Left*, *Bottom* of the bounding box that can be retrieved by GetAnnotBBox(), GetAnnotEx(), or GetPageAnnotEx(), for example.

The annotation extends to top, right when scaling to a larger size. The function rebuilds the appearance stream after changing the position or size if possible.

Note that the function considers the current coordinate system (bottom up or top down). In order to change the position or size of an existing annotation, the coordinate system should be set to *pcBottomUp* beforehand (see SetPageCoords() for further information).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeBookmark

Syntax:

```
SI32 pdfChangeBookmark(
   const PPDF* IPDF,   // Instance pointer
   SI32 ABmk,          // Handle of a bookmark
   const char* Title,  // New title
   SI32 Parent,        // New parent bookmark
   UI32 DestPage,      // New destination page
   SI32 Open)          // Open the bookmark?
```

This function changes an existing bookmark. There is no difference between imported bookmarks and bookmarks which were created with DynaPDF. The parameter *ABmk* requires a valid bookmark handle. A bookmark handle is a simple array index ranging from zero to GetBookmarkCount() -1.

The parameter *Title* can be NULL if it should be deleted. The parameter *Open* will be ignored if the bookmark contains no children.

The function does not check whether the destination page exists at the time the bookmark is changed. If the destination page does not exist when the file is closed then it will be set to page 1.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252 only. To create a title in an arbitrary encoding, convert the string to Unicode with the function ConvToIncode() and use the Unicode version to change the bookmark.

To get the handle of a specific bookmark use the functions FindBookmark() and FindNextBookmark() to find the bookmark.

It is also possible to enumerate all bookmarks by using GetBookmarkCount() and GetBookmark() until the used bookmark was found (see GetBookmark() for further information).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeNamedDest

Syntax:

```
LBOOL pdfChangeNamedDest(
   const PPDF* IPDF,           // Instance pointer
   TObjType ObjType,          // Object type to which the handle belongs
   UI32 Handle,               // Object handle
   const char* NewDestName)   // New destination name

typedef enum TObjType
{
   otAction,      // Ok
   otAnnotation,  // Ok
   otBookmark,    // Ok
   otCatalog,     // Unsupported
   otField,       // Unsupported
   otPage,        // Unsupported
   otPageLink     // Ok
}TObjType;
```

The function changes the name of a named destination. Named destinations can be accessed by link annotations, bookmarks, and actions. For link annotations the object type otPageLink or otAnnotation can be used.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeFont

Syntax:

```
SI32 pdfChangeFont(
   const PPDF* IPDF, // Instance pointer
   SI32 Handle)      // Handle of the new font
```

This function changes or sets the font to the one of the parameter *Handle*. *Handle* must be a valid font handle that was returned by SetFont(), SetFontEx(), SetCIDFont(), or LoadFont().

The font size is taken from the current font if any. If no font is active when the function is called, the font size will be set to the value that was used to load font the last time.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeFontEx

Syntax:

```
LBOOL pdfChangeFontEx(
   const PPDF* IPDF, // Instance pointer
   SI32 Handle,      // Font handle
   double FontSize,  // Font size
   TFStyle Style)    // Style flags
```

This function changes or sets the font to the one of the parameter *Handle*. *Handle* must be a valid font handle that was returned by SetFont(), SetFontEx(), SetCIDFont(), or LoadFont().

The style flags should be the same as in the initial SetFont() call with exception of the flags *fsUnderline* or *fsStriked* which can be used to underline or strikeout text.

The parameter *FontSize* must be greater zero.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeFontSize

Syntax:

```
SI32 pdfChangeFontSize(
   const PPDF* IPDF) // Instance pointer
   double Size)      // New font size
```

This function changes the font size of the current font. The font size must be greater zero.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeFontStyle

Syntax:

```
LBOOL pdfChangeFontStyle(
    const PPDF* IPDF, // Instance pointer
    TFStyle Style)    // New font style

typedef SI32 TFStyle;
#define fsNone        0
#define fsUnderlined  4
#define fsStriked     8
```

This function changes the style of the current font. Only the style flags fsUnderlined and fsStriked can be set with this function. The font styles fsBold and fsItalic requires another font file that must be selected with SetFont().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeFontStyleEx

Syntax:

```
LBOOL pdfChangeFontStyleEx(
    const PPDF* IPDF, // Instance pointer
    TFStyle Style)    // New font style
```

The function changes the font style of a font like ChangeFontStyle() but it accepts also font styles like fsBold or fsItalic. These flags can be used to explicitely enable the emulation of a missing font style.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeJavaScript

Syntax:

```
SI32 pdfChangeJavaScript(
    const PPDF* IPDF,       // Instance pointer
    UI32 AHandle,           // Handle of global JavaScript
    const char* NewScript)  // New JavaScript
```

The function replaces a global JavaScript with a new one. The parameter *AHandle* must be a valid handle of a global JavaScript. The parameter *NewScript* must be a valid JavaScript. The script is not checked by DynaPDF whether it is valid or not. Note that older Acrobat versions do not support all JavaScript functions. Due to several bugs in certain Acrobat versions invalid scripts can cause access violations in Adobe's Acrobat. Test your scripts carefully with all Acrobat versions which must be supported.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. However, because JavaScript 1.2 is not Unicode compatible, Unicode encoded scripts are translated to a platform specific encoding prior to interpretation by the JavaScript engine. This conversion is done in the viewer application and can cause errors. The usage of scripts in Unicode format should be avoided if possible.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeJavaScriptAction

Syntax:

```
SI32 pdfChangeJavaScriptAction(
   const PPDF* IPDF,        // Instance pointer
   UI32 AHandle,            // Handle of a JavaScript action
   const char* NewScript)   // New JavaScript
```

The function replaces the script of a JavaScript action with a new one. The parameter *AHandle* must be a valid handle of a JavaScript action. The parameter *NewScript* must be a valid JavaScript. The script is not checked by DynaPDF whether it is valid or not. Note that older Acrobat versions do not support all JavaScript functions. Due to several bugs in certain Acrobat versions invalid scripts can cause access violations in Adobe's Acrobat. Test your scripts carefully with all Acrobat versions which must be supported.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Because JavaScript 1.2 is not Unicode compatible, Unicode encoded scripts are translated to a platform specific encoding prior to interpretation by the JavaScript engine. This conversion is done in the viewer application and can cause errors. The usage of scripts in Unicode format should be avoided if possible.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeJavaScriptName

Syntax:

```
SI32 pdfChangeJavaScript(
   const PPDF* IPDF, // Instance pointer
   UI32 AHandle,     // Handle of global JavaScript
   const char* Name) // New name of the script
```

This function changes the name of a global JavaScript. The parameter *Name* defines the name of this script. This name is **NOT** used to identify a function inside the JavaScript; it is only used in Adobe's

Acrobat to select a specific script. The script name must be unique. Different scripts cannot not use the same name.

Return value:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeLinkAnnot

Syntax:

```
SI32 pdfChangeLinkAnnot(
   const PPDF* IPDF, // Instance pointer
   UI32 Handle,      // Annotation handle
   const char* URL)  // URL or file path
```

This function changes the link of a file link or web link annotation. The parameter *AHandle* must be a valid handle of a file link, page link, or web link Annotation. The parameter *URL* can be either a file path or an URL to an internet resource; it must not be an empty string or NULL. Note that you cannot change the type of the annotation with this function. For example, a web link annotation requires still a URL to an internet resource, while a file link annotation requires a path to a file on a local hard drive.

Remarks:

Imported file link or web link annotations are mostly defined as page link annotation because it is impossible to distinguish between these types during import. Use the function FindLinkAnnot() if you need to change an imported file link or web link annotation.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeOCGName

Syntax:

```
LBOOL pdfChangeOCGName(
   const PPDF* IPDF,  // Instance pointer
   UI32 Handle,       // OCG handle
   const char* Value) // New name (required)
```

The function changes the name of an Optional Content Group (OCG). See also Layers (Optional Content).

Remarks:

This function is implemented as in Ansi and Unicode compatible variant.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ChangeSeparationColor

Syntax:

```
LBOOL pdfChangeSeparationColor(
   const PPDF* IPDF,          // Instance pointer
   UI32 CSHandle,             // Handle of the separation color space
   UI32 NewColor,             // New color in the alternate color space
   TExtColorSpace Alternate,  // Alternate color space
   SI32 AltHandle)            // Color space handle or -1 for device spaces
```

The function changes the color of a separation color space. The new color value must be defined in the alternate color space. The alternate color space can be any device or ICC based color space including Lab. If a non-device color space is used, the parameter *AltHandle* must be set to the color space handle of the alternate color space.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CheckCollection

Syntax:

```
LBOOL pdfCheckCollection(
   const PPDF* IPDF) // Instance pointer
```

The function checks whether user defined data fields in embedded files are consistently defined with collection fields. User defined collection fields can be used to provide additional information relating to embedded files. It is also possible to use such fields to sort the list of embedded files in an arbitrary manner. However, the data types used in the embedded file specification must correspond to the definition in the collection.

This function is especially useful to check the integrity of the file if multiple PDF collections were merged into one file.

See also CreateCollection(), CreateCollectionField().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CheckConformance

Syntax:

```
SI32 pdfCheckConformance(
   const PPDF* IPDF,                         // Instance pointer
   TConformanceType Type,                    // PDF Version to check
   TCheckOptions Options,                    // various options
   const void* UserData,                     // User defined pointer
   TOnFontNotFoundProc* OnFontNotFound,      // Font replacement callback
   TOnReplaceICCProfile* OnReplaceICCProfile) // ICC profile callback
```

```
typedef enum
{
    ctPDFA_1b_2005, // Compatibility to PDF/A 1b 2005
    ctNormalize,    // Normalization is useful for print workflows
    ctPDFA_2b,      // Convert the file to PDF/A 2b if possible
    ctPDFA_3b,      // Convert the file to PDF/A 3b if possible
    /* The following constants convert the file to PDF/A 3b and set the whished ZUGFeRD, ZUGFeRD 2.0
     * or Factur-X conformance level in the XMP metadata. CheckConformance() does not validate the
     * XML invoice but it checks whether it is present. The embedded XML invoice must be named as
     * follows (case sensitive and without quotes):
     *
     *   ZUGFeRD 1.0: "ZUGFeRD-invoice.xml"
     *   ZUGFeRD 2.0: "zugferd-invoice.xml"
     *   Factur-X:    "factur-x.xml"
     *   XRechnung:   "xrechnung.xml"
     *
     * Setting the correct ZUGFeRD or Factur-X conformance level is important since this value
     * defines which fields must be present in the XML invoice.
     */
    // ZUGFeRD 1.0
    ctZUGFeRD_Basic,     // Basic profile
    ctZUGFeRD_Comfort,   // Comfort profile
    ctZUGFeRD_Extended,  // Extended profile
    // ZUGFeRD 2.0:
    ctZUGFeRD2_Minimum,  // Minimum profile
    ctZUGFeRD2_Basic,    // Basic profile
    ctZUGFeRD2_Basic_WL, // Basic WL profile
    ctZUGFeRD2_Comfort,  // EN 16931 profile
    ctZUGFeRD2_Extended, // Extended profile
    // Factur-X / ZUGFeRD 2.1:
    ctFacturX_Minimum,   // Minimum profile
    ctFacturX_Basic,     // Basic profile
    ctFacturX_Basic_WL,  // Basic WL profile
    ctFacturX_Comfort,   // EN 16931 profile
    ctFacturX_Extended,  // Extended profile
    ctFacturX_XRechnung, // German XRechnung profile
    ctPDFA_2u,           // Convert the file to PDF/A 2u if possible
    ctPDFA_3u,           // Convert the file to PDF/A 3u if possible
    ctPDFA_4,            // Convert the file to PDF/A 4 if possible. Embedded files are supported but
                         // no file attach annotations.
    ctPDFA_4e,           // Convert the file to PDF/A 4e if possible. This version allows 3D contents
                         // in rich media annotations.
    ctPDFA_4f            // Convert the file to PDF/A 4f if possible. This version allows file attach
                         // annotations.
}TConformanceType;

typedef UI32 TCheckOptions;
#define coDefault                0x0010FFFF // Default flags
#define coEmbedSubsets           0x00000001 // Already done, does nothing (obsolete)
#define coDeleteTransferFuncs    0x00000002 // Imported files only
#define coDeleteMultiMediaContents 0x00000004 // Imported files only
#define coDeleteActionsAndScripts 0x00000008 // Imported files only
#define coDeleteInvRenderingIntent 0x00000010 // Imported files only
#define coFlattenFormFields      0x00000020 // Flatten form fields
#define coReplaceV4ICCProfiles   0x00000040 // Imported files only
#define coDeleteEmbeddedFiles    0x00000080 // Imported files only
#define coDeleteOPIComments      0x00000100 // Imported files only
#define coDeleteSignatures       0x00000200 // Imported files only

#define coDeletePostscript       0x00000400 // Delete Postscript XObjects
#define coDeleteAlternateImages  0x00000800 // Delete alternate images
#define coReComprJPEG2000Images  0x00001000 // JPEG 2000 images are prohibited in PDF/A 1. The
                                            // conversion to PDF/A 1 will fail if the flag is
                                            // absent and if a file contains JPEG 2000 images.
#define coResolveOverprint       0x00002000 // PDF/A 2 and 3. Set the overprint mode to 0 if
                                            // overprint mode = 1 and if overprinting for fill or
                                            // stroke is true and if a ICCBased CMYK color space is
                                            // used. Note that DeviceCMYK is treated as ICCBased
```

```
                                            // color space due to implicit color conversion rules.
#define coMakeLayerVisible          0x00004000 // PDF/A 2 and 3 prohibit invisible layers.
#define coDeleteAppEvents           0x00008000 // PDF/A 2 and 3. Application events are prohibited in
                                            // PDF/A. The view state will be applied.
#define coReplCCITTFaxWithFlate     0x00010000 // Imported files only
#define coApplyExportState          0x00020000 // Meaningful only if coDeleteAppEvents is set. Apply
                                            // the export state.
#define coApplyPrintState           0x00040000 // Meaningful only if coDeleteAppEvents is set. Apply
                                            // the print state.

#define coDeleteReplies             0x00080000 // Delete annotation replies. If absent, replies will be
                                            // converted to regular text annotations.
#define coDeleteHalftones           0x00100000 // Delete halftone screens.
#define coFlattenLayers             0x00200000 // PDF/A 1b only. Flatten layers if any.
#define coDeletePresentation        0x00400000 // Presentations are prohibited in PDF/A 2 and 3.

#define coCheckImages               0x00800000 // Images will be decompressed to identify damages.
#define coDeleteDamagedImages       0x01000000 // Meaningful only if coCheckImages is set.
#define coRepairDamagedImages       0x02000000 // Meaningful only if coCheckImages is set. If set, try
                                            // to recompress a damaged image. The new image is maybe
                                            // incomplete but error free. This flag can be combined
                                            // with coDeleteDamagedImages to delete the image if
                                            // recompression fails.
#define coNoFontEmbedding           0x10000000 // Normalization only
#define coFlushPages                0x20000000 // Write converted pages directly into the output file
                                            // to reduce the memory usage.
#define coAllowDeviceSpaces         0x40000000 // If set, device color spaces will not be replaced with
                                            // ICC based color spaces. This flag is meaningful for
                                            // normalization only.
#define coResetAnnotAppearance      0x80000000 // If set, appearance streams of annotations are rebuild
                                            // before executing CheckConformance().
// Common default flags for normalization.
#define coDefault_Normalize (coAllowDeviceSpaces | coNoFontEmbedding)
// Common default flags for different PDF/A versions:
#define coDefault_PDFA_1    (coDefault | coFlattenLayers)
#define coDefault_PDFA_2    (coDefault | coDeletePresentation)
#define coDefault_PDFA_3    (coDefault_PDFA_2 & ~coDeleteEmbeddedFiles)
#define coDefault_PDFA_4    (coDefault_PDFA_2 & ~coDeleteEmbeddedFiles)

typedef SI32 PDF_CALL TOnFontNotFoundProc(
    const void* Data,     // Pointer UserData of CheckConformance
    const void* PDFFont,  // PDF font handle
    const char* FontName, // PostScript name of the font
    SI32 Style,           // Font style
    SI32 StdFontIndex,    // Standard font index
    LBOOL IsSymbolFont);  // Set to 1 if the font is a symbol font

typedef SI32 PDF_CALL TOnReplaceICCProfile(
    const void* Data,     // Pointer UserData of CheckConformance
    TICCProfileType Type, // Required ICC profile type to be embedded
    SI32 ColorSpace);     // Color space handle

typedef enum
{
    ictGray = 0,
    ictRGB  = 1,
    ictCMYK = 2,
    ictLab  = 3
}TICCProfileType;
```

The function converts a PDF file to a specific PDF standard according to the parameter *Type.*

The regular DynaPDF versions contain a restricted version of CheckConformance() which cannot be used if one or more pages from external PDF files were imported.

The conversion of arbitrary imported PDF files requires a separate license of the PDF/A Extension.

The PDF/A Extension represents in fact the unrestricted version of this and related functions (ReplaceFont() and so on) which are required to convert arbitrary PDF files to PDF/A.

If the parameter *Type* is set to ctNormalize the function checks the PDF file for errors, rebuilds all embedded fonts, optionally embeds non-embedded fonts, and repairs potential file errors if possible. The resulting PDF file is easy to read and should not produce any printing error. Normalization is useful in print workflows to detect potential errors before the file will be printed. This feature can be used to check imported PDF files or pages. It is not meaningful when creating new PDF files with DynaPDF.

Note that the flag **if2Normalize** must be set with SetImportFlags2() when importing external PDF files. This flag enables additional error checks during import.

Note also that CheckConformance() is no preflight function! It is possible to check whether a PDF file can be converted to PDF/A but it is **not** possible to check whether a file is already a valid PDF/A file!

This is a big difference since a preflight function must apply many additional checks, e.g. if PDF objects are formatted as described in the PDF/A standard, as well as all other requirements are meet. Such checks are not required in a converter since the file structure and most objects will be rebuild when the file is stored. It is not planned to extend the function with preflight capabilities in the near future.

The function supports a large set of processing options but most of them are only meaningful if imported PDF files or imported pages should be converted to PDF/A.

Notice

CheckConformance() cannot check glyph outlines of simple TrueType fonts because the required overhead would be too large. However, the function makes sure that outlines will be checked when CloseFile() is called. Therefore, be aware that CloseFile() can fail also if CheckConformance() succeeds.

Important callback functions

OnFontNotFound

The callback function *OnFontNotFound* is not required to check newly created PDF files but it should be set when converting imported PDF files. As shown in the example code, a default font like Arial can be loaded whenever a PDF font cannot be found on the system. The function checks whether the font metrics of the replacement font are compatible with the original PDF font.

OnReplaceICCProfile

The *OnReplaceICCProfile* callback function is optional too, but mostly required! The callback function is called whenever a device color space is used and if no corresponding output intent or default ICC

based color space is available. It is also called if an embedded ICC profile must be replaced due to damages or if an unsupported profile version is used.

The conversion capabilities are very limited if this callback function is not set! If you get an error like " File uses multiple device color spaces!" then please set this callback function so that ICC profiles can be loaded if necessary.

General requirements when creating PDF/A files:

- When importing PDF files or pages, the flag **ifPrepareForPDFA** must be set with SetImportFlags().

- Fonts must be embedded.

- Images must not contain an alpha channel (PDF/A 1b only).

- The flag *gfUseImageColorSpace* should be set when inserting images. This flag makes sure that embedded ICC profiles will be considered. See SetGStateFlags() for further information.

- PDF/A 1a files are tagged PDF files. That means the entire PDF file must be tagged and the file must be created in the logical reading order. CheckConformance() does not validate the structure tree and it does not check whether untagged elements are used on a page. See CreateStructureTree(), OpenTag(), and so on for further information.

- PDF/A files require an Ouptut Intent. The output intent represents the destination color space for which the file was created. The output intent must be added **after** CheckConformance() was executed. The required ICC profile must be attached depending on the return value of the function (see Return Values below).

With CheckConformance() you can check whether prohibited features were used. Once the right settings are found, it is normally no longer required to call this function when creating new PDF files. The parameter *CheckOptions* should be set to coDefault or to zero so that prohibited features raise an exception.

If you have a license of the PDF/A Extension for DynaPDF, then you can also convert arbitrary imported PDF files to PDF/A 1b, 2b, 3b, 4, 4e, and 4f.

In this case, a few additional notes should be considered:

- External CMaps should be loaded delayed with SetCMapDir() if possible. The CMaps are sometimes required, especially for Asian PDF files.

- It is strongly recommendet to set an error callback function before CheckConformance() is executed. The function passes warnings and error messages to the callback function if set. These messages should be stored in an error log so that the user can check whether changes were made. Errors are always fatal; the return value is ignored in this case. When a warning is passed to the error callback function (this is the case when something must be changed), then the return value can be either 0 to accept the change or -1 to break processing. In the latter case no PDF file will be produced.

It is also possible to redirect all error messages and warnings into the error log of DynaPDF. See SetErrorMode() for further information.

- The *OnFontNotFont* callback function is called when a non-embedded font is not available and if no alternative font could be found on the system (DynaPDF uses already a mapping table to find good alternative fonts for many common fonts). Note that many PDF files contain invalid font names. Since PDF files created by a specific application use mostly a unique naming scheme, it is often possible to create your own mapping table for specific font names so that such files can still be converted.

  If your application knows a suitable alternate font then you can replace the PDF font with the new one with ReplaceFont() or ReplaceFontEx(). The return value of the callback function must be the retun value of ReplaceFont() or ReplaceFontEx() in this case. If no suitable alternate font is known, then you can try to use a default font like Arial or return -1 to break processing. A standard replacement font should be loaded with ReplaceFont().

  CheckConformace() checks whether the alternate font contains compatible glyph metrics and whether all required glyphs are avaialble. Although it is very often possible to replace unknown fonts with Arial, it should be checked whether the appearance of the new font is acceptable.

  Since different fonts use different glyph widths, text rendered with a replacement font looks often not as good as if the same text was rendered with a standard font that a viewer used to display the text, also if the same font was used in both cases.

  The reason why this happens is very simple: When a PDF viewer uses a standard font to display text of a non-available font, then glyphs will be scaled to the widths of the original non-embedded font (the glyph widths are stored in the PDF file). That is the reason why it is often not noticeable that a text was rendered with an alternate font.

  However, this scaling is no longer available when the font becomes embedded. To achieve the same result it would be required to create a synthetic font that contains scaled outlines for every glyph but such a feature is not available in DynaPDF.

- The *OnReplaceICCProfile* callback function is called when an additional default ICC based color space must be created (this is the case when a page uses mixed color spaces), or when a PDF file contains either damaged or ICC profiles of an unsupported version.

  It is relatively seldom required to replace embedded ICC profiles but it is often required to create additional default color spaces. So, it is very important that this callback function is present. In the callback function you must call ReplaceICCProfile() with the requested color profile type or return -1 if no such profile is available. The return value of the callback function must be set to the one of ReplaceICCProfile().

  The most common ICC profiles, which are also used in Adobes Acrobat or Reader, can be downloaded from the Adobe website. These profiles are freely available and can be

delivered with your application. Please consider the license agreement of the profile package.

The pointer *UserData* of CheckConformance() is always passed unchanged to the callback functions. This parameter is not available and not required in C#, VB .Net, or PHP.

Type3 font conversion

CheckConformance() is able to convert the 14 PDF standard fonts to Type3 counterparts if no suitable system font is available (see SetFont/The 14 Standard Fonts for further information). However, due to issues in Adobe's Acrobat or Reader, and due to the way how Type3 fonts are defined in PDF, several restrictions must be considered:

- Whereas all other font formats define only the shape of a glyph, Type3 fonts define the shape and the way how glyphs must be painted, e.g. filled, stroked, or both. If a Type3 font should be used with different text draw modes, then separate fonts must be created for each mode, e.g. a font for filled glyphs, one for stroked and one for filled and stroked glyphs.

- It is also not possible to draw stroked Type3 glyphs with arbitrary line widths, dash patterns, or different style parameters as it is possible with every other font format since the specs say that all these parameters must be explicitly defined in glyph content streams!

- With exception of Acrobat 8 and 9, Acrobat and Reader use the current fill color instead of the stroke color to render stroked Type3 glyphs that were defined with the d1 operator. The only way to avoid this issue is to define the text color in every glyph. This requires again a seperate font for every color with which the font should be used!

Summay

We can assume that it doesn't make sense to create dozens of fonts due to the idiotic definition of the Type3 font format. This function creates exactly one Type3 counterpart for a standard font with the graphics state parameters when the font is first used. If the font will be used with another draw mode later then the function returns immediatly with an error and no PDF file will be produced.

If a stroked version of a font was created then it does not check whether the line cap style, line join style, line dash pattern or line width will be changed if the font is used with the same text draw mode later.

Normalization

When normalizing PDF files set also the flag **if2Normalize** with SetImportFlags2(). This flag enables a few additional checks when importing a PDF file. Note that it is not meaningful to normalize a PDF file before calling CloseFile() if the file was fully created by DynaPDF beforehand. This would just waste processing time.

Normalization is a process to make sure that a PDF file is error free and does not cause printing errors. The function applies almost the very same checks as for PDF/A 2 conversion. Missing glyphs are reported but do not break processing.

Note that it is in fact not possible to determine whether a glyph is really missing. The function can only check whether a glyph on index 0 is referenced somewhere or if a font does not contain a requested glyph and falls back to notdef. Also the latter case can be wanted behaviour.

Index 0 is reserved for the notdef glyph in all font formats but many PDF creators decided to store the first glyph on index 0. This was mostly an optimization in older PDF libraries but this kind of optimization violates font standards. So, whether a glyph is really missing cannot be determined algorithmically but we can assume that something went wrong during the file creation.

However, for a printer it is not relevant whether the glyph to be printed is located on index 0 or somewhere else. So, whether missing glyph warnings should be ignored or not is up to the caller.

Font embedding

Printing errors on Postscript devices have mostly to do with non-embedded fonts. The function can embed non-embedded fonts but it is also possible to convert glyphs of non-embedded fonts to outlines with Optimize(). The latter solution is usually better because font substitution can lead to visible distortions if the glyph metrics are not fully compatible.

The reason why the conversion of non-embedded fonts to outlines produces often better results is the fact that glyphs of embedded fonts are not scaled to the destination widths as it would be the case if the font is not embedded (PDF fonts contain the widths of all glyphs, also if the font is not embedded).

Converted outlines preserve this scaling and that is the reason why the result looks often good also if a substituted font contains incompatible metrics.

If you want to combine Optimize() with CheckConformance() without storing the result in a temporary file then set the flag *ofInMemory* but do **not** set the flag *ofNewLinkNames* since this can cause errors in the CheckConformance() call. Note that processing in memory should be done for relatively small files only since the memory usage grows quickly if many pages need to be converted.

PDF/A 1b:

When creating new PDF/A 1a or PDF/A 1b files, the following features are prohibited:

- The fill or stroke alpha constant in an extended graphics state must be 1.0 if present (see CreateExtGState()).

- Transparency groups, blend modes, as well as alpha channels in images.

- Layers (CreateOCG(), CreateOCMD() and all related functions).

- Annotations which are not defined in PDF 1.4. Highlight annotations cannot be used since these annotations require the blend mode Multiply.

- Form fields (form fields will be flattened if present). Note that check boxes use the font ZapfDingbats which is mostly not present on a Windows system.

- Embedded ICC profiles with a major version higher than 2. Version 4 profiles cannot be used in PDF/A 1 files.

- PDF/A files cannot be encrypted. The usage of CloseFileEx() or CloseAndSignFileEx() is not allowed.

- All features which are not defined in PDF 1.4.

PDF/A 2b, 2u, 3b, 3u

PDF/A 2b, 2u, 3b, 3u based on PDF 1.7 and hence support more features like transparency or optional contents (layers). The only difference between PDF/A 2b and 3b is that the latter version supports also embedded files. The following features are prohibited:

- Annotations which are not defined in PDF 1.7.

- Form fields (form fields will be flattened if present).

- Overprinting is permitted but the overprinting mode cannot be set to 1 if an ICCBased CMYK color space is used. Due to implicit color conversion rules this applies also to DeviceCMYK.

- Application events are prohibited in PDF/A 2 and 3 (see AddOCGToAppEvent() for further information).

- Annotation replies are still prohibited (see SetAnnotMigrationState() for further information).

- PDF/A files cannot be encrypted. The usage of CloseFileEx() or CloseAndSignFileEx() is not allowed.

- PDF/A 3b: Embedded files must be associated with a PDF object. See AssociateEmbFile() for further information.

PDF/A 4, 4e, 4f

PDF/A 4 files based on PDF 2.0. PDF/A 4e and 4f support embedded files but file attach annotations are supported by PDF/A 4f only.

Main differences in comparison to PDF/A 3:

- The flag *coFlushPages* is **not** supported since the special color space handling requires a cleanup run when closing the file. Links to color space resources must might be changed in this run. This is only possible if parts of the document have not already been written to the output file.

- Overprinting is fully supported.

- Geospatial and rectilinear measurement properties are supported.

- Type1 and Type5 hafltone screens are supported.

- PDF/A 4e enables beneeth embedded files the usage of 3D contents in RichMedia annotations. Note that RichMedia annotations are based on Flash.

- File attach annotations are supported by PDF/A 4f only.

It is strongly recommended to set the import flag if2UseProxy with SetImportFlags2() to reduce the memory usage when importing PDF files.

Since DynaPDF must rebuild all content streams of the PDF file, the flag *coFlushPages* can also be set to reduce the memory usage during conversion (not supported by PDF/A 4). This flag can be set if a PDF file is larger than 5 or 10 MB for example. For small PDF files the flag should be absent.

If the flag *coFlushPages* was set then a few objects must be written as indirect references. If you want a perfectly organized PDF file then import the resuling PDF file again and close the file, finished. This removes unnecessary references. However, be aware that this saves only a few bytes.

ZUGFeRD, ZUGFeRD 2.0, Factur-X / ZUGFeRD 2.1, XRechnung

The only difference between PDF/A 3b and ZUGFeRD, ZUGFeRD 2.0, or Factur-X and ZUGFeRD 2.1 or higher is that the file must contain an embedded XML invoice. The embedded invoice must be named as follows (case-sensitive and without quotes):

- ZUGFeRD 1.0:   "ZUGFeRD-invoice.xml"
- ZUGFeRD 2.0:   "zugferd-invoice.xml"
- Factur-X / ZUGFeRD 2.1: "factur-x.xml"
- XRechnung: "xrechnung.xml"

Note that ZUGFeRD 2.1 or higher and Factur-X are technically indenticly defined, not only in PDF! Therefore, both formats share the same version constants!

The invoice can be attached with AttachFile() or AttachFileEx(). The embedded file must also be associated with the global catalog object with AssociateEmbFile(). This step is optional. If the file was not already associated with the catalog object then it will be associated automatically.

However, if set, the parameter *Relationship* must be set to *arAlternative*, *arData*, or *arSource* depending on the output version:

Profile	Relationship
ctZUGFeRD_Basic	arAlternative
ctZUGFeRD_Comfort	arAlternative
ctZUGFeRD_Extended	arAlternative
ctZUGFeRD2_Minimum	arData or arSource
ctZUGFeRD2_Basic	arAlternative
ctZUGFeRD2_Basic_WL	arData or arSource
ctZUGFeRD2_Comfort	arAlternative
ctZUGFeRD2_Extended	arAlternative
ctFacturX_Minimum	arData or arSource
ctFacturX_Basic	arAlternative
ctFacturX_Basic_WL	arData or arSource
ctFacturX_Comfort	arAlternative
ctFacturX_Extended	arAlternative

ctFacturX_XRechnung	arSource

Note that the relationship for the profiles Basic, Comfort (EN 16931), and Extended must be set differently depending on the country for which the invoice will be created. An invoice created for a German recipient must set the relationship to *arAlternative* while the same invoice created for a recipient in France must set it to *arSource*.

CheckConformance() sets the relationship to *arAlternate* or *arData* for the MINIMUM and BASIC WL profiles for ZUGFeRD 2.0 or Factur-X / ZUGFeRD 2.1 if it was not already set. If *arSource* should be used then call AssociateEmbFile() for the emdedded file (the attached XML invoice). Set the parameter *DestObject* to *adCatalog* in this case.

If the embedded invoice cannot be found then a fatal error will be raised and no PDF file will be created. If the file was found then the remaining settings will be applied if not already present.

ZUGFeRD, and Factur-X are permanently growing standards. Minor changes in a standard do often not affect the PDF container. For example, the FacturX constants are compatible with at least ZUGFeRD 2.2, and with Factur-X up to version 1.0.0.6. Factur-X 1.0.0.6 and ZUGFeRD 2.2 are technically identically defined. The name ZUGFeRD is used for historical reasons.

Please send us an email if you're unsure whether a specific version is supported by DynaPDF.

Notice:

Although it would be technically no problem to create one large PDF file for multiple invoices, this is not allowed in these standards. A valid ZUGFeRD or Factur-X invoice file can contain one invoice only.

PDF/A conversion and normalization with enabled font embedding depends strongly on the availability of fonts. The ability to convert standard fonts to Type3 was mainly added for the font ZapfDingbats because this font is not available on most systems.

Return values

If the PDF file is compatible to PDF/A 1b the function returns one of the following values:

- **0** -> The PDF file is fully compatible to PDF/A 1b. Close the file with CloseFile(), finished.
- **1** -> A RGB ICC profile must be added to the document with AddOutputIntent().
- **2** -> A CMYK ICC profile must be added to the document with AddOutputIntent().
- **3** -> A Gray, RGB, or CMYK ICC profile must be added to the document with AddOutputIntent() or AddOutputIntentEx(). DeviceGray compatible ICC profiles are rarely available since this is mostly just a gamma table. However, you can use a RGB or CMYK profile instead (a sRGB profile is preferred due to the smaller size).
- **< 0** -> A negative return value indicates that an error occurred during conversion to PDF/A. Note that the function raises always fatal exceptions, there is no PDF file in memory if the conversion fails.

Remarks:

The return value of the function must not be ignored. If a required ICC profile will not be added to the file, the resulting PDF file will not be compatible to PDF/A! The PDF file must be closed with CloseFile(). The usage of CloseFileEx() is prohibited because PDF/A files must not be encrypted.

CheckConformance() can be called after the entire document was fully created. You must not call the function multiple times on the same document in memory!

Example (C++):

```cpp
// This example converts an arbitrary PDF file to PDF/A. In order to execute this code with a
// licensed version of DynaPDF, you need also a license of the PDF/A Extension. If you don't have
// such a license then don't set a license key! Otherwise the function would return with the error
// that claims that this feature is not available in your license...

// Warnings and errors are passed to the error callback function. The messages should normally be
// stored in an error log.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char* ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}
// This function will be called when a non-embedded font was not found on the system.
// DynaPDF uses already an internal mapping table for the most known good matches. However,
// some drivers write invalid font names into the file, e.g. when the PDF file was created
// from a PCL recourse. For such cases you can create your own mapping table or a default
// font.

// Since DynaPDF 3.0.49.148 the function is no longer called for the 14 PDF standard fonts
// because DynaPDF can now create Type3 versions of these fonts if no suitable system font
// is available.

SI32 PDF_CALL pdf_OnFontNotFount(const void* Data, const void* PDFFont,
    const char* FontName, SI32 Style, SI32 StdFontIndex, LBOOL IsSymbolFont)
{
   // Here you could use your own mapping table.
   // In this example we replace the font simply with Arial
   if (WeightFromStyle(Style) < 500)
   {
      // Only the weights 500 and 700 of Arial are installed by default.
      // If you have also light variants then it is not required to change the style.
      Style &= 0xF;
      Style |= fsRegular;
   }
   return pdfReplaceFont(Data, PDFFont, "Arial", Style, true);
}

SI32 PDF_CALL pdf_OnReplaceICCProfile(const void* Data, TICCProfileType Type, SI32 ColorSpace)
{
   // The ICC profiles to be used should normally be configured by the user.
   switch(Type)
   {
      case ictGray:
         return pdfReplaceICCProfile(Data, ColorSpace,
         "c:/Windows/System32/spool/drivers/color/gray_gamma2.2.icm");
      case ictRGB:
         return pdfReplaceICCProfile(Data, ColorSpace,
         "c:/Windows/System32/spool/drivers/color/sRGB Color Space Profile.icm");
      case ictCMYK:
         return pdfReplaceICCProfile(Data, ColorSpace,
         "c:/Windows/System32/spool/drivers/color/EuropeISOCoatedFOGRA27.icc");
      default: return -1;
   }
}
```

```cpp
SI32 PDFToPDFA(const void* PDF, const char* InFileName, const char* OutFileName)
{
    pdfCreateNewPDF(PDF, NULL);
    // Don't override the producer with "DynaPDF 4.0..."
    pdfSetDocInfo(PDF, diProducer, NULL);
    // The flag ifPrepareForPDFA is required!
    pdfSetImportFlags(PDF,ifImportAll | ifImportAsPage | ifPrepareForPDFA);
    // Optional but recommended to reduce the memory usage
    pdfSetImportFlags2(PDF, if2UseProxy);
    if (pdfOpenImportFile(PDF, InFileName, ptOpen, NULL) < 0)
    {
        pdfFreePDF(PDF);
        return -1;
    }
    pdfImportPDFFile(PDF, 1, 1.0, 1.0);
    switch(pdfCheckConformance(PDF, ctPDFA_1b_2005, coDefault_PDFA_1,
        PDF, // We need the instance pointer in the callback functions
        pdf_OnFontNotFount,
        pdf_OnReplaceICCProfile))
    {
        case 1:  pdfAddRenderingIntent(PDF, m_RGBProfile);  break;
        case 2:  pdfAddRenderingIntent(PDF, m_CMYKProfile); break;
        case 3:  pdfAddRenderingIntent(PDF, m_GrayProfile); break;
        default: break;
    }
    // Still a PDF file in memory?
    if (pdfHaveOpenDoc(PDF))
    {
        if (!pdfOpenOutputFile(PDF, OutFileName))
        {
            pdfFreePDF(PDF);
            return -2;
        }
        pdfCloseFile(PDF);
    }
    TPDFError msg;
    msg.StructSize = sizeof(msg);
    SI32 e, errMsgCount = pdfGetErrLogMessageCount(PDF);
    for (e = 0; e < errMsgCount; e++)
    {
        pdfGetErrLogMessage(PDF, e, &msg);
        printf("Error: %s\n", msg.Message);
    }
    return 0;
}

int main(int argc, char* argv[])
{
    // Process as many files as possible with one instance
    void* pdf = pdfNewPDF();
    if (!pdf) return -3; // Out of memory?
    pdfSetOnErrorProc(pdf, NULL, PDFError);

    pdfSetLicenseKey(pdf, "...");

    // Not required but strongly recommended. External CMaps should be available when converting
    // imported PDF files. The files are only loaded if necessary.

    // Use an absolute path if possible! Otherwise the CMaps cannot be loaded if the working
    // directory was changed.
    pdfSetCMapDir("c:/MyAppPath/Resource/CMap/", TLoadCMapFlags(lcmDelayed | lcmRecursive));

    SI32 retval = PDFToPDFA(pdf, "c:/test.pdf", "c:/cppout.pdf");

    pdfDeletePDF(pdf);
    return retval;
}
```

CheckFieldNames

Syntax:

```
SI32 pdfCheckFieldNames(
    const PPDF* IPDF) // Instance pointer
```

This function checks all currently defined interactive form fields for invalid duplicate field names. Use this function if multiple interactive forms were imported. DynaPDF does not check or change duplicate field names during import. Therefore, check the field names and change invalid names if necessary before closing the document. If an interactive form contains duplicate field names of fields with different types, the document will be damaged.

Remarks:

If a form contains invalid duplicate field names, the names must be changed with SetFieldName(). However, if such a field is used in a JavaScript action or by a global JavaScript, the JavaScript(s) must also be changed to avoid error messages in Adobe's Acrobat.

Global JavaScripts can be accessed with the function GetJavaScript() and changed with the function ChangeJavaScript(). A JavaScript action can be accessed with the function GetJavaScriptAction() and changed with the function ChangeJavaScriptAction().

Return values:

The function returns the handle of the first invalid field name that was found, that is a value greater or equal zero. If no invalid field name was found the return value is **-1**.

CircleAnnot

Syntax:

```
SI32 pdfCircleAnnot(
    const PPDF* IPDF,     // Instance pointer
    double PosX,          // y-coordinate of the annotation
    double PosY,          // x-coordinate of the annotation
    double Width,         // Width of the annotation
    double Height,        // Height of the annotation
    double LineWidth,     // Line width of the circle or ellipse
    UI32 FillColor,       // Fill color or NO_COLOR. See description
    UI32 StrokeColor,     // Stroke color or NO_COLOR. See description
    TPDFColorSpace CS,    // Color space of the fill and stroke colors
    const char* Author,   // Optional author
    const char* Subject,  // Optional subject
    const char* Comment)  // Optional comment
```

The function draws a circle annotation on the current open page. If the parameters *Width* and *Height* are equal the function draws a circle, an ellipse otherwise. If the annotation should be drawn without a border, set the parameter *LineWidth* to zero or *StrokeColor* to the special constant NO_COLOR.

If the interior should be transparent set *FillColor* to the special constant NO_COLOR.

Although the line width can be set to any positive floating point value, Adobe's Acrobat or Reader restrict the line width to 0 through 12 units. The line width should be restricted in the same way to avoid issues in Adobe viewer products.

Remarks:

This function is implemented in an Ansi and Unicode compatible variant. Ansi strings are interpreted in the Windows code page 1252.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails, the return value is a negative error code.

ClearAutoTemplates

Syntax:

```
SI32 pdfClearAutoTemplates(
   const PPDF* IPDF) // Instance pointer
```

This function deletes the array of templates which are automatically added to newly created pages. See AutoTemplate() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ClearErrorLog

Syntax:

```
void pdfClearErrorLog(
   const PPDF* IPDF) // Instance pointer
```

The clears the error log. The error log can be cleared before another function will be executed. This makes it easier to determine whether errors occurred during execution of a specific function.

The error log is always cleared when the PDF file in memory is released.

ClearHostFonts

Syntax:

```
SI32 pdfClearHostFonts(
   const PPDF* IPDF) // Instance pointer
```

This function deletes the array of fonts included in all currently defined font search paths. After the list was deleted it is still possible to use the fonts which are already in use. The 14 standard fonts are also available depending on whether the property UseStdFonts (Get/SetUseStdFonts()) is true or

false. If no font was already in use and if the property UseStdFonts is set to false it is not possible to select a font until a new font search path was added (see AddFontSearchPath()).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ClipPath

Syntax:

```
SI32 pdfClipPath(
    const PPDF* IPDF,        // Instance pointer
    TClippingMode ClipMode, // Clipping mode (Winding or Even-Odd)
    TPathFillMode FillMode) // Path fill mode
```

```
typedef enum
{
    cmEvenOdd, // Use the Even-Odd rule
    cmWinding  // Use the nonzero Winding Number rule
}TClippingMode;
```

```
typedef enum
{
    fmFillNoClose,          // not allowed, path must be closed
    fmStrokeNoClose,        // not allowed, path must be closed
    fmFillStrokeNoClose,    // not allowed, path must be closed
    fmFill,                 // fill the path (winding)
    fmStroke,               // stroke the path (winding)
    fmFillStroke,           // fill and stroke the path (winding)
    fmFillEvOdd,            // fill the path (even-odd)
    fmFillStrokeEvOdd,      // fill and stroke the path (even-odd)
    fmFillEvOddNoClose,     // not allowed, path must be closed
    fmFillStrokeEvOddNoClose, // not allowed, path must be closed
    fmNoFill,               // close but do not paint the path
    fmClose                 // same as fmNoFill
}TPathFillMode;
```

This function marks the current path as clipping path. The function must be called after a closable path was created. A path that consists of a MoveTo() and LineTo() only call cannot be closed!

A clipping path can also be filled, stroked, or both in one pass. However, the combination of a clipping path operator with a path painting operator is seldom used and not supported in all PDF viewers. To avoid unnecessary problems a path should always be clipped and painted in two separate steps, also if this causes some unnecessary overhead.

Once the clipping was created and activated with ClipPath() you can draw arbitrary contents into it, such as images, text, or vector graphics.

A clipping path is part of the current graphics state. The only way to deactivate a clipping path is to restore the graphics state with RestoreGraphicState(). This assumes that it was saved with SaveGraphicState() before the clipping path was created.

Note that it is not possible to extend or widen an active clipping path. It is only possible to intersect it with a new one. The intersection of two clipping paths is never larger than the initial clipping path.

Remarks:

Text objects are handled separately in PDF for use as clipping path. See SetTextDrawMode() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CloseAndSignFile

Syntax:

```
SI32 pdfCloseAndSignFile(
   const PPDF* IPDF,      // Instance pointer
   const char* CertFile, // File path to a PKCS#12 certificate file
   const char* Password, // Password to decrypt the cert's private key
   const char* Reason,   // Optional reason string
   const char* Location) // Optional signers location string
```

This function finishes the PDF file, digitally signs it, and frees all used resources if the file was not created in memory. The parameter *CertFile* must be a valid path to a PKCS#12 certificate file.

DynaPDF signs PDF files on Windows with the Windows Crypt API. This signature handler supports certificates with up to 4096 bits key length.

On non-Windows systems the function uses the AiCrpyto library to sign PDF files. This signature handler supports certificates with up to 1024 bits key lengths only. However, if certificates with a stronger encryption should be used it is possible to sign the file with an external signature handler (see CloseAndSignFileExt() for further information).

The parameter *Password* holds the password to decrypt the private key. The parameters *Reason* and *Location* are both optional. These strings are printed into the signature field if set.

Digital signatures are stored in a signature field which is a special type of Interactive Form field. If no signature field exist when the function is called then DynaPDF creates an invisible signature field. Hidden signature fields appear in the signatures tab of Adobe's Acrobat.

If one or more signature fields exist when the function is called then the first signature field is used for signing. Signature fields can be created with the function CreateSigField() but it is also possible to import empty signature fields from an external PDF file.

Importing signed PDF files

Signed PDF files can only be changed, without invalidating existing signatures, when changes are stored with an incremental update. An incremental update is a special way to modify a PDF file; changes are appended to the end of the file, leaving its original contents intact. This technique is required since altering any existing bytes in the file invalidates existing signatures.

However, incremental updates are not supported by DynaPDF that is the reason why only empty signature fields are imported. Because DynaPDF creates always a completely new PDF file, it makes no sense to import existing signatures, they would become invalid.

Possible function errors

When trying to sign a PDF file the function must first parse the certificate file before a signature can be created. This process can fail, due to an invalid file path or due to an unsupported certificate file format that was passed to the function. The PDF file will only be closed and signed if the certificate file was valid and if no other error occurred during the parsing process.

If an error occurred due to an invalid certificate file, invalid file path and so on, the function returns zero and passes an error message to the error callback function if any. All errors during the parsing process of the certificate file are non-fatal so that the PDF file is still in memory when the function returns. It is then possible to change the certificate file and trying to sign the PDF file again.

To determine whether the PDF was already deleted or of it is still in memory use the function HaveOpenDoc(). The function returns true if the PDF file is still in memory. If the function returns false, the last error was fatal and the PDF file has already been deleted.

The buffer of a memory based PDF file can be returned by GetBuffer() after this function was called. Note that GetBuffer() does not free the used resources because it returns a pointer to the original buffer. After the buffer was processed by your application call FreePDF() to free the uses resources.

See also CreateNewPDF(), GetBuffer(), FreePDF().

Remarks:

Further information about digital signatures can be found under Digital Signatures. To encrypt and sign a PDF file use the function CloseAndSingFileEx(). The appearance of a signature field can be user defined. See CreateSigFieldAP() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CloseAndSignFileEx

Syntax:

```
SI32 pdfCloseAndSignFileEx(
   const PPDF* IPDF,        // Instance pointer
   const char* OpenPwd,     // Optional Open password
   const char* OwnerPwd,    // Optional Owner password
   TKeyLen KeyLen,          // Key length (40 or 128 bit)
   TRestrictions Restrict,  // Restrictions
   const char* CertFile,    // File path to a PKCS#12 certificate file
   const char* Password,    // Password to decrypt the cert's private key
   const char* Reason,      // Optional reason string
   const char* Location)    // Optional signers location string
```

This function finishes the PDF file, encrypts it, digitally signs it, and frees all used resources if the file was not created in memory. The first five parameters have the same meaning as of the function CloseFileEx(). The remaining parameters are identical to CloseAndSignFile(). This function simply combines both functions because a PDF file must be encrypted and digitally signed in one pass. The encryption parameters are described in detail under CloseFileEx().

The parameter *CertFile* must be a valid path to a PKCS#12 certificate file. DynaPDF supports certificate files with 1024 bits encrypted private keys only. The parameter *Password* holds the password to decrypt the private key. The parameters *Reason* and *Location* are both optional. These strings are printed into the signature field if set.

Digital signatures are stored in a signature field which is a special type of Interactive Form field. If no signature field exist when the function is called then DynaPDF creates an invisible signature field. Hidden signature fields appear in the signatures tab of Adobe's Acrobat.

If one or more signature fields exist when the function is called then the first signature field is used for signing. Signature fields can be created with the function CreateSigField() but it is also possible to import empty signature fields from an external PDF file.

Importing signed PDF files

Signed PDF files can only be changed, without invalidating an existing signature, when changes are stored with an incremental update. An incremental update is a special way to modify a PDF file; changes are appended to the end of the file, leaving its original contents intact. This technique is required since altering any existing bytes in the file invalidates existing signatures.

However, incremental updates are not supported by DynaPDF that is why only empty signature fields are imported. Because DynaPDF creates always a completely new PDF file, it makes no sense to import existing signatures, they would become invalid.

Possible function errors

When trying to sign a PDF file the function must first parse the certificate file before a signature can be created. This process can fail, due to an invalid file path or due to an unsupported certificate file format that was passed to the function. The PDF file will be closed and signed if the certificate file was valid and no other error occurred during the parsing process of the certificate file.

If an error occurred during the parsing process, e.g. invalid certificate file, invalid file path and so on, the function returns zero and passes an error message to the error callback function if any. All errors during the parsing process of the certificate file are non-fatal so that the PDF file is still in memory when the function returns. It is then possible to change the certificate file and trying to sign the PDF file again.

To determine whether the PDF was already deleted or of it is still in memory call the function HaveOpenDoc(). The function returns true if the PDF file is in memory. If the function returns false, the last error was fatal and the PDF file has already been deleted.

The buffer of a memory based PDF file can be returned by GetBuffer() after this function was called. Note that GetBuffer() does not free the used resources because it returns a pointer to the original buffer. After the buffer was processed by your application call FreePDF() to free the uses resources.

See also CreateSigFieldAP(), CreateNewPDF(), GetBuffer(), FreePDF().

Remarks:

Further information about digital signatures can be found under Digital Signatures. The appearance of a signature field can be user defined. See CreateSigFieldAP() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CloseAndSignFileExt

Syntax:

```
LBOOL pdfCloseAndSignFileExt(
   const PPDF* IPDF,                 // Instance pointer
   struct TPDFSigParms* SigParms) // Important parameters

typedef enum
{
   htDetached, // Byte ranges of the finish PDF file will be returned
   htSHA1      // The SHA1 hash to be signed will be returned
}THashType;

struct TPDFSigParms
{
   UI32          StructSize;   // Must be set to sizeof(TSignParms)
   UI32          PKCS7ObjLen; // Max length of the signed PKCS#7 object
   THashType     HashType;     // See description
   const BYTE*   Range1;       // Out -> Hash or byte range
   UI32          Range1Len;    // Out -> Length of the buffer
   const BYTE*   Range2;       // Out -> Set only if HashType = htDetached
   UI32          Range2Len;    // Out -> Length of the buffer
   const char*   ContactInfoA; // Optional, e.g. an email address
   const UI16*   ContactInfoW; // Optional, e.g. an email address
   const char*   LocationA;    // Optional location of the signer
   const UI16*   LocationW;    // Optional location of the signer
   const char*   ReasonA;      // Optional reason why the file was signed
   const UI16*   ReasonW;      // Optional reason why the file was signed
   const char*   SignerA;      // Optional, see comment
   const UI16*   SignerW;      // Optional, see comment
   LBOOL         Encrypt;      // If true, the file will be encrypted
   // These members will be ignored if Encrypt is set to false
   const char*   OpenPwd;      // Optional open password
   const char*   OwnerPwd;     // Optional owner password
   TKeyLen       KeyLen;       // Key length to encrypt the file
   TRestrictions Restrict;     // What should be restricted?
};
```

The function closes the PDF file and returns either the file hash or the byte ranges to be signed by an external signature handler. The member *StructSize* must be initialized to sizeof(TPDFSigParms) (C/C++ only).

Most string values can be set either with an Ansi or Unicode string. If both strings are present the Ansi version takes precedence. The string *Signer* is normally taken from the PKCS#7 certificate and

not displayed in PDF viewer applications. The string is only used if the issuer of the certificate cannot be extracted.

If *Encrypt* is set to true the PDF file will also be encrypted.

Signing a PDF file with an external signature handler is a five step process:

1. Open a certificate from the certificate store or file
2. Compute the size of the signed PKCS#7 object with a dummy string
3. Call CloseAndSignFileExt() to obtain the file hash or the byte ranges to be signed
4. Sign the provided hash or byte ranges with a cryptographic library
5. Call FinishSignature() to write the signature to the PDF file and to finish the signing process.

As described above the size of the PKCS#7 object must be computed **before** the function can be called. How this must be done depends on the used signature handler. When using the Windows CryptAPI, pass a 20 bytes long dummy string to CryptSignMessage() and set the parameter *pbSignedBlob* to NULL. The size of the PKCS#7 object is then copied to the parameter *pcbSignedBlob*. When creating a detached signature the length of the dummy string can be just one byte long because the length of the PKCS#7 object does not depend on the string length.

The member *HashType* specifies whether the function should return the SHA1 hash or the byte ranges of the PDF file. In the latter case the signature handler must create a detached signature. However, when using programming languages like Visual Basic, VB. Net, or C# it is recommended to sign a SHA1 hash because it is not required to copy the PDF buffer in this case.

The hash algorithm that is used to sign the hash or byte ranges can be MD2, MD5, SHA1, SHA256, SHA384, SHA512, or RIPEMD160. MD2 and MD5 are not documented in the PDF Reference but work very well too.

Detached signatures require no data to be encapsulated in the PKCS#7 SignedData field. When signing a SHA1 hash the signature handler must store the hash in the SignedData field of the PKCS#7 object.

Supported PKCS#7 Format

PKCS #7 objects are ASN1 encoded binary objects. The ASN1 standard defines a set of Basic Encoding Rules (BER) which describe how ASN1 objects must be encoded. This Standard defines also a set of Distinguished Encoding Rules (DER) and a set of Canonical Encoding Rules (CER) both of which provide constraints on the Basic Encoding Rules (BER). The key difference between them is that DER uses the definite length form of encoding while CER uses the indefinite length form.

PDF compatible PKCS#7 objects must be DER encoded because Adobe's Acrobat or Reader does not support indefinite length encoding. However, not all cryptographic providers support definite length encoding. Whether this is the case or not is also often not documented.

The Windows CryptAPI creates DER encoded PKCS#7 objects which are fully PDF compatible. However, when using another cryptographic library and if Adobe's Acrobat or Reader reports an

error like "Error reading BER encoded object" when trying to validate the signature, the library does probably not create definite length encoded PKCS#7 objects.

Hardware Certificates

Signing a PDF file with a hardware certificate works exactly in the same way as with a software certificate as long as the certificate was installed in the certificate store of the operating system.

Hardware certificates like smart cards, USB sticks, or something similar are normally delivered with software that is able to install the certificate on the operating system. Once a certificate was installed in the certificate store it can be used like any other certificate.

The Windows CryptAPI for example, hides the certificate type fully for the developer. Whether the user selects a software or hardware certificate is simply not of interest because nothing special must be done in either case. You still call CryptSignMessage(), that's all. If a password is required the CryptAPI displays automatically a dialog to enter the password. The communication with the hardware is done in background.

Remarks:

If the function succeeds the signature must be written to the PDF file with FinishSignature(). Take a look into the example external_signatures to determine how the function can be used. It is usually best to open the the output file right before FinishSignature() will be called (see OpenOutputFile()).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CloseFile

Syntax:

```
LBOOL pdfCloseFile(
    const PPDF* IPDF) // Instance pointer
```

This function finishes the PDF file and frees all used resources if the file was not created in memory. The buffer of a memory based PDF file can be returned by GetBuffer() after this function was called. Note that GetBuffer() does not free the used resources because it returns a pointer to the original buffer. After the buffer was processed by your application call FreePDF() to free the used resources. See also CreateNewPDF(), GetBuffer(), FreePDF().

Remarks:

To encrypt a PDF file use CloseFileEx() instead of CloseFile().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CloseFileEx

Syntax:

```
LBOOL pdfCloseFileEx(
    const PPDF* IPDF,        // Instance pointer
    const char* OpenPwd,     // Open password (can be NULL)
    const char* OwnerPwd,    // Owner password (can be NULL)
    TKeyLen KeyLen,          // Key length (40 or 128 bit)
    TRestrictions Restrict)  // Flags to restrict certain features

typedef enum
{
    kl40bit    = 0, // RC4 Encryption -> PDF 1.2, Acrobat 3 or higher
    kl128bit   = 1, // RC4 Encryption -> PDF 1.4, Acrobat 5 or higher
    kl128bitEx = 2, // RC4 Encryption -> PDF 1.5, Acrobat 6 or higher
    klAES128   = 3, // AES Encryption -> PDF 1.6, Acrobat 7 or higher
    klAES256   = 4, // AES Encryption -> PDF 1.7, Acrobat 9 or higher
    klAESRev6  = 5  // AES Encryption -> PDF 2.0, Acrobat X or higher
}TKeyLen;

typedef SI32 TRestrictions;
#define rsDenyNothing      0x00000000 // Encrypt the file only
#define rsDenyAll          0x00000F3C // Deny anything
#define rsPrint            0x00000004 // Deny printing
#define rsModify           0x00000008 // Deny modification of contents
#define rsCopyObj          0x00000010 // Deny copying of contents
#define rsAddObj           0x00000020 // No commenting
/* 128/256 bit encryption only -> ignored if 40 bit encryption is used */
#define rsFillInFormFields 0x00000100 // requires rsModify + rsAddObj
#define rsExtractObj       0x00000200 // requires rsModify
#define rsAssemble         0x00000400 // requires rsModify
#define rsPrintHighRes     0x00000800 // Disable high res. printing
#define rsExlMetadata      0x00001000 // PDF 1.5 Exclude metadata streams
#define rsEmbFilesOnly     0x00002000 // PDF 1.6 AES Encryption only
```

The function finishes the PDF file, encrypts it, and frees all used resources if the file was not created in memory. The buffer of a memory based PDF file can be accessed with GetBuffer() after the function was called. Note that GetBuffer() does not free the used resources because it returns a pointer to the original buffer. After the buffer was processed by your application call FreePDF() to free the used resources. See also CreateNewPDF(), GetBuffer(), FreePDF().

The parameter *KeyLen* specifies the encryption filter and key length which should be used to encrypt the PDF file. The parameter *Restrict* specifies which features should be disabled in a viewer application. Note that several restriction flags depend on the used encryption filter. Flags which are not supported by the encryption filter will be ignored.

However, the restriction flags cannot be combined in any order; only specific flag combinations are allowed. Valid flag combinations are described in detail on the next page.

Passwords

PDF files support two different passwords to decrypt a PDF file. A PDF viewer asks for the owner password when the user tries to change the encryption settings. So, the owner password grands full access to the PDF file. If no open password is set, a viewer opens the PDF file with user privilegs without asking for a password.

If an open password is set, the PDF file cannot be opened without entering the open password. The open password causes the file to be opened with user privilegs. So, the user cannot change the security settings and only non-restricted features can be used.

Beginning with PDF 1.6 it is also possible to encrypt embedded files only (flag rsEmbFilesOnly). The PDF file is left unencrypted in this case but access to the embedded files can be restriced. The open password can be used to grand access on the embedded file to authorized persons only. A viewer asks for the password only if it is set and if the user tries to open or extract the embedded file.

Note that this feature is supported since Adobe's Acrobat or Reader 7. Prior versions cannot open such a PDF file also if it is unencrypted.

In no password is set the PDF file is still encrypted and the file is opened with user privilegs in a viewer. However, the seceurity settings can be easily changed in this case if the user tests an empty password.

Because the PDF file can be decrypted without requiring a password if no open password was set, it is easy to remove the entire encryption settings, this can be done by many tools incl. DynaPDF.

Remarks:

A PDF file can always be decrypted with a PDF library like DynaPDF if the user or owner password is not set. So, if a PDF file must be secure then both password must be set!

Password encodings

On Windows, Linux, and Unix operating systems passwords are converted to the code page 1252 (WinAnsi) and then to PDFDoc encoding. On Mac OS X or iOS passwords are converted to MacRoman and then to PDFDoc encoding. PDFDoc encoding is a superset of WinAnsi and MacRoman encoding that ensures that a password can be correctly interpreted on these operating systems.

UTF-8 Passwords

All Ansi functions accept UTF-8 Unicode strings as input if the flag gfAnsiStringIsUTF8 is set (see SetGStateFlags() for further information). However, only the encryption handlers klAES256 and klAESRev6 support Unicode passwords.

For every other encryption handler DynaPDF converts the string to the code page 1252 on Windows, Linux, or Unix operating systems, or to MacRoman on Mac OS X and iOS.

Encryption flags

As mentioned earlier only specific flag combinations are allowed to use depending on the encryption filter. We want now determine how the 40 bit encryption flags can be used in comparison to the Acrobat input mask. The encryption mask in Acrobat 9 looks as follows:
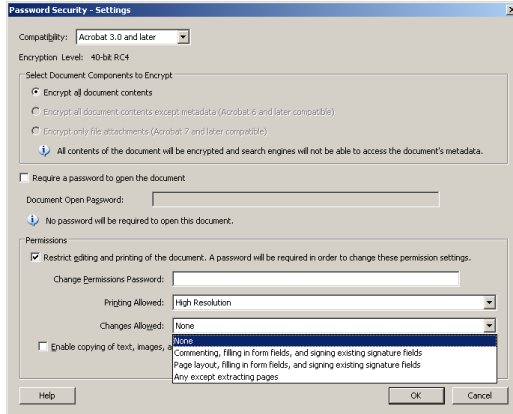
The combo box at "*Printing Allowed*" sets the flag **rsPrint** if printing should be **disabled**. The combo box at "*Changes Allowed*" sets the following flag combinations:

- *None*:   **rsModify + rsAddObj**
- *Commenting, Filling in form fields and signing*: **rsModify**
- *Page layout, filling in form fields*:   **No additional flags!**
- *Any except extracting pages*:   **No additional flags!**

As you can see above the third and forth options set the same flags. The forth value is just a duplicate of the third with another description.

The check box "*Enable copying of text, images, and other…*" sets the flag **rsCopyObj** if it is **unchecked**! Multiple flags must be combined with a binary **or** operator.

The input mask for the key lengths kl128bit (Acrobat 5) and kl128bitEx (Acrobat 6) look nearly identically. However, only the latter one supports the additional flag rsExludeMetaData.
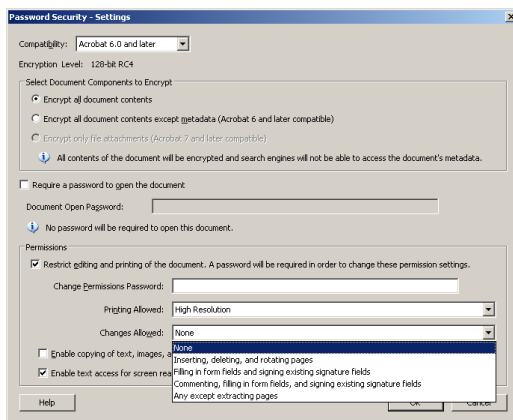
The first combo box at "*Printing Allowed*" contains now three choice values which sets the following flags:

- *None*:   **rsPrint**
- *Low Resolution (150 DPI)*: **rsPrintHighRes**
- *High Resolution:*   **No additional flags!**

The second combo box at "*Changes Allowed*" sets the following flags:

- *None*:   **rsModify + rsAddObj + rsFillInFormFields + rsAssemble**
- *Inserting, deleting, and…*: **rsModify + rsAddObj + rsFillInFormFields**
- *Filling in form field…*:   **rsModify + rsAddObj**
- *Commenting, filling in…*:   **rsModify**
- *Any except extracting…*:   **No additional flags!**

The check box "*Enable copying of text, images, and other…*" sets the flag **rsCopyObj**, if unchecked!
The check box "*Enable text access for screen readers…*" sets the flag **rsExtractObj**, if unchecked!

The key lengths klAES128 and klAES256 support also the flag rsEmbFiles only. If this flag is set all other flags are ignored becaus the PDF file is left unencrypted in this case (with exception of embedded files).

Remarks:

Due to a bug in Acrobat 6 the permission *rsPrintHighRes* is not displayed in the security settings but it is still applied.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CloseImage

Syntax:

```
LBOOL pdfCloseImage(
    const PPDF* IPDF) // Instance pointer
```

The function closes the current open image file and frees the internal used resources if the image was not created in memory. The buffer of a memory based image file can be returned by GetImageBuffer() after this function was called. Note that GetImageBuffer() does not free the internal used resources because it returns a pointer to the original buffer. After the buffer was processed by your application call FreeImageBuffer() to free the used resources. See also CreateImage(), GetImageBuffer(), FreeImageBuffer().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CloseImportFile

Syntax:

```
LBOOL pdfCloseImportFile(
    const PPDF* IPDF) // Instance pointer
```

This function closes an open import file that was be opened by OpenImportFile() or OpenImportBuffer(). If no import file was opened before the functions returns without an error.

This function cannot fail; the return value is always 1.

CloseImportFileEx

Syntax:

```
LBOOL pdfCloseImportFileEx(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle)      // File handle
```

The function closes a specific import file and deletes the corresponding parser instance. The parameter *Handle* must be a valid file handle that was returned by OpenImportFile() or OpenImportBuffer(). See OpenImportFile() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ClosePath

Syntax:

```
LBOOL pdfClosePath(
    const PPDF* IPDF,        // Instance pointer
    TPathFillMode FillMode) // see below
typedef enum
{   // Nonzero Winding Number Rule
    fmFillNoClose,              // Fill but do not close the path
    fmStrokeNoClose,            // Stroke but do not close the path
    fmFillStrokeNoClose,        // Fill and stroke but do not close the path
    fmFill,                     // Fill the path
    fmStroke,                   // Stroke the path
    fmFillStroke,               // Fill and stroke the path
    // Even-Odd Rule
    fmFillEvOdd,                // Fill the path
    fmFillStrokeEvOdd,          // Fill and stroke the path
    fmFillEvOddNoClose,         // Fill but do not close the path
    fmFillStrokeEvOddNoClose,   // Fill and stroke but do not close the path
    fmNoFill,                   // Discard the path
    fmClose                     // Close the path and begin a new sub path
}TPathFillMode;
```

In PDF all vector graphics are paths. Paths can be filled, stroked, or both. Filled path can be drawn by applying the nonzero winding number rule or the even-odd rule (see Path Construction and Painting for further information). Both rules produce different results on complex paths but they have no effect on simple paths like rectangles, circles, and so on.

The filling rules are can be used to produce holes in a path; an area that is left unpainted. The draw direction is important when drawing simple shapes like rectangles and so on. See SetDrawDirection() for further information.

The flag fmNoFill can be used to discard a path that was already drawn.

Remarks:

Paths are invisible until they will be filled or stroked. DynaPDF displays a warning if an unused path is in memory when drawing an image, text, or when closing the page. The function checks whether a path is in memory prior execution. If no open path is detected the function returns with an error message.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CloseTag

Syntax:

```
LBOOL pdfCloseTag(
  const PPDF* IPDF) // Instance pointer
```

The function closes the current open tag that was opened by OpenTag() beforehand.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ComputeBBox

Syntax:

```
LBOOL pdfComputeBBox(
  const PPDF* IPDF,          // Instance pointer
  struct TPDFRect ADDR BBox, // TPDFRect structure
  TCompBBoxFlags Flags)      // See below
typedef UI32 TCompBBoxFlags;
#define cbfNone             0x00000000
#define cbfIgnoreWhiteAreas 0x00000001 // Ignore white areas
#define cbfParse1BitImages  0x00000002 // Process 1 bit images
#define cbfParseGrayImages  0x00000004 // Process gray images
#define cbfParseColorImages 0x00000008 // Process color images
#define cbfParseAllImages   0x0000000E // Process all images
```

The function computes the visible area of the current opened page. If images are processed, DynaPDF decompresses the images and computes the visible area by checking for non-white pixels. Images which use a complex color space like DeviceN or Separation are converted into the alternate color space beforehand. Please note that every non-white pixel is considered as part of the visible area.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ConnectPageBreakEvent

Syntax:

```
Public Sub ConnectPageBreakEvent(
  ByVal Connect As Boolean) // If True, the event is connected
```

This procedure connects the OnPageBreakEvent with the internal callback function so that the event function can be raised by the wrapper class CPDF. This function is included in the Visual Basic and VB .Net interfaces only. The manually connection of the event function is required to avoid different runtime behaviour of the function WriteFText().

ConvColor

Syntax:

```
UI32 pdfConvColor(
  const double* Color,     // Array of color components
  UI32 NumComps,           // Number of color components
  TExtColorSpace SourceCS, // Source color space
  const void* IColorSpace, // Pointer to internal color space object
  TExtColorSpace DestCS)   // Destination color space
```

The function converts a color of a PDF color space to a device color space. The function was originally designed to convert colors of a PDF color space which was returned by the content parser, but it is also possible to use the function stand alone, e.g. to calculate a spot color of a DeviceN or Separation color space.

If used with the content parser, the required parameters *Color*, *NumComps*, *SourceCS*, and *IColorSpace* are provided in the TSetFillColor or TSetStrokeColor callback functions. See ParseContent() for further information.

The parameter *DestCS* defines the destination color space into which the color should be converted. The destination color space must be a device color space (esDeviceGray, esDeviceRGB, or esDeviceCMYK).

If the function should be used stand alone, the parameters *SourceCS* and *IColorSpace* are provided in the TPDFColorSpaceObj structure, see GetColorSpaceObj() for further information.

The array *Color* represents a color defined in the PDF color space. The array must be at least NumInComponents values long. The number of color components is provided in the TPDFColorSpaceObj structure.

The color components must be defined in the range of the color space, that is typically 0 through 1 for Device, ICCBased, Calibrated, DeviceN, NChannel, and Separation color spaces.

The default range of a Lab color space is [0..100, -128..127, -128..127] . The range of the L channel is fixed, it ranges always from 0 though 100. The range of the a and b components can be restricted if the color space contains a Range array.

The range of an Indexed color space is 0 through NumColors – 1. In any case, values which fall outside the valid range will be adjusted to the nearest valid value.

In most cases the function is used to calculate the color of spot colorants. Spot colors can be defined in Separation, DeviceN, and NChannel color spaces. NChannel is an extended DeviceN color space that contains always attributed while they are optional for DeviceN color spaces. The attributes dictionary of a DeviceN or NChannel color space lists all spot colorants in an array of Separation color spaces (see GetColorSpaceObj()). Use the Separation colorspaces to calculate the color values in this case. If a DeviceN color space contains no attributes dictionary then it is still possible to enumerate and calculate the color values of spot colorants which are used by the color space but it is some more work.

DeviceN and Separation color spaces are subtractive color spaces. Thus, a tint value of 0.0 denotes the lightest color that can be achieved with a given colorant, and 1.0 denotes the darkest.

If you want to calculate the color of a spot colorant then initialize the color channel that corresponds to the spot colorant to 1.0 and the remaining channels to 0.0, if any.

Now it is important to know what is a spot colorant? A Separation color space supports two special colorant names: All and None. The value All refers to all channels of the output device and None produces no visible output.

DeviceN supports 5 reserved values: the colorant names Cyan, Magenty, Yellow, and Black are always treated as device colorants, and the special colorant name None produces no visible output.

All other colorant names are considered as spot colorants. Colorant names must be compared case-sensitive.

So, if you want to calculate the color value of a spot colorant, then you must first compare the colorant names against the reserved or special colorant names of the color space. If a spot color was found, initialize that color channel to 1.0 and all others to 0.0. Continue until all colorants of the color space were visited, see example below.

Return values:

If the function succeeds the return value is the color value defined in the destination color space. If the function fails the return value is zero without further error indication. However, the function cannot fail if valid values were passed to the function.

Example (C++):

```cpp
...
// Returns true if the colorant is a spot colorant
bool IsSpotColor(const char* Colorant, bool Separation)
{
    if (Separation)
        return (strcmp(Colorant, "None") && strcmp(Colorant, "All"));
    else
        return (strcmp(Colorant, "Cyan")   && strcmp(Colorant, "Magenta")
            && strcmp(Colorant, "Yellow") && strcmp(Colorant, "Black")
            && strcmp(Colorant, "None"));
}
// Helper function to process the spot colors in DeviceN, NChannel, and
// Separation color spaces
void ProcessColorSpace(TPDFColorSpaceObj &CS)
{
    UI32 i, color;
    double inClr[32] = {0}; // More than 32 channels cannot occur!
    for (i = 0; i < CS.ColorantsCount; i++)
    {
        if (IsSpotColor((char*)CS.Colorants[i], CS.Type == esSeparation))
        {
            inClr[i] = 1.0;
            // Do something with the color...
            color = pdfConvColor(inClr,
                                    CS.NumInComponents,
                                    CS.Type,
```

```
                                        CS.IColorSpaceObj,
                                        esDeviceCMYK);
            inClr[i] = 0.0;
        }
    }
}

void EnumSpotColors(const void* PDF)
{
    UI32 j;
    SI32 i, count;
    TPDFColorSpaceObj cs;
    TDeviceNAttributes at;
    count = pdfGetColorSpaceCount(PDF);
    for (i = 0; i < count; i++)
    {
        if (pdfGetColorSpaceObj(PDF, i, &cs))
        {
            switch(cs.Type)
            {
                case esDeviceN:
                case esNChannel:
                {
                    if (cs.IAttributes)
                    {
                        // Process the separation color spaces directly
                        if (pdfGetDeviceNAttributes(cs.IAttributes, &at))
                        {
                            for (j = 0; j < at.SeparationsCount; j++)
                            {
                                if (pdfGetColorSpaceObjEx(at.Separations[j],&cs))
                                    ProcessColorSpace(cs);
                            }
                        }
                        continue;
                    }
                }
                case esSeparation: ProcessColorSpace(cs); continue;
                default:                                          continue;
            }
        }
    }
}
...
```

ConvertColors

Syntax:

```
LBOOL pdfConvertColors(
  const PPDF* IPDF,       // Instance pointer
  TColorConvFlags Flags,  // See below
  const float* Add)       // Optional, see description below

typedef enum
{
   ccfBW_To_Gray  = 0, // Default, RGB B/W to gray
   ccfRGB_To_Gray = 1, // RGB to gray
   ccfToGrayAdjust = 2  // Lighten or darken colors
}TColorConvFlags;
```

The function converts inline RGB operators in the current open page to gray. At time of publication only inline operators like g, rg, G, and RG are processed. CMYK colors will be left unchanged. The parameter *Add* is optional and considered only if the constant ccfToGrayAdjust is used.

The conversion of RGB inline color operators can be useful if the document was created with a GDI application, e.g. Microsoft Word, Excel and so on and if the file must be split into black & white and color pages. IsColorPage() can then be used to determine whether the page contains still colored objects. The advantage is that RGB black & white values, which are used for text and vector graphics, are already removed so that IsColorPage() returns only true if any other object uses a color, e.g. an image.

The constant ccfToGrayAdjust can be used to convert RGB colors to gray and to darken or lighten RGB and gray colors. The parameter *Add* must be a pointer of a floating point variable. The value can be in the range -1.0 through 1.0. Colors in PDF are stored as floating point values in the range 0.0 through 1.0. Since DeviceGray is an additive color space, 0 denotes black and 1.0 white. If all color operators should be set to black for example, then set the variable to -1.0. The function performs a range check. If the color becomes out of range then it will be adjusted to the nearest value that is inside the allowed color range.

Example (C++):

```
// This example converts all RGB and gray colors to black.
...
pdfEditPage(PDF, 1);
float add = -1.0;
pdfConvertColors(PDF, ccfToGrayAdjust, &add);
pdfEndPage(PDF);
...
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ConvertEMFSpool

Syntax:

```
SI32 pdfConvertEMFSpool(
    const PPDF* IPDF,       // Instance pointer
    const char* SpoolFile, // The spool file that should be converted
    double LeftMargin,      // Additional margin
    double TopMargin,       // Additional margin
    TSpoolConvFlags Flags) // see below
typedef enum
{
    spcDefault           = 0,
    spcIgnorePaperFormat = 1, // See description
    spcDontAddMargins    = 2, // See description
    spcLoadSpoolFontsOnly = 4, // Load embedded fonts only
    spcFlushPages        = 8  // Write every page to the output file
}TSpoolConvFlags;
```

The function converts an EMF spool file to PDF. The function loads also embedded fonts which are required to sucessfully convert the embedded EMF files. The function does not create a new PDF file and it does not close the PDF file after all pages were converted. The EMF pages will be added to the end of the exiting pages of the PDF file in memory (see example below the function description).

Before calling the function, the page format of the first page should be set with SetBBox() or with SetPageWidth() / SetPageHeight(). The initial page format is usually taken from the DEVMODE structure of the print job. The current page format is used by default to calculate margins which are maybe required.

The current orientation is set to all pages which will be added (see SetOrientation()). The current color space can be set to any device color space. If the color space is set to DeviceGray or DeviceCMYK then all colors will be converted to that color space. If the current color space is set to a non-device space, then it will be changed to DeviceRGB.

Flag	Description
spcIgnorePaperFormat	If set, all pages are scaled to the width of the current page format.
spcDontAddMargins	If set, the page format is calculated from the rclFrame rectangle of each EMF file as usual but the current page format is not used to calculate margins which are maybe required. The parameters *Left-* and *TopMargin* will still be considered.
	This flag can be set if the margins for the print spool are known. It is meaningful only if the flag spcIgnorePaperFormat is absent.
spcLoadSpoolFontsOnly	If set, the function loads only embedded fonts and does not convert any EMF file. This flag can be useful if you want to use your own spool file parser and if the parser is not able to load embedded fonts. The flags mfUseUnicode and mfIgnoreEmbFonts must be set when converting the EMF files of the spool file.

Flag	Description
spcFlushPages	If set, the function writes every page directly to the output file to reduce the memory usage. This flag is meaningful only if the PDF file is not created in memory. Note also that it is not possible to access already flushed pages again with EditPage().

Spool files contain sometimes subset fonts and corresponding delta fonts which must be merged and converted back to regular TrueType fonts. DynaPDF supports these font formats natively. Microsoft's fontsub.dll is not used for the conversion. This makes it possible to use the function also on non-Windows operating systems like Linux, Unix, or Mac OS X.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Unicode paths are converted to UTF-8 on non-Windows operating systems.

Please note that spool files can be very large. Since the dynapdfm.dll comes without 64 bit file support, the spool file cannot exceet the 2 GB limit. C or C++ developers should use the regular dynapdf.dll instead. Customers with a DynaPDF Enterprise license can compile the Multithreaded DLL variant with Visual Studio 2005 or higher to enable 64 bit file support. Note that the library depends then on the Visual Studio Redistributable package of the used VS version.

Return values:

The return value depends on the flag spcLoadSpoolFontsOnly. If the flag is set, the function returns the number of processed font records on success, or the number of converted pages otherwise. If the function fails, the return value is a negative error code.

Example (C++):

```cpp
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode,
                       const char* ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0; // We try to continue
}

SI32 ConvertSpoolFile(const void* PDF, const char* FileName,
                      const char* OutFile)
{
   if (pdfCreateNewPDF(PDF, OutFile) < 0) return -3;
   // In this example we don't need to access the converted pages.
   // We can set the flag spcFlushPages to reduce the memory usage.
   pdfConvertEMFSpool(PDF, FileName, 0.0, 0.0, spcFlushPages);
   // The output intent represents the destination color space for which
   // the file was created. It makes sure that we get consistent color
   // results in different PDF viewers and when printing the PDF file.
   pdfAddOutputIntent(PDF, "../icc/sRGB.icc"); // Optional but recommended
   pdfCloseFile(PDF);
```

```c
}

int main(int argc, char* argv[])
{
   if (argc < 2) return -1;
   void* pdf = pdfNewPDF();
   if (!pdf) return -2; // Out of memory?
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   // The resolution is used for images. 300 or 600 DPI are recommended.
   pdfSetResolution(pdf, 600);
   pdfSetCompressionFilter(pdf, cfFlate); // Default
   pdfSetColorSpace(pdf, csDeviceRGB);    // Default
   pdfSetPageCoords(pdf, pcTopDown);       // Recommended
   // The page format should be set to the value of the DEVMODE structure.
   pdfSetPageFormat(pdf, pfDIN_A4);
   return ConvertSpoolFile(pdf, argv[1]);
}
```

ConvToUnicode

Syntax:

```c
UI16* pdfConvToUnicode(
   const PPDF* IPDF,    // Instance pointer
   const char* AString, // Null-terminated string
   TCodepage CP)        // Used base encoding

typedef enum
{
  cp1250,              // Eastern European
  cp1251,              // Cyrillic
  cp1252,              // Latin 1 Western European
  cp1253,              // Greek
  cp1254,              // Turkish
  cp1255,              // Hebrew
  cp1256,              // Arabic
  cp1257,              // Baltic
  cp1258,              // Vietnamese
  cp8859_2,            // Latin 2 Central Europe
  cp8859_3,            // Latin 3 Maltese
  cp8859_4,            // Baltic
  cp8859_5,            // Cyrillic
  cp8859_6,            // Arabic
  cp8859_7,            // Greek
  cp8859_8,            // Hebrew
  cp8859_9,            // Latin 5 Turkish
  cp8859_10,           // Latin 6 Nordic Area
  cp8859_13,           // Latin 7 Baltic Rim
  cp8859_14,           // Latin 8 Celtic
  cp8859_15,           // Latin 9 French
  cp8859_16,           // Latin 10 Hungarian
```

```
  cpSymbol,              // Symbol
  cp437,                 // DOS USA
  cp737,                 // DOS Greek
  cp775,                 // DOS Baltic Rim
  cp850,                 // DOS Multilingual
  cp852,                 // DOS Slavic
  cp855,                 // DOS Cyrillic
  cp857,                 // DOS Turkish
  cp860,                 // DOS Portuguese
  cp861,                 // DOS Icelandic
  cp862,                 // DOS Hebrew
  cp863,                 // DOS French (Canada)
  cp864,                 // DOS Arabic
  cp865,                 // DOS Nordic
  cp866,                 // DOS Russian
  cp869,                 // DOS Modern Greek
  cp874,                 // DOS Thai
  cpUnicode,             // We convert to Unicode -> NOT ALLOWED!
  cpCJK_Big5_Uni,        // Big5 plus HKSCS extension
  cpCJK_EUC_JP_Uni,      // EUC-JP
  cpCJK_EUC_KR_Uni,      // EUC-KR
  cpCJK_EUC_TW_Uni,      // CNS-11643-1992 (Planes 1-15)
  cpCJK_GBK_Uni,         // MS code page 936 (GB2312, EUC-CN plus GBK)
  cpCJK_GB12345_Uni,     // GB-12345-1990 (Trad. Chinese form of GB-2312)
  cpCJK_HZ_Uni,          // Mixed ASCII / GB-2312 encoding
  cpCJK_2022_CN_Uni,     // ISO-2022-CN-EXT (GB-2312 plus ISO-11643)
  cpCJK_2022_JP_Uni,     // ISO-2022-JP
  cpCJK_2022_KR_Uni,     // ISO-2022-KR
  cpCJK_646_CN_Uni,      // ISO-646-CN (GB-1988-80)
  cpCJK_646_JP_Uni,      // ISO-646-JP (JIS_C6220-1969-RO)
  cpCJK_IR_165_Uni,      // ISO-IR-165 (extended version of GB-2312)
  cpCJK_932_Uni,         // Microsoft extended version of SHIFT_JIS
  cpCJK_949_Uni,         // EUC-KR extended with UHC (Unified Hangul Codes)
  cpCJK_950_Uni,         // Microsoft extended version of Big5
  cpCJK_JOHAB_Uni,       // JOHAB
  // These are character sets, not code pages. The character sets can
  // not be used for Unicode conversion.
  cpShiftJIS,            // Use cpCJK_932_Uni or cpCJK_EUC_JP_Uni instead.
  cpBig5,                // Use cpCJK_Big5_Uni instead.
  cpGB2312,              // Use cpCJK_GBK_Uni instead.
  cpWansung,             // Use cpCJK_949_Uni or cpCJK_JOHAB_Uni instead.
  cpJohab                // Use cpCJK_JOHAB_Uni instead.
  cpMacRoman             // Mac Roman encoding
}
```

This function converts a string of a specific 8 bit or CJK encoding to Unicode. It can be used especially to convert strings for use with bookmarks or other global objects such as text annotations and so on which do not support 8 bit encodings but Unicode.

The function can be used multiple times without causing a memory leak. The internal used conversion buffer is freed automatically before the function is executed. However, the buffer can be freed manually with the function FreeUniBuf().

Remarks:

This function depends not on an open PDF file; it can be used at any time. The conversion algorithms are binary save but because no string length is used the string must not contain null characters (except to terminate the string). No error messages are displayed if this function fails.

Return values:

If the function succeeds the return value is a pointer to null-terminated Unicode string in the byte ordering supported by the CPU. If the function fails the return value is NULL.

CopyChoiceValues

Syntax:

```
LBOOL pdfCopyChoiceValues(
   const PPDF* IPDF, // Instance pointer
   UI32 Source,      // Source field handle
   UI32 Dest,        // Destination field handle
   LBOOL Share)      // If true, both fields share the same list items
```

The function copies the list items, default value, and value of the source field to the destination field. The source and destination field can be combo and list boxes. It is allowed to copy the list items of a list box to a combo box or vice versa.

If *Share* is true, the function adds just a reference to the list item so that both fields share the same list items. This is still the case if more list items will be added to such a field. If *Share* is false, the list items will be copied to the destination field. Both fields contain independent list items in this case.

Already existing list items of the destination field will be deleted.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Create3DAnnot

Syntax:

```
SI32 pdfCreate3DAnnot(
   const PPDF* IPDF,      // Instance pointer
   double PosX,           // X-Coordinate of the annotation
   double PosY,           // Y-Coordinate of the annotation
   double Width,          // Width of the annotation
   double Height,         // Height of the Annotation
   const char* Author,    // Optional author
   const char* Name,      // Optional annotation name
   const char* U3DFile,   // Required U3D or PRC file
   const char* Image)     // Optional image to display the closed annotation
```

The function creates a 3D annotation from a U3D or PRC file. The image is used to create the default appearance of the annotation; it is shown when the annotation is deactivated or when viewing the file with a viewer that does not support 3D annotations. If no image file is provided the background is filled with the current field background color (see SetFieldBackColor()). Note that the default background color is transparent, that means no background will be drawn if the color was not changed beforehand.

The annotation's name can be used to access the annotation from a JavaScript or JavaScript action. The name must be unique if defined. Make sure that no other annotation exists with the same name. Note that DynaPDF performs no duplicate check during creation of the annotation.

A 3D annotation requires usually at least a default view that defines the rendering mode and lighting scheme to get suitable results (see Create3DView()). Without a view the viewer application uses default settings to display the annotation.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

Create3DBackground

Syntax:

```
LBOOL pdfCreate3DBackground(
    const PPDF* IPDF,  // Instance pointer
    const void* IView, // 3D View handle
    UI32 BackColor);   // RGB background color
```

The function creates a background dictionary for a 3D annotation that is used to change the annotation's background color. The background color must be defined in RGB color space.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Create3DGotoViewAction

Syntax:

```
SI32 pdfCreate3DGotoViewAction(
    const PPDF* IPDF,      // Instance pointer
    UI32 Base3DAnnot,      // 3D Annotation handle
    const void* IView,     // 3D View handle or
    T3DNamedAction Named)  // Named 3D action

typedef enum
{
    naDefault,
    naFirst,
    naLast,
    naNext,
    naPrevious
}T3DNamedAction;
```

A GoTo3DView action can be used to dynamically change the view of a 3D annotation. If the parameter *IView* is not NULL the created action opens explicitly the defined view. The parameter *Named* is ignored in this case. If *IView* is NULL, the named action is used instead.

A GoTo3DView action can be used like any other action. The action can be added with AddActionToObj() to a bookmark, page link, or interactive form field, e.g. a button field.

Remarks:

Due to certain bugs in the 3D rendering engine of Adobe's Acrobat Reader versions prior to 7.0.9, go to 3D view actions do not properly work in these Acrobat Reader versions. The bugs were solved in Acrobat Reader 7.0.9. Make sure that your customers use the latest version.

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

Create3DProjection

Syntax:

```
LBOOL pdfCreate3DProjection(
   const PPDF* IPDF,        // Instance pointer
   const void* IView,       // 3D View handle (required)
   T3DProjType ProjType,    // Projection type
   T3DScaleType ScaleType,  // Scaling type
   double Diameter,         // Scale factor, see below
   double FOV)              // Field of View, see below

typedef enum
{
   pt3DOrthographic, // Orthographic projection
   pt3DPerspective   // Perspective projection
}T3DProjType;

typedef enum
{
   st3DValue,  // Default for orthographic projection, see description
   st3DWidth,  // Default for perspective projection
   st3DHeight,
   st3DMin,
   st3DMax
}T3DScaleType;
```

The function creates a projection dictionary for a 3D annotation. A projection dictionary defines the mapping of 3D camera coordinates onto the target coordinate system. A projection dictionary can only be added to a 3D view of a 3D annotation. The 3D view must be created beforehand with Create3DView().

The parameter *ProjType* defines the kind of projection that should be used, e.g. orthographic or perspective projection. Depending on the projection type the parameters *ScaleType*, *Diameter*, and *FOV* have different meanings:

Perspective projection

The parameter *ScaleType* specifies the scaling used when projecting the 3D artwork onto the annotation's target coordinate system. If *ScaleType* is set to st3DValue the parameter *Diameter* defines the diameter of the circle formed by the intersection of the near plane and the cone specified by *FOV*. *Diameter* must be a positive number in this case. If *ScaleType* is set to another value than st3DValue the parameter *Diameter* is ignored.

The parameter *FOV* must be a number between 0.0 and 180.0, inclusive, specifying the field of view of the virtual camera in degrees. It defines a cone in 3D space centered on the z axis and a circle where the cone intersects the near clipping plane. The circle, along with the value of *Diameter* and *ScaleType*, specify the scaling of the projected artwork when rendered in the 2D plane of the annotation. See also PDF Reference 1.7.

Orthographic projection

The parameter *ScaleType* specifies the strategy for scaling to fit the near plane's x and y coordinates onto the annotation's target coordinate system. The value st3DValue refers to absolute scaling, that means no scaling should occur due to binding. The parameter *Diameter* is ignored if orthographic projection is used.

The parameter *FOV* specifies the scale factor to be applied to both the x and y coordinates when projecting onto the annotation's target coordinate system (the z coordinate is discarded). The default value is 1.0.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Create3DView

Syntax:

```
void* pdfCreate3DView(
   const PPDF* IPDF,      // Instance pointer
   UI32 Base3DAnnot,      // 3D Annotation handle
   const char* Name,      // Unique name that describes the view (required)
   LBOOL SetAsDefault,    // If true, the view is used as default view
   double* Matrix,        // Optional 3D transformation matrix
   double CamDistance,    // Camera distance or 0.0 if not known
   T3DRenderingMode RM,   // 3D Rendering mode
   T3DLightingSheme LS)   // 3D Lighting scheme

typedef enum
{
   rmBoundingBox,
   rmHiddenWireframe,
   rmIllustration,
   rmShadedIllustration,
   rmShadedVertices,
   rmShadedWireframe,
   rmSolid,
   rmSolidOutline,
   rmSolidWireframe,
   rmTransparent,
   rmTranspBBox,
   rmTranspBBoxOutline,
   rmTranspWireframe,
   rmVertices,
   rmWireframe,
   rmNotSet
}T3DRenderingMode;
```

```
typedef enum
{
   lsArtwork, // Lights from file
   lsBlue,
   lsCAD,
   lsCube,
   lsDay,
   lsHard,
   lsHeadlamp,
   lsNight,
   lsNoLights,
   lsPrimary,
   lsRed,
   lsWhite,
   lsNotSet
}T3DLightingSheme;
```

The function creates a 3D view for a 3D annotation. A 3D annotation can contain multiple 3D views while one 3D view should always be marked as default view. Note that a default view does not appear in the list of views in the model tree of Adobe's Acrobat.

A 3D view must always define a unique name that describes the view, e.g. Left, Right, Top, or any other string. Note that the name is shown in the model tree, it should be human readable.

The parameter *Matrix* can be used to transform the 3D coordinate space, e.g. to rotate the 3D artwork. The matrix is optional, if set, it must consist of an array of double that contains exactly 12 entries. A 3D matrix is usually created interactively in a 3D viewer application. Without a visual feedback it is practically impossible to define the matrix.

The parameter *CamDistance* specifies a distance in the camera coordinate system along the z axis to the center of orbit for this view. If zero, the viewer application must determine the center of orbit. See PDF Reference 1.7 for further information.

Notice:

If a correct 3D matrix cannot be calculated in your application then set the parameter *Matrix* to NULL and **NOT** to the identity matrix. The viewer application tries then to fit the 3D artwork into the annotation's bounding box.

The last two parameters *RM* and *LS* define the 3D rendering mode and lighting scheme to be used to render the 3D artwork.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is a pointer to the view. If the function fails the return value is NULL.

CreateAltFontList

Syntax:

```
SI32 pdfCreateAltFontList(
    const PPDF* IPDF) // Instance pointer
```

The function creates an alternate font list, that can be filled with data by SetAltFonts(). It is possible to create an arbitrary number of alternate font lists. Multiple lists can be created because different fonts and languages require different fallback fonts. For example, one could create one list for proportional fonts and another one for fixed pitch fonts, or different lists for different languages.

Each list can be activated or deactivated with ActivateAltFontList() arbitrary often. Changing the active alternate font list is very fast and causes no overhead.

Once a font list was created it can be filled with data with SetAltFonts(). A font list that is no longer needed can be deleted with DeleteAltFontList().

Please note that alternate font lists are used only, if complex text layout was enabled. This can be done by setting the flag *gfComplexText* with SetGStateFlags().

Return values:

If the function succeeds the return value is the handle of the font list, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateAnnotAP

Syntax:

```
SI32 pdfCreateAnnotAP(
    const PPDF* IPDF, // Instance pointer
    UI32 Annot)       // Annotation handle
```

The function creates a user defined appearance stream for an annotation and activates it so that arbitrary contents can be drawn into it, such as text, images, or vector graphics. An appearance stream is a normal template in the size of the annotation. The template must be closed when finish with EndTemplate().

Note that the template is reserved for the annotation and must not be used on pages or other templates.

At time of publication user defined appearance streams can be created for Stamp and FreeText annotations. However, the main application is the creation of user defined stamps. The appearance stream of a FreeText annotation should correspond to the annotation's value.

It is allowed to insert an EMF file into a user defined appearance stream. If the contents should be imported from an external PDF file then import the wished page with ImportPage() and place the resulting template into this one with PlaceTemplate() or PlaceTemplateEx(). The latter version is preferred because it considers the original page orientation and bounding boxes. See PlaceTemplateEx() for further information.

Return values:

If the function succeeds the return value is the handle of the template that was created for the annotation, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateArticleThread

Syntax:

```
SI32 pdfCreateArticleThread(
    const PPDF* IPDF,         // Instance pointer
    const char* ThreadName)   // Name of the thread
```

This function creates a new article thread. An article thread is a container for articles which must be created separately with AddArticle(). A document can contain an arbitrary count of article threads. The names of the threads should be unique otherwise it is not possible to distinguish between them in Adobe's Acrobat.

Some types of documents may contain sequences of content items that are logically connected but not physically sequential. For example, a news story may begin on the first page of a newsletter and run over onto one or more non-consecutive interior pages. To represent such sequences of physically discontinuous but logically related items, a PDF document may define one or more articles.

The sequential flow of an article is defined by an article thread; the individual content items that make up the article are called beads on the thread. PDF viewer applications such as Adobe's Acrobat provide navigation facilities to allow the user to follow a thread from one bead to the next.

Remarks:

Due to a bug in Acrobat 6 articles can cause a zoom out. The pages of a document appear then as small thumb nails. It is highly recommended to test your articles with Acrobat 6.

This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252 only. To create an article thread in an arbitrary encoding convert the string to Unicode with the function ConvToIncode() first and use the Unicode version to create the article thread.

Return values:

If the function succeeds the return value is the handle of the article thread, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateAxialShading

Syntax:

```
SI32 pdfCreateAxialShading(
    const PPDF* IPDF, // Instance pointer
    double sX,        // X-Coordinate of the start point
    double sY,        // Y-Coordinate of the start point
    double eX,        // X-Coordinate of the end point
    double eY,        // Y-Coordinate of the end point
    double SCenter,   // Shading center
    UI32 SColor,      // Start color
    UI32 EColor,      // End color
    LBOOL Extend1,    // Extend the shading beyond the start point
    LBOOL Extend2)    // Extend the shading beyond the end point
```

Axial shadings define a color blend or gradient that varies along a linear axis between two endpoints and extends indefinitely perpendicular to that axis. The shading may optionally be extended beyond either or both endpoints by continuing the boundary colors indefinitely.

The shading center defines the point from where the first color will blend into the other. A value of 1 determines the exact center between the starting and ending point of the shading. Smaller values shift the shading center in direction to the start point, greater values in the direction to the end point.

Axial shadings can be drawn into a clipping path to restrict painting into this path. If the shading is drawn outside of a clipping path it is applied to the entire page. Not that extended shadings are opaque, objects behind the shading becomes invisible if they are overprinted by the shading.

Shadings are drawn by using the current coordinate system. It is recommended to understand that shadings have its own dimension like a normal shape. The parameters *Extend1* and *Extend2* extend the shading beyond its dimension. If the shading is extended it must normally be drawn into a clipping path to avoid overprinting of other objects.

Shadings support the color spaces DeviceGray, DeviceRGB, and DeviceCMYK. The color values of the start and end color must be defined in the current color space. See also SetColorSpace().

Shadings are invisible as long they are not drawn by using the function ApplyShading(). ApplyShading() requires a shading handle that was returned by this function.

Remarks:

See the function ApplyShading() for an example application.

Return values:

If the function succeeds the return value is the handle of the shading, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateBarcodeField

Syntax:

```
SI32 pdfCreateBarcodeField(
    const PPDF* IPDF,              // Instance pointer
    const char* Name,             // Field name
    SI32 Parent,                  // Parent field or -1
    double PosX,                  // X-Coordinate of the barcode field
    double PosY,                  // Y-Coordinate of the barcode field
    double Width,                 // Width in unscaled units
    double Height,                // Height in unscaled units
    struct TPDFBarcode* Barcode)  // see below

struct TPDFBarcode
{
    UI32       StructSize;    // Must be set to sizeof(TPDFBarcode)
    const char* CaptionA;     // Optional
    const UI16* CaptionW;     // Optional
    float       ECC;          // 0..8 for PDF417, or 0..3 for QRCode
    float       Height;       // Height in inches (field height / 72.0)
    float       nCodeWordCol; // Number of codewords per barcode coloumn
    float       nCodeWordRow; // Codewords per barcode row (PDF417)
    UI32        Resolution;   // Resolution
    const char* Symbology;    // Must be PDF417, QRCode, or DataMatrix
    float       Version;      // Should be 1
    float       Width;        // Width in inches (field width / 72.0)
    float       XSymHeight;   // Only needed for PDF417. The vertical distance between two barcode
                              // modules, measured in pixels. The ratio XSymHeight / XSymWidth shall
                              // be an integer value. For PDF417, the acceptable ratio range is from
                              // 1 to 4. For QRCode and DataMatrix, this ratio shall always be 1.
    float       XSymWidth;    // The horizontal distance, in pixels, between two barcode modules.
};
```

The function creates a barcode field.

The structure *TPDFBarcode* must be fully initialized with correct values for the wished barcode type. An example for the PDF417 barcode is provided below the function description. For other barcodes please consult the correcponding barcode reference.

The value of a barcode field is normally dynamically set with a JavaScript action that is associated with the OnCalc event of the barcode field. It is also possible to use a global JavaScript function or other events depending on when the barcode should be updated or how the value must be calculated.

Because DynaPDF does not contain a barcode library it is recommended to set the the NeedAppearance flag of the AcroForm to true (see SetNeedAppearance()). Alternatively barcodes can be initialized with a JavaScript action, e.g. in the pages OnOpen event. Without initialisation the barcode field appears as gray box on the page.

Barcode field require a specific set of field flags which must be present:

flags = ffPrint | ffReadOnly | ffNoExport | ffMultiline | ffDoNotSpellCheck;

DynaPDF sets the above flags when creating the field. Changing the field flags can cause errors in Adobe's Acrobat or prevent that the barcode becomes rendered.

The parameter *Name* must be a unique name for the barcode field.

If the coordinate system is bottom-up the point *PosX, PosY* defines the lower left corner of the barcode field. If the coordinate system is top-down it defines the upper left corner.

The barcode covers the entire visible area of the field. So, the field is rendered without a border. Changing the border color or border width has no effect.

Remarks:

A barcode field is a text field that contains a barcode dictionary. When enumerating fields with GetFieldEx() check whether *IBarcode* is set to determine whether the field is a barcode field or an ordinary text field.

Return values:

If the function succeeds the return value is the field handle, a value greater or equal zero. If the function fails the return value is a negative error code.

Example (C++):

```cpp
// Error callback function
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode,
const char* ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}
SI32 TestBarcode(void)
{
   float w, h;
   TPDFBarcode b;
   SI32 retval, f, act;
   PPDF* pdf = pdfNewPDF();
   if (!pdf) return -1; // Out of memory?
   pdfSetOnErrorProc(pdf, NULL, PDFError);

   pdfCreateNewPDF(pdf, "barcode.pdf");
   pdfSetPageCoords(pdf, pcTopDown);

   pdfAppend(pdf);

   // Two text fields to enter some test data...
   pdfCreateTextField(pdf, "Name", -1, false, -1, 50, 50, 150, 20);
   pdfCreateTextField(pdf, "Email", -1, false, -1, 50, 80, 150, 20);

   // A button to reset the form
   f   = pdfCreateButton(pdf, "Reset", "Reset", -1, 250, 50, 100, 20);
   act = pdfCreateResetAction(pdf);
   pdfAddActionToObj(pdf, otField, oeOnMouseUp, act, f);
```

```
w = 200.0f; // Field width
h = 200.0f; // Field height

// Initialize the structure with zero
memset(&b, 0, sizeof(b));
b.StructSize   = sizeof(TPDFFieldEx); // Required!
b.ECC          = 5.0f;
b.Height       = h / 72.0f;
b.nCodeWordCol = 10.0f;
b.nCodeWordRow = 90.0f;
b.Resolution   = 300;
b.Symbology    = "PDF417";
b.Version      = 1.0f;
b.Width        = w / 72.0f;
b.XSymHeight   = 8.0f;
b.XSymWidth    = 4.0f;
f = pdfCreateBarcodeField(pdf, "Barcode", -1, 50.0, 200.0, w, h, &b);

// Generic script to update the barcode
char updateBarcode[] =
   "function strTabDelimited(oParam)\n"
   "{\n"
   "    var bNeedTab = false;\n"
   "    var strNames = \"\";\n"
   "    var strValues = \"\";\n"
   "    for (var i = 0; i < oParam.oDoc.numFields; ++i)\n"
   "    {\n"
   "       var strFieldName = oParam.oDoc.getNthFieldName(i);\n"
   "       var f = oParam.oDoc.getField(strFieldName);\n"
   "       if (f != oParam.oTarget && f.type != \"button\")\n"
   "       {\n"
   "          if (bNeedTab)\n"
   "          {\n"
   "             if (oParam.bFieldNames)\n"
   "                 strNames += \"\\t\";\n"
   "             strValues += \"\\t\";\n"
   "          }\n"
   "          if (oParam.bFieldNames)\n"
   "              strNames += strFieldName;\n"
   "          strValues += f.value;\n"
   "          bNeedTab = true;\n"
   "       }\n"
   "    }\n"
   "    if (oParam.bFieldNames)\n"
   "        return strNames + \"\\n\" + strValues;\n"
   "    else\n"
   "        return strValues;\n"
   "}\n"
   "try\n"
```

```
        "{\n"
        "    event.value = strTabDelimited({oDoc: this, oTarget:"
        "event.target, bFieldNames: true});\n"
        "}\n"
        "catch(e)\n"
        "{\n"
        "    event.value = \" \";\n"
        "};\n";

    act = pdfCreateJSAction(pdf, updateBarcode);
    // Add the action to the barcode field
    pdfAddActionToObj(pdf, otField, oeOnCalc, act, f);

    pdfEndPage(pdf);

    // Make sure that a viewer updates the field appearance
    pdfSetNeedAppearance(pdf, true);

    retval = pdfCloseFile(pdf);
    pdfDeletePDF(pdf);
    return retval;
}
```

CreateButton

Syntax:

```
SI32 pdfCreateButton(
    const PPDF* IPDF,    // Instance pointer
    const char* Name,    // Name of the button
    const char* Caption, // Caption
    SI32 Parent,         // Parent group field or -1
    double PosX,         // X-Coordinate of the button
    double PosY,         // Y-Coordinate of the button
    double Width,        // Width in unscaled units
    double Height)       // Height in unscaled units
```

This function creates a push button. The parameter *Name* must be a unique name for the button field. It is not allowed to create two buttons with an identical name within the hierarchy in which they appear.

If the coordinate system is bottom-up the point *PosX, PosY* defines the lower left corner of the button. If the coordinate system is top-down it defines the upper left corner.

Like all form fields, the width and height is measured incl. the line width of the border. The size of normal vector graphics is measured without the line width; this must be taken into account when calculating the width and height of a form field.

The border is drawn by using the current line width and border style (see SetBorderStyle()). However, interactive form fields support natively the line width 1, 2 and 3 units only (thin, medium,

thick). Other values can be applied but the appearance can be changed by Adobe's Acrobat when the form is reset by a Reset Form Action.

The appearance can be influenced with the following properties:

- `Get/SetFieldBackColor()` `// Background color`
- `Get/SetFieldBorderColor()` `// Border color`
- `Get/SetFieldColor()` `// Change a specific color of a field`
- `Get/SetFieldTextColor()` `// Color of the caption`
- `Get/SetLineWidth()` `// Line width of the border`

The border and background color can be set to NO_COLOR, the background or border appears then transparent. The border appears also transparent when the line width was set to zero.

DynaPDF supports also image buttons. To create an image button store the handle of the button in a variable and add one or more images to the different button states (see AddButtonImage() for further information). The caption of the button will be overridden when an image to the up state is added.

Buttons are used to execute an action; the following events are supported by button fields:

- `OnMouseUp`
- `OnMouseDown`
- `OnMouseExit`
- `OnFocus`
- `OnBlur`

Remarks:

This function is implemented in an Ansi and Unicode compatible version. However, CJK encodings or Unicode are **NOT** supported by form fields. It is possible to use Unicode strings but the font must **NOT** use the code page cpUnicode. Unicode strings must contain characters of the actual used 8 bit code page.

Buttons requires a font, if no font was set before Helvetica is used to draw the caption by using the code page 1252. The caption is centred horizontally and vertically. By default the caption is drawn on multiple lines if it does not fit into a single line. To avoid a line break inside a button field remove the flag ffMultiline (see SetFieldFlags()).

Buttons are excluded from printing by default. If a button should be printable set the flag ffPrint (see SetFieldFlags() for further information).

Remarks:

Interactive form fields can be structured into several groups by passing a handle of a group field to the parameter *Parent*. See CreateGroupField() for further information.

Return values:

If the function succeeds the return value is the field handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateCheckBox

Syntax:

```
SI32 pdfCreateCheckBox(
    const PPDF* IPDF,      // Instance pointer
    const char* Name,      // Name of the check box (see note below)
    const char* ExpValue,  // Export value (required)
    LBOOL Checked,         // If true, the check box appears checked
    SI32 Parent,           // A parent group or radio button field or -1
    double PosX,           // X-Coordinate of the button
    double PosY,           // Y-Coordinate of the button
    double Width,          // Width of the check box
    double Height)         // Height of the check box
```

This function creates a check box. A check box can be a subfield of a radio button or of a group field (see also CreateRadioButton(), CreateGroupField()). If the parameter *Parent* is **-1** the check box is creates as stand alone field.

The parameter *Name* is ignored if the check box is a sub control of a radio button field, it can be NULL in the latter case. However, the name is required if the check box is used as stand alone field.

The parameter *ExpValue* defines the export value of the check box for the on state. This value is exported if the state of the check box is on and if the form is submitted to a web server. The export value must not be an empty string. The export value for the off state is always "*Off*".

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the button. If the coordinate system is top-down it defines the upper left corner.

Like all form fields the width and height is measured incl. the line width of the border. The size of normal vector graphics is measured without the line width; this must be taken into account when calculating the width or height of a form field. The line width of the border is derived from the current graphics state (see SetLineWidth()), it should be either 0, 1, 2, or 3 units (no border, thin, medium, or thick). The border style can be changed with the functions SetBorderStyle() or SetFieldBorderStyle().

Check boxes use always the standard font ZapfDingbats to draw the check box symbol. It is not possible to change the font but it is possible to use a custom font size. The font size is used from the active font; the kind of font that is active is ignored. There is no need to set ZapfDingbats as active font to enable the usage of a custom font size. The font size is calculated to an optimal value if the size of the active font is set to 1 unit or if no font is active.

The check box character can be changed with the property Get/SetCheckBoxChar(). If the check box character is ccCircle, the border appears circular if the check box is a sub control of a radio button field. If the check box is used stand alone, the border appears always rectangular.

Specific flags supported by check boxes only:

- `ffRadioIsUnion`: If set, a group of radio buttons within a radio button field that use the same export value for the on state will turn on and off in unison; that is if one is checked, they are all checked. This flag requires Acrobat 6 or higher.

Remarks:

Interactive form fields can be structured into several groups by passing a handle of a group field to the parameter *Parent*. See CreateGroupField() for further information.

Return values:

If the function succeeds the return value is the field handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateCIEColorSpace

Syntax:

```
SI32 pdfCreateCIEColorSpace(
  const PPDF* IPDF,  // Instance pointer
  TExtColorSpace CS, // Allowed values: esCalGray, esCalRGB, esLab
  float* WhitePoint, // White point -> required
  float* BlackPoint, // Optional black point
  float* Gamma,      // Optional gamma per component or Range in Lab
  float* Matrix)     // Optional transformation matrix -> CalRGB only
```

The function creates a CIE-based color space. Three CIE-based color spaces are supported: CalGray, CalRGB, and Lab.

The semantics of CIE-based color spaces are defined in terms of the relationship between the space's components and the tristimulus values X, Y, and Z of the CIE 1931 XYZ space. The CalRGB and Lab color spaces (PDF 1.1) are special cases of three-component CIE-based color spaces, known as CIE-based ABC color spaces. These spaces are defined in terms of a two-stage, nonlinear transformation of the CIE 1931 XYZ space. See PDF Reference for further information.

A CalGray color space (PDF 1.1) is a special case of a single-component CIEbased color space, known as CIE-based A color space. This type of space is the one-dimensional (and usually achromatic) analog of CIE-based ABC spaces. Color values in a CIE-based A space have a single component, arbitrarily named A.

The white point is an array of three numbers [XW YW ZW] specifying the tristimulus value, in the CIE 1931 XYZ space, of the diffuse white point. The numbers XW and ZW must be positive, and YW must be equal to 1.0. The parameter *WhitePoint* is always required.

The optional black point is an array of three numbers [XB YB ZB] specifying the tristimulus value, in the CIE 1931 XYZ space, of the diffuse black point. All three of these numbers must be non-negative. Default value: [0.0 0.0 0.0].

The parameter *Gamma* is optional and its meaning is different depending on the underlying color space. If the color space is CalRGB or CalGray the array must contain gamma values for each component. Default value: [1.0 1.0 1.0].

The Lab color space does not support gamma but it supports a range array which can be used to define the valid ranges of the a* and b* components of the color space. So, the parameter *Gamma* can be named as *Range* when a Lab color space will be created. If set, the array must contain exactly 4 values in the form [a_{min} a_{max} b_{min} b_{max}]. Default value: [-100 100 -100 100]. The valid range of the L* component is always 0 to 100.

The parameter *Matrix* is optional and used in CalRGB color spaces only. If set, the array must contain exactly 9 numbers representing a 3 x 3 matrix which specifies the linear interpretation of the decoded A, B, and C components of the color space with respect to the final XYZ representation.

How to define CalGray colors?

Since CalGray colors consist of only one component which ranges from 0 to 255 the colors can be defined as if you would use DeviceGray. No further conversion function is required.

How to define CalcRGB colors?

CalRGB can be used in the same way as DeviceRGB. Each component of a CalRGB color space ranges from 0 to 255. Use the macro PDF_RGB() to convert color components to a 32 bit integer. Other programming languages than C/C++ support mostly a function like rgb() which does essentially the same as the C macro PDF_RGB().

How to define Lab colors?

The Lab color space is a three component space where each component supports different ranges:

- `L*    0 to 100`
- `a* -128 to 127`
- `b* -128 to 127`

The ranges above define the maximum ranges which can be used in a Lab color space. In comparison to RGB each component is defined as signed char while in RGB each component is defined as unsigned char. C/C++ programmers can still use the macro PDF_RGB() to convert Lab colors to 32 bit integers.

If you use another programming language it is usually best to convert negative component values to unsigned char before passing the value to a function like rgb().

Example:

```
if (component_value < 0)
   result = 256 + component_value; // Note that we add a negative number!
else
   result = component_value;
```

Notice:

Non-device color spaces cannot be used for interactive objects such as annotations or form fields. The active color space must always be changed to a device color space before creating interactive objects; see SetColorSpace(). Note also that annotations support DeviceRGB only. Form fields support DeviceGray, DeviceRGB, and DeviceCMYK.

Remarks:

Use the function SetExtColorSpace() to activate the color space in the graphics state. To set a color of a CIE-based color space convert the color value to a 32 bit integer as described above and pass the result to SetFillColor(), SetStrokeColor(), or SetColors().

Return values:

If the function succeeds the return value is the color space handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateColItemDate

Syntax:

```
LBOOL pdfCreateColItemDate(
   const PPDF* IPDF,    // Instance pointer
   UI32 EmbFile,        // Handle of an embedded file
   const char* Key,     // The key defined in the collection field
   long Date,           // The date time value as integer
   const char* Prefix)  // Optional prefix to be added to the value
```

The function creates a user defined collection item which accepts a date time value. The parameter *Key* must be the key that was used in the related collection field in which the value should be shown.

The parameter *Date* is passed to the Ansi function localtime() or gmtime() to create a PDF compatible date time value. Therefore, the value must be defined in the format which these functions accept. It is usually defined as the number of seconds elapsed since midnight (00:00:00), January 1, 1970. However, depending on the operating system, the base date can be different.

The parameter *Prefix* is optional. If set, it is concatenated with the text string presented to the user. This entry is ignored when a PDF viewer application sorts the items in the collection.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CreateColItemNumber

Syntax:

```
LBOOL pdfCreateColItemNumber(
   const PPDF* IPDF,    // Instance pointer
   UI32 EmbFile,        // Handle of an embedded file
   const char* Key,     // The key defined in the collection field
   double Value,        // The value for the key
   const char* Prefix)  // Optional prefix to be added to the value
```

The function creates a user defined collection item which accepts an arbitrary number as value.

The parameter *Key* must be the key that was used in the related collection field in which the value should be shown. The parameter *Value* can be any positive or negative number. The parameter *Prefix* is optional. If set, it is concatenated with the text string presented to the user. This entry is ignored when a PDF viewer application sorts the items in the collection.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CreateColItemString

Syntax:

```
LBOOL pdfCreateColItemString(
   const PPDF* IPDF,    // Instance pointer
   UI32 EmbFile,        // Handle of an embedded file
   const char* Key,     // The key defined in the collection field
   const char* Value,   // The value for the key
   const char* Prefix)  // Optional prefix to be added to the value
```

The function creates a user defined collection item which accepts an arbitrary string as value.

The parameter *Key* must be the key that was used in the related collection field in which the value should be shown. The parameter *Value* can be an arbitrary string. However, the string should be human readable because it is displayed in the user interface.

The parameter *Prefix* is optional. If set, it is concatenated with the text string presented to the user. This entry is ignored when a PDF viewer application sorts the items in the collection.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CreateCollection

```
LBOOL pdfCreateCollection(
   const PPDF* IPDF, // Instance pointer
   TColView View)    // Initial view

typedef enum
{
   civNotSet,  // Internal, not usable
   civDetails,
   civTile,
   civHidden,
   civCustom   // Not usable. The view is presented by a SWF file.
}TColView;
```

The function marks the current PDF file in memory as PDF collection, also known as PDF Package.

Beginning with PDF 1.7, PDF documents can specify how a viewer application's user interface presents collections of file attachments, where the attachments are related in structure or content. Such a presentation is called a portable collection.

The intent of portable collections is to present, sort, and search collections of related documents, such as email achieves, photo collections, and engineering bid sets. There is no requirement that files in a collection have an implicit relationship or even a similarity; however, showing differentiating characteristics of related documents can be helpful for document navigation.

A collection consists of a normal PDF document (container PDF) that can be created in the usual way, and of a set of file attachments. The difference between a normal PDF file and a portable collection is that the viewer application provides navigation facilities for the file attachments and it is possible to open attached PDF files directly in the viewer application without loosing the navigation plane of the container PDF.

When a PDF 1.7-compliant viewer application first opens a PDF document containing a collection, it displays the contents of the initial document, along with a list of the embedded files.

The initial document is usually taken from the embedded files while the container PDF consists of only one page that displays a message that this PDF file is a collection that requires a PDF 1.7 compliant viewer which is aware of PDF collections.

Older viewer applications display just the container PDF while a PDF 1.7 compliant viewer opens the initial document taken from the embedded files in this case.

However, the initial document can be the container PDF or one of the embedded documents. The default initial document is the container PDF. If an embedded file should be initially opened then mark the embedded file as default file with SetColDefFile().

Notice:

Adobe has changed the way how PDF Collections are displayed in Acrobat 9 or higher. So, the information provided in the following section is no longer fully valid. The main difference is that

Acrobat 9 or higher displays the embedded files as thumbnails in a navigation pane. The fields FileName and Description are shown below or beside the thumbnail. The other fields are displayed in the Properties dialog of an embedded file.

Acrobat 9 or higher support also SWF files to manage the collection presentation. It is not possible to create SWF based collections with DynaPDF but it is possible to import such a collection and to add further embedded files to it.

> The embedded files of a collection are shown in a list view in Acrobat or Reader 8. Without adding further information to the collection, the list view contains a set of default columns such as file name, description, modification date and so on, displayed in the current language of the viewer. The information displayed in the rows is taken from the embedded files.

> The information displayed in the list view can be customized to your own requirements. To understand how this works keep in mind that a list view consists of columns and rows. The columns which should be displayed are defined in the collection (collection fields) while the information displayed in the rows is taken from the embedded files.

> To customize the list view of a collection you must first create one or more collection fields with CreateCollectionField(). The function defines the column name as well as the key and data type to be used for that column. The information for the collection field is stored in the embedded files.

> Therefore, each field that was defined in the collection should have a related collection item in **all** embedded files file specification dictionaries. However, only user defined fields require a collection item. Depending on the data type that was defined for a specific collection field, the collection items can be added to the embedded files with the following functions:

> - CreateColItemDate()
> - CreateColItemNumber()
> - CreateColItemString()

> The functions above require always the embedded file's handle, the key, and the data for that key. The key was defined in the prior CreateCollectionField() call. Note that the key of a collection field is used to find the data in the embedded files file specification dictionaries. The column name is used to display it in the user interface.

> Once the collection fields and items were defined it is possible to mark one field as sort field. The list is then sorted by this field either in ascending or descending order (see SetColSortField() for further information).

Remarks:

DynaPDF contains example projects which demonstrate how collection fields and items can be created. Use the function CheckCollection() to determine whether the connection between collection fields and collection items is valid.

The container PDF should always consist of only one page that displays the compatibility message as described above and shown in the demo projects. Collections which are created in this way are very suitable for merging multiple collections into one file. If the container PDF is a normal PDF file, many advantages of collections are lost. Especially merging of such files is then as difficult as with normal PDF files.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CreateCollectionField

Syntax:

```
SI32 pdfCreateCollectionField(
    const PPDF* IPDF,          // Instance pointer
    TColColumnType ColType,    // Data type for the column
    SI32 Column,               // Column index or -1, see below
    const char* Name,          // Column name to be displayed in the viewer
    const char* Key,           // Key to find the data for that column
    LBOOL Visible,             // If false, the column is not displayed
    LBOOL Editable)            // If true, the column is editable


typedef enum
{
    cisCreationDate, // Predefined column, no collection item required
    cisDescription,  // Predefined column, no collection item required
    cisFileName,     // Predefined column, no collection item required
    cisModDate,      // Predefined column, no collection item required
    cisSize,         // Predefined column, no collection item required
    cisCustomDate,   // User defined date
    cisCustomNumber, // User defined number
    cisCustomString  // User defined string
}TColColumnType;
```

The function creates a user defined collection field. A collection field represents a column in the list view of the user interface of a PDF 1.7 compliant PDF viewer. The parameter *ColType* defines the type of the column that should be added to the list. Predefined columns are already available in the embedded files file specification dictionaries; there is no need to create collection items for these fields.

User defined fields are available for three basic data types depending on the value that should be stored in the collection item. User defined column types require always relating collection items in all embedded files file specification dictionaries. A collection item holds the data that should be displayed in a given column. See also CreateColItemDate(), CreateColItemNumber(), CreateColItemString().

The parameter *Name* represents the column name that should be displayed in the list view of the viewer application. The parameter *Key* is used to find the value for the column in the embedded files file specification dictionaries. It should be defined as a 7 bit ASCII string without any special character. The key is also required to create the relating collection items for the embedded files.

Make sure that you use exactly the same key to create the collection items and that the right data type is used.

Use the function SetColSortField() to sort the list of embedded files by a specific collection field.

It is usually best to check the validity of the collection with CheckCollection() after it was fully created. This is especially important if multiple collections were merged into one file.

The order in which collection fields and collections items are created can be arbitrary. The collection items can be created before the collection fields and vice versa.

Remarks:

This function is implemented in an Ansi and Unicode compatible variant. The relationship between collection fields and collection items is explained in more detail under CreateCollection(). DynaPDF contains also example projects which demonstrate how collection fields and items can be created. Use the function CheckCollection() to determine whether the connection between collection fields and collection items is valid.

Return values:

If the function succeeds the return value is the handle of the collection field, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateComboBox

Syntax:

```
SI32 pdfCreateComboBox(
   const PPDF* IPDF, // Instance pointer
   const char* Name, // Name of combo box
   LBOOL Sort,       // Sort the choice values
   SI32 Parent,      // Parent group field or -1
   double PosX,      // X-Coordinate of the combo box
   double PosY,      // Y-Coordinate of the combo box
   double Width,     // Width of the combo box
   double Height)    // Height of the combo box
```

This function creates a combo box. The choice values of the combo box must be added with the function AddValToChoiceField(). This function requires the field handle that was returned by this function.

A combo box can contain value that is not included in the array of choice values. This value can be used to display a string in the combo box when no choice value is selected, e.g. "Select a value!".

The field value can be set with the function SetFieldExpValue() as follows:

The parameter *ValIndex* of the function SetFieldExpValue() must be set to PDF_MAX_INT and the parameter *Value* must contain the wished value or NULL. If *Value* is NULL or an empty string, a maybe existing field value will be deleted. The choice values of the combo box get the state unselected when setting a field value in this way.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the button. If the coordinate system is top-down it defines the upper left corner.

Like all form fields the width and height is measured incl. the line width of the border. The size of normal vector graphics is measured without the line width; this must be taken into account when calculating the width or height of a form field. The line width of the border is derived from the current graphics state (see SetLineWidth()), it should be either 0, 1, 2, or 3 units (no border, thin, medium, or thick). The border style can be changed with the functions SetBorderStyle() or SetFieldBorderStyle().

The values of the combo box are sorted if the parameter *Sort* is true.

A combo box requires a font. If no font is active the standard font Helvetica is used. Note: If the form must be compatible to Acrobat 4 the used fonts by the form fields must be restricted to the 14 standard fonts (see SetFont() for further information). Acrobat 4 does not support interactive forms that use other fonts than the 14 standard fonts. Such a form requires Acrobat 5 or higher.

Form fields support natively the code pages 1252 and MacRoman. However, other code pages and Unicode can be used too as long as a few external cmaps are available. See SetFieldFont() for further information.

The font size of a combo box is set to "auto" if either the font size of the active font is set to 1.0 unit or if no font is active when the field is created.

Specific flags supported by combo boxes:

- `ffEdit`   `// Enable adding of values in Adobe's Acrobat`
- `ffSorted`   `// Change the sort flag if necessary`
- `ffCommitOnSelCh`   `// Submit the new value immediately (PDF 1.5)`
- `ffDoNotSpellCheck` `// Disable spell checking (requires ffEdit)`

Remarks:

Interactive form fields can be structured into several groups by passing a handle of a group field to the parameter *Parent*. See CreateGroupField() for further information.

Return values:

If the function succeeds the return value is the handle of the combo box, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateDeviceNColorSpace

Syntax:

```
SI32 pdfCreateDeviceNColorSpace(
  const PPDF* IPDF,            // Instance pointer
  const char** Colorants,      // Array of colorant names (required)
  UI32 NumColorants,           // Number of colorants in the array
  const char* PostScriptFunc,  // Postscript calculator function
(required)
  TExtColorSpace Alternate,    // Alternate color space
  SI32 Handle)                 // Handle of the alternate color space or -
1
```

The function creates a DeviceN color space. DeviceN color spaces (PDF 1.3) can contain up to 32 color components. They provide greater flexibility than is possible with standard device color spaces such as DeviceCMYK or with individual Separation color spaces. For example, it is possible to create a DeviceN color space consisting of only the cyan, magenta, and yellow color components, with the black component excluded.

Colors of a DeviceN color space are always treated as subtractive colors, even if the device produces output for the designated component by an additive method. Thus, a tint value of 0 denotes the lightest color that can be achieved with the given colorant and 255 the darkest. This convention is the same one as for DeviceCMYK color components but opposite to the one for DeviceGray or DeviceRGB.

DeviceN was designed to represent color spaces containing multiple components that correspond to colorants of some target device. As with Separation color spaces, PDF consumer applications must be able to approximate the colorants if they are not available on the current output device, such as a display. To accomplish this, the color space definition provides a tint transformation function that can be used to convert all the components to an alternate color space.

The tint transformation function consists of a PostScript calculator function that is called with n tint values and returns m color component values, where n is the number of components needed to specify a color in the DeviceN color space and m is the number of components required by the alternate color space.

Note: Painting in the alternate color space may produce a good approximation of the intended color when only opaque objects are painted. However, it does not correctly represent the interactions between an object and its backdrop when the object is painted with transparency or when overprinting is enabled (see also CreateExtGState()).

A DeviceN color space supports the special colorant name *None* which produces no visible output on the device. When a DeviceN color space is painting the named device colorants directly, color components corresponding to *None* colorants are discarded. However, when the DeviceN color space reverts to its alternate color space, those components are passed to the tint transformation function, which can use them as desired.

Note: A DeviceN color space whose component colorant names are all *None* always discards its output, just the same as a Separation color space for *None*; it never reverts to the alternate color space. Reversion occurs only if at least one color component (other than *None*) is specified and is not available on the device.

Encoding of Colorant Names

Colorant names are interpreted in the code page 1252 by default. Because colorant names are stored in UTF-8 Unicode format in PDF, it is also possible to pass UTF-8 encoded Unicode strings to the function. However, the function must be able to distinguish between both string formats. To achieve this, the parameter *NumColorants* accepts the special flag 0x10000000 that specifies that the *Colorants* array contains UTF-8 encoded strings. The flag must be combined with the number of colorants with a binary or operator:

```
numColorants |= 0x10000000;                      // C/C++, C#
numColorants  = numColorants Or &H10000000 // Visual Basic
numColorants := numColorants or $10000000  // Delphi
```

DeviceN Attributes

DeviceN and NChannel color spaces can contain optional attributes describing further characteristics of the color space. The most important attributes are the definitions of the spot and process colorants which are used by the color space. These attributes are required to enable certain features in a viewer application, e.g. the Output Preview in Adobe's Acrobat depends on it.

Notice: Although the definition of spot and process colorants is optional, it is bad practice to create a DeviceN color space without this information.

Spot colorant attributes

Spot colorants can be defined as normal Separation color spaces and then added as an attribute to the color space with AddDeviceNSeparations(). The advantage is that the alternate color space and tint transformation function of the Separation color space describe the appearance of that colorant alone, whereas those of a DeviceN color space describe only the appearance of its colorants in combination. The definition of the spot colorants is optional but strongly recommended because it is the only way to render a spot colorant alone. Note that the Output Preview in Adobe's Acrobat depends on properly defined spot colorants.

Process colorant attributes

If a DeviceN color space contains components of a process color space then it should include information about the color space in which these components are defined. This can be done with the function AddDeviceNProcessColorants() which accepts an array of colorant names and the underlying process color space. The process color space can be any device or CIE based color space. If an ICCBased color space is specified, it must provide calibration information appropriate for the process color components specified in the names array of the DeviceN color space.

The array of colorant names must correspond, in order, to the components of the process color space. For example, an RGB color space must have three names corresponding to red, green, and blue. The names may be arbitrary (that is, not the same as the standard names for the color space components) but must match those specified in the DeviceN color space, even if not all components are present in the DeviceN color space.

The defintion of process colorants of a DeviceCMYK color space is optional since the colorant names Cyan, Magenta, Yellow, and Black are always considered as color components of a DeviceCMYK color space.

How to create the PostScript Calculator Function?

A PostScript calculator function is represented as a string containing code written in a small subset of the PostScript language. The language that can be used in a PostScript calculator function contains expressions involving integers, real numbers, and boolean values only. There are no composite data structures such as strings or arrays, no procedures, and no variables or names.

The following operators are supported in a PostScript calculator function:

Operator type	Operators				
Arithmetic operators	abs	cvi	floor	mod	sin
	add	cvr	idiv	mul	sqrt
	atan	div	ln	neg	sub
	ceiling	exp	log	round	truncate
	cos				
Relational, boolean, and bitwise operators	and	false	le	not	true
	bitshift	ge	lt	or	xor
	eq	gt	ne		
Conditional operators	if	ifelse			
Stack operators	copy	exch	pop		
	dup	index	roll		

For more information on these operators, see Appendix B of the PostScript Language Reference, Third Edition. The operand syntax for PostScript calculator functions follows PDF conventions rather than PostScript conventions. The entire code defining the function is enclosed in braces { }. Braces also delimit expressions that are executed conditionally by the **if** and **ifelse** operators:

boolean { expression } if

boolean { expression1 } { expression2 } ifelse

This construct is purely syntactic; unlike in PostScript, no "procedure objects" are involved.

When executing the function, the application pushes first the tint values of the setcolor operator or the corresponding image components on the stack. The function is then executed and the remaining

components on the stack are adjusted to the valid input range of the alternate color space and then passed to it.

Note also that the remaining components on the stack must match the number of input colorants of the alternate color space. It is an error to leave more or less components on the stack than supported by the alternate color space.

Example 1:

In this example we create a DeviceN color space that contains the process colorants Cyan, Magenta, and Yellow. The alternate color space is of course DeviceCMYK. The definition of the PostScript calculator function is very simple in this example because we need to add the missing Black component only. The array *cls* includes also the Black component here because we need the colorant names later to add the definition of the process components to the DeviceN color space:

```
...
const char* cls[] = {"Cyan", "Magenta", "Yellow", "Black"};
const char   ps[] = "{0}"; // Just add the missing Black component

pdfAppend(pdf);

// Note that the Black component is not part of the DeviceN color space
SI32 cs = pdfCreateDeviceNColorSpace(pdf, cls, 3, ps, esDeviceCMYK, -1);
// Optional but strongly recommended: Add the defintion of the process
// colorants to the DeviceN color space.
pdfAddDeviceNProcessColorants(pdf, cls, 4, esDeviceCMYK, -1);

// Draw a rectangle in the alternate DeviceCMYK color space
pdfSetFillColorSpace(pdf, csDeviceCMYK);
pdfSetFillColor(pdf, PDF_CMYK(135, 65, 160, 0));
pdfRectangle(pdf, 50.0, 50.0, 200.0, 100.0, fmFill);
// Now we use the DeviceN color space. The colors of both rectangles must
// be identically. If you see a difference, then disable the output
// preview in Adobe's Acrobat, since the color is converted into the
// simulation profile otherwise...
pdfSetExtColorSpace(pdf, cs);
BYTE color[] = {135, 65, 160};
pdfSetFillColorEx(pdf, color, 3);
pdfRectangle(pdf, 50.0, 150.0, 200.0, 100.0, fmFill);

pdfEndPage(pdf);
...
```

Example 2:

In this example we want to define a DeviceN color space that contains 2 spot colors and the process color yellow. The alternate color space for the spot colors is DeviceCMYK in this example and the process colorant is defined in this color space too.

The tint transformation function for such a color space must be able to compute one color from arbitrary combinations of these three colors. So, we need essentially a blend function expressed in PostScript syntax.

The definition of a blend function is not difficult but long because the entire code is stack based. However, let us first keep in mind what we need to do. We have already three values on the stack when the function is executed. These are the tint values which were passed to the DeviceN color space. We need to multiply the tint values with the four components of the alternate color representations of all three channels and finally we need to combine these three colors so that we get the resulting output color. This sounds difficult, because a lot of multiplications and additions are required to produce the wished result, but you'll see it's easier as it looks like.

The used colors in the DeviceN color space are defined as follows:

Colorant name	CMYK representation
PANTONE 345 CVC (coated)	{0.38, 0.00, 0.34, 0.0} or { 97, 0, 87, 0}
PANTONE 293 CVC (coated)	{1.00, 0.56, 0.00, 0.0} or {255, 143, 0, 0}
Yellow	{0.00, 0.00, 1.00, 0.0} or { 0, 0, 255, 0}

The CMYK color values of the spot colors are taken from the color picker of Adobe's Photoshop.

Now, we have all information we need to create the PostScript calculator function.

The multiplication of the color channels with the tint values must be done for all three colors included in the color space. Because these are repeating operations we can easily develop a function that creates the necessary code for us:

```
// The following function creates the PostScript code to multiply the tint
// values with the components of the CMYK color representation. 0 and 1
// color values are optimized to more efficient code.
UI32 AddOperator(char* Dest, float Value, UI32 Index)
{
   if (Value == 0.0f)      // Zero is always zero
      return sprintf(Dest, "0 ");
   else if (Value == 1.0f) // 1.0 represents the tint value
      return sprintf(Dest, "%d index ", Index);
   else // Other color values must be multiplied with the tint value
      return sprintf(Dest, "%d index %.2f mul ", Index, Value);
}
// Note that this function does not allocate memory. The string must be long enough to hold the
// resulting PostScript code. The function can be used to create the blend function for DeviceN color
// spaces with up to 32 color channels. The alternate color space must be DeviceCMYK or an ICCBased
// color space whose base color space is in turn DeviceCMYK. Since the alternate color space must be
// DeviceCMYK, NumColorants is always a multiple of 4!
void CreateBlendFunction(
    char* Dest,        // Destination string buffer
    float* Colorants,  // Array of color values
    UI32 NumColorants) // Must be a multiple of 4!
{
    UI32 i, j, len = 1;
    UI32 numColors = NumColorants / 4;
    UI32    index = numColors - 1;
    Dest[0] = '{'; // The PostScript code must be enclosed in braces
    for (i = 0; i < 4; i++, index++)
    {
        for (j = 0; j < NumColorants; j += 4)
        {
            // Create code to multiply the tint values with the color values
```

```
            len += AddOperator(Dest + len, Colorants[i+j], index);
        }
        // Add the resulting components. No range check is required...
        for (j = 0; j < numColors - 1; j++)
            len += sprintf(Dest + len, "add ");
    }
    // We are almost finished. Place the tint values on top of the stack
    len += sprintf(Dest + len, "%d 4 roll", index +1);
    // Remove the tint values from the stack
    for (j = 0; j < numColors; j++)
        len += sprintf(Dest + len, " pop");
    strcpy(Dest + len, "}"); // Finish the code with a brace
}
```

Now we have a simple function that creates the required PostScript code for us. The usage is as follows:

```
// This is our error callback function.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode,
const char* ErrMessage, SI32 ErrType)
{
    printf("%s\n", ErrMessage);
    return 0;
}

SI32 TestDeviceNColorSpace(const PPDF* PDF)
{
    SI32 cs;
    char psFunc[512]; // Buffer that holds the PostScript code
    const char* cls[] = {"PANTONE 345 CVC", "PANTONE 293 CVC", "Yellow"};
    const char* pcs[] = {"Cyan", "Magenta", "Yellow", "Black"};
    float colors[] =
    {   // The CMYK color values are taken from Adobe's Photoshop
        0.38f, 0.00f, 0.34f, 0.0f, // Definition of the first spot color
        1.00f, 0.56f, 0.00f, 0.0f, // Definition of the second spot color
        0.00f, 0.00f, 1.00f, 0.0f  // Definition of the process color
    };

    pdfSetOnErrorProc(PDF, NULL, PDFError);
    if (!pdfCreateNewPDF(PDF, "test.pdf")) return -2;

    pdfAppend(PDF);

    // We need to create the PostScript calculator function first because
    // it is a required parameter of pdfCreateDeviceNColorSpace().
    CreateBlendFunction(psFunc, colors, sizeof(colors) / sizeof(float));
    // Create the DeviceN color space
    cs = pdfCreateDeviceNColorSpace(
        PDF,           // Instance pointer
        cls,           // The colorants array
        3,             // Number of colorants in the array
        psFunc,        // Our PostScript tint transformation function
        esDeviceCMYK,  // Alternate color space
        -1);           // No handle is required for a device color space

    if (cs < 0) return cs;

    // We create also Separation color spaces for the spot colors and add
    // these color spaces as an attribute to the DeviceN color space:
    UI32 separations[2];
    // First spot color
    separations [0] = pdfCreateSeparationCS(
        PDF,
        cls[0],
        esDeviceCMYK,
        -1,
        PDF_CMYK(97, 0, 87, 0)); // 0.38 * 255, 0 * 255, 0.34 * 255, 0 * 255
```

```
    // Second spot color
    separations[1] = pdfCreateSeparationCS(
        PDF,
        cls[1],
        esDeviceCMYK,
        -1,
        PDF_CMYK(255, 143, 0, 0)); // 1.0 * 255, 0.56 * 255, 0 * 0, 0 * 0

    // Add the separation color spaces to the DeviceN color space.
    pdfAddDeviceNSeparations(
        PDF,          // Instance pointer
        cs,           // DeviceN color space handle
        cls,          // Colorants array
        separations,  // Separation color space handles for these colorants
        2);           // Number of Separation color spaces

    // Because the DeviceN color space uses a process color we add also
    // the attributes of the process color space to it. Note that all
    // colorant names must be defined, also if the DeviceN color space
    // uses only one component of the process color space.
    pdfAddDeviceNProcessColorants(
        PDF,          // Instance pointer
        cs,           // DeviceN color space handle
        pcs,          // Array of process colorants
        4,            // Number of process colorants
        esDeviceCMYK, // The used process color space
        -1);          // No handle is required for a device color space

    // The DeviceN color space is now fully created so that we can use it
    BYTE color[] = {150, 100, 100};
    pdfSetExtColorSpace(PDF, cs);     // Set the DeviceN color space
    pdfSetFillColorEx(PDF, color, 3); // Set a fill color

    // Draw a rectangle with this color
    pdfRectangle(PDF, 50.0, 690.0, 200.0, 100.0, fmFill);

    pdfEndPage(PDF);

    return pdfCloseFile(PDF);
}

int main(int argc, char* argv[]
{
    SI32 retval;
    PPDF* pdf = pdfNewPDF();
    if (!pdf) return -1; // Out of memory?

    retval = TestDeviceNColorSpace(pdf);

    pdfFreePDF(pdf);
    return retval;
}
```

As you can see the creation of a DeviceN color space with multiple spot and process colorants is not difficult. With the helper function that creates the required PostScript calculator function, you can easily create DeviceN color spaces with up to 32 color channels.

Return values:

If the function succeeds the return value is the color space handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateDPartRoot

Syntax:

```
LBOOL pdfCreateDPartRoot(
   const PPDF* IPDF,         // Instance pointer
   const char** NodeNameList, // Required
   UI32 NodeNamesCount,       // Number of array values
   SI32 RecordLevel)          // Record level or 0
```

The function creates the *DPartRoot* dictionary that must be present before *DPart* nodes can be created with AddDPartNode().

Note that the function will fail if a *DPartRoot* dictionary was already imported or created by this function. You can check whether a *DPartRoot* dictionary exists with HaveDPartRoot().

DPart stands for Document Part. *DParts* are a feature of PDF/VT, a PDF format for the printing industry.

PDF pages are internally stored in a page tree structure. Pages in this tree are organized for fast access to abitrary pages. Beginning with PDF/VT a second tree structure was added, the *DPart* tree. Unlike the page tree, the *DPart* tree can be used to group pages and to store additational metadata for these nodes or pages.

Overlapping page ranges are prohibited because page objects, which are referenced by a *DPart* node, get a reference of this node. One page can reference exactly one *DPart* node. This allows direct access to the additional metadata of a *DPart* node.

DPart metadata (DPM) is stored in PDF syntax, while regular metadata streams in PDF are XML based. DPM requires no additional XML or XMP parser and is much faster to parse.

The required parameter *NodeNameList* represents a label or name for each *DPart* node level when displaying the *DPart* tree on screen with a tree control or when exporting it to XML, for example.

A child node increments the level by 1. When the children of a node were processed or displayed, the level is decremented by 1. Every *DPart* node has a parent node. The root node is the one that is created by this function.

Example:

"N" stands for Node, "P" for Parent node. The number at the end represents the tree node level.

```
N1,PRoot 0
   N2,P1 1
      N3,P2 2
      N4,P2 2
      N5,P2 2
   N6,P1 1
      N7,P6 2
         N8,P7 3
            N9,P8 4
            N10,P8 4
```

```
N11,P1 1
```

The tree would be displayed or exported to XML as follows if *NodeNameList* would contain these 5 entries: "Root", "Pages", "RecordsList1", "RecordList2", "Other":

```
Root
    Pages
        RecordList1
        RecordList1
        RecordList1
    Pages
        RecordList1
            RecordList2
                Other
                Other
    Pages
```

As you can see the naming sheme is rather simple. What happens if no name was defined for a specific level is implementation specific.

The parameter *RecordLevel* specifies where each *DPart* node of that level corresponds to a recipient record. If this is unknown or if there are might be multiple levels then set the parameter to 0.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CreateExtGState

Syntax:

```
SI32 pdfCreateExtGState(
    const PPDF* IPDF,          // Instance pointer
    struct TPDFExtGState* GS) // Extended graphics state structure

struct TPDFExtGState
{
    UI32             AutoStrokeAdjust; // PDF_MAX_INT if not set
    TBlendMode       BlendMode;        // Default bmNotSet
    float            FlatnessTol;      // -1.0 if not set
    UI32             OverPrintFill;    // PDF_MAX_INT if not set
    UI32             OverPrintStroke;  // PDF_MAX_INT if not set
    UI32             OverPrintMode;    // PDF_MAX_INT if not set
    TRenderingIntent RenderingIntent;  // riNone if not set
    float            SmoothnessTol;    // -1.0 if not set
    float            FillAlpha;        // -1.0 if not set
    float            StrokeAlpha;      // -1.0 if not set
    UI32             AlphaIsShape;     // PDF_MAX_INT if not set
    UI32             TextKnockout;     // PDF_MAX_INT if not set
    LBOOL            SoftMaskNone;     // Disables a soft mask
    void*            Reserved1;
    void*            Reserved2;
    void*            Reserved3;
    void*            Reserved4;
    void*            Reserved5;
    void*            Reserved6;
    void*            Reserved7;
    void*            Reserved8;
};
```

```
typedef enum
{
    bmNotSet     = 0,
    bmNormal     = 1,
    bmColor      = 2,
    bmColorBurn  = 3,
    bmColorDodge = 4,
    bmDarken     = 5,
    bmDifference = 6,
    bmExclusion  = 7,
    bmHardLight  = 8,
    bmHue        = 9,
    bmLighten    = 10,
    bmLuminosity = 11,
    bmMultiply   = 12,
    bmOverlay    = 13,
    bmSaturation = 14,
    bmScreen     = 15,
    bmSoftLight  = 16
}TBlendMode;

typedef enum
{
    riAbsoluteColorimetric = 0,
    riPerceptual           = 1,
    riRelativeColorimetric = 2,
    riSaturation           = 3,
    riNone                 = 4 // Internal
}TRenderingIntent;
```

The function creates an extended graphics state object from the structure *GS*. An extended graphics state can be used to adjust certain properties which are not part of the normal graphics state. Such properties can only be modified with a special PDF object called the Extended Graphics State.

The most important properties of an extended graphics state are the transparency settings (PDF 1.4) and the overprint control (PDF 1.2 or 1.3, see below).

The structure *GS* must always be initialized with InitGState(). After the structure was initialized certain members of the structure can be changed. Pass the variable then to CreateExtGState(). The function creates an internal graphics state object and returns the handle of it. The extended graphics state can now be applied with SetExtGState().

Note that there is only way to restore the values of an extended graphics state, you must set a second extended graphics state that restores the values changed before.

The following values are allowed:

- `AlphaIsShape`   -> 0 or 1 (PDF 1.4, default unknown, does nothing)
- `AutoStrokeAdjust` -> 0 or 1 (PDF 1.4, default 0)
- `BlendMode`   -> see enum TBlendMode (PDF 1.4, default bmNormal)
- `FlatnessTol`   -> 0.0..1.0, default none
- `FillAlpha`   -> 0.0..1.0 (PDF 1.4, default 1.0)
- `OverPrintFill`   -> 0 or 1 (PDF 1.2 if used alone, PDF 1.3 otherwise,
                                         default 0)
- `OverPrintStroke` -> 0 or 1 (PDF 1.3, default 0)
- `OverPrintMode`   -> 0 or 1 (PDF 1.3, default 0)

- RenderingIntent   -> see enum TRenderingIntent (PDF 1.1, default
                          riRelativeColorimetric)
- SmoothnessTol;   -> 0.0..1.0 (PDF 1.3, default none)
- StrokeAlpha   -> 0.0..1.0 (PDF 1.4, default 1.0)
- TextKnockout   -> 0 or 1   (PDF 1.4, default 1)
- SoftMaskNone   -> true or false, default false

A value that should not be modified must not be changed.

The structure contains many reserved fields which are not used at this time. Future extensions will replace reserved fields.

If an image should be inserted with transparency it is usually best to disable image transparency which based on color key masking (see SetUseTransparency()). Images use the fill alpha constant as alpha channel. Blend modes can be applied too. However, the result of blend modes depends strongly on the used color space. If the image is not defined in an ICC based color space then the result depends on the color management settings.

Whenever transparency related settings are changed (FillAlpha, StrokeAlpha, or BlendMode) it is strongly recommended to attach an ICC profile of the output color space for which the file was created with AddOutputIntent().

If no output intent was specified, Adobe's Acrobat or Reader simulate a default CMYK color space as output color space if transparency is used on a page, also if the page uses RGB colors only! This differs from normal RGB files which use no transparency and causes often unwanted color changes.

Note that this rule applies also to images which contain an alpha channel!

If the PDF file uses RGB colors only, then attach an sRGB profile to the file. Such a profile is very small but it makes a huge difference whether the profile is present or not.

Remarks:

Transparency and blend modes are discussed in detail in the section PDF Transparency.

Return values:

If the function succeeds the return value is the extended graphics state handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateGeospatialMeasure

Syntax:

```
SI32 pdfCreateGeospatialMeasure(
   const PPDF* IPDF,         // Instance pointer
   UI32 Viewport,            // Viewport handle (required)
   const char* Attributes)   // JSON like string. See description
```

The function creates a geospatial measure dictionary. This kind of measure dictionary can be used to convert PDF units to geographic coordinates.

A measure dictionary is used by the measure tool of Adobe's Acrobat and by other PDF viewers which contain a PDF compatible measure tool.

Notice:

> This is a low level function for more experienced users who are familiar with PDF. It is assumed that a copy of the ISO Standard 32000-2 or newer is available by these users.

The parameter *Viewport* must contain a valid viewport handle that was created by CreateViewport().

The required parameter *Attributes* must be a JSON like string (Javascript Object Notation) that contains the necessary parameters for the coordinate conversion. Please have a look at the section JSON Parser to determine how JSON strings can be created and which extensions are supported.

The function creates a dictionary that contains already the keys **/Type** and **/Subtype**. These keys are reserved and must not be present.

Notice:

> It seems that Adobes Acrobat does not consider the key /PDU (Preferred Display Units), e.g. /PDU[/KM/SQKM/DEG]. In all tests /PDU was ignored by Acrobat. It is not known when this key will be considered or whether it is supported at all.

Please note that double quotes in a WKT string must be preceded by a backslash since double quotes are a string terminator in JSON grammar.

Example JSON string:

In order to test this string copy it to a text file and load the file afterwards into a string object and pass it to the function. Note that the string must be null-terminated. The same string defined as C/C++ literal string would require more complicated escaping due to the double quotes in the WKT string.

The example JSON string was created from this source file:

https://sg.geodatenzentrum.de/public/gdz/themenkarten/wss-2022-07-strassentunnel.pdf

```
/Bounds[0 0 0 1 1 1 1 0 0 0]
/GCS
{
   /Type/PROJCS
   /WKT
   "PROJCS[\"WGS_1984_Lambert_Conformal_Conic\",GEOGCS[\"GCS_WGS_1984\",DA
   TUM[\"D_WGS_1984\",SPHEROID[\"WGS_1984\",6378137.0,298.257223563]],PRIM
   EM[\"Greenwich\",0.0],UNIT[\"Degree\",0.0174532925199433]],PROJECTION[\
   "Lambert_Conformal_Conic\"],PARAMETER[\"False_Easting\",0.0],PARAMETER[
   \"False_Northing\",0.0],PARAMETER[\"Central_Meridian\",10.5],PARAMETER[
   \"Standard_Parallel_1\",48.66666666666666],PARAMETER[\"Standard_Paralle
   l_2\",53.66666666666666],PARAMETER[\"Latitude_Of_Origin\",51.0],UNIT[\"
   Meter\",1.0]]"
}
/GPTS[47.09291 6.12161 54.98462 5.30494 54.99117 15.53156 47.09843
14.74047]
/LPTS[0 0 0 1 1 1 1 0]
```

A valid geospatial measure dictionary requires at least the keys /GCS and /GPTS. All other keys are optional.

Example C++:

```cpp
SI32 TestGeo(const PPDF* PDF, wchar_t* OutFileName)
{
   pdfCreateNewPDFW(PDF, OutFileName);
   pdfSetDocInfoA(PDF, diProducer, NULL);

   SI32 bufSize;
   // This example assumes that the above example string was copied to the file geo.txt.
   char* attr = GetFileBuf("e:/geo.txt", bufSize);
   // This code creates just an emptry page and adds a viewport to it. Open the file in
   // Acrobat and check the coordinates with the Geospatial Location Tool, for example.
   // It must show valid latitute and longitute coordinates. Distances can be measured
   // too.
   pdfAppend(PDF);
      pdfSetBBox(PDF, pbMediaBox, 0.0, 0.0, 841.88806, 1190.54883);
      SI32 vp = pdfCreateViewport(PDF, "Test", 51.2673, 88.08751, 793.90894, 1085.48682);
      pdfCreateGeospatialMeasure(PDF, vp, attr);
      free(attr);
   pdfEndPage(PDF);
   return pdfCloseFile(PDF);
}
```

Return values:

If the function succeeds the return value is the handle of the measure dictionary, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateGoToAction

Syntax:

```
SI32 pdfCreateGoToAction(
    const PPDF* IPDF,    // Instance pointer
    TDestType DestType,  // Destination type
    UI32 PageNum,        // Destination page
    double a,            // Various, depends on destination type
    double b,            // Various, depends on destination type
    double c,            // Various, depends on destination type
    double d)            // Various, depends on destination type

typedef enum
{
    dtXY_Zoom,   // Three parameters (a, b, c) -> (X, Y, Zoom)
    dtFit,       // No parameters
    dtFitH_Top,  // One parameter    (a)
    dtFitV_Left, // One parameter    (a)
    dtFit_Rect,  // Four parameters  (left, bottom, right, top)
    dtFitB,      // No parameters
    dtFitBH_Top, // One parameter    (a)
    dtFitBV_Left // One parameter    (a)
}TDestType;
```

A go-to action changes the view to a specific destination (page, location, and magnification factor). This action type is useful if a specific destination must be used by several objects such as bookmarks and page links.

Destination type	Description
dtXY_Zoom	Display the page designated by page with the coordinates (left, top) positioned at the top-left corner of the window and the contents of the page magnified by the factor zoom. A zero value for any of the parameters left top or zoom specifies that the current value of that parameter is to be retained unchanged. **Example**: `pdfCreateGotoAction(pdf, dtXY_Zoom, 1, 0, 750, 0, 0);`
dtFit	Display the page designated by page with its contents magnified just enough to fit the entire page within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the page within the window in the other dimension. This destination type has no parameters, the values of a, b, c, d are ignored.

Destination type	Description

dtFitH_Top	Display the page designated by page with the vertical coordinate top positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of the page within the window.
	
	Example:
	
	``
	// The parameter a specifies the top coordinate
	pdfCreateGotoAction(pdf, dtFitH_Top, 1, 565, 0, 0, 0);
	``

dtFitV_Left	Display the page designated by page with the horizontal coordinate left positioned at the left edge of the window and the contents of the page magnified just enough to fit the entire height of the page within the window.
	
	Example:
	
	``
	// The parameter a specifies the left edge
	pdfCreateGotoAction(pdf, dtFitV_Left, 1, 60, 0, 0, 0);
	``

dtFitRect	Display the page designated by page with its contents magnified just enough to fit the rectangle specified by the coordinates left bottom right and top entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the rectangle within the window in the other dimension. Note, the maximum zoom factor supported by Adobe's Acrobat is limited to 16 (Acrobat 4/5) or 64 if Acrobat 6 is used. It is not possible to zoom into the rectangle if it is too small.
	
	Example:
	
	``
	pdfCreateGotoAction(pdf,dtFit_Rect,1,150,550,450,700);
	``

The destination types *dtFitB, dtFitBH_Top, and dtFitBV_Left* use always the media box of the page to fit the page into the window. All other destination types use the crop box if any.

Remarks:

Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateGoToActionEx

Syntax:

```
SI32 pdfCreateGoToActionEx(
   const PPDF* IPDF, // Instance pointer
   UI32 NamedDest)   // Handle of a named destination
```

The function creates a go-to action which uses a named destination to open the target page. A named destination can be used if the destination should be accessible from another PDF file. See also CreateNamedDest().

Remarks:

Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateGoToEAction

Syntax:

```
SI32 pdfCreateGoToEAction(
   const PPDF* IPDF,         // Instance pointer
   TEmbFileLocation Location, // see below
   const char* Source,       // Required for external files
   UI32 SrcPage,             // If the file is located in an annotation
   const char* Target,       // The target file that should be opened
   const char* DestName,     // A named destination to be opened
   UI32 DestPage,            // Only used if DestName is NULL
   LBOOL NewWindow)          // Open the file in a new window?

typedef enum
{
   eflChild,        // The file is embedded in the current file
   eflChildAnnot,   // The file is embedded in a file attachment annotation
   eflExternal,     // The file is an embedded file in an external file
   eflExternalAnnot, // The file is embedded in a file attachment annotation
   eflParent,       // The file is located in the parent document
   eflParentAnnot   // The file is located in a file attachment annotation
}TEmbFileLocation;
```

The function creates an embedded GoTo action. This action type opens an embedded PDF file that is located in the current, parent, or in an external PDF file.

Embedded files can be attached directly, e.g. with AttachFile(), or embedded in file attachment annotations with FileAttachAnnot().

The parameter *Location* specifies where the embedded file can be found. File attachment annotations are stored in pages and not globally like ordinary file attachments. Therefore, the source page in

which the annotation can be found must be known if the target file is stored in a file attachment annotation (parameter *SrcPage*). The first page number is denoted by 1.

The parameter *Source* specifies the source file if the embedded file is located in an external PDF file. This parameter is only used if *Location* is set to eflExternal or eflExternalAnnot. It can be set to NULL in all other cases.

The parameter *Target* is required; it specifies either the name of the embedded PDF file or the name of the annotation in which the file is stored. The name of a file attachment annotation must be set with SetAnnotString(). The names of all annotations in a page must be unique. The function does not check whether the annotation or embedded file exists. Therefore, the target file can be embedded before or after the action was created.

The name of an embedded file is the file name if it was attached with AttachFile() or AttachFileEx(). The function extracts the file name automatically if a path was passed to the function.

It is not allowed to use different string formats in AttachFile() or SetAnnotString() and in the corresponding CreateGoToEAction() call. The same restriction applies to named destinations. If a file was inserted with the Unicode version then you must use the Unicode version of CreateGoToEAction() too. Otherwise the embedded file cannot be found.

The destination page that should be opened can be either specified directly with the parameter *DestPage* or with a Named Destination (see CreateNamedDest() for further information). Named destinations take precedence.

Remarks:

A path to an external PDF file should be defined as relative path. Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateGoToRAction

Syntax:

```
SI32 pdfCreateGoToRAction(
    const PPDF* IPDF,     // Instance pointer
    const char* FileName, // File path to external PDF file
    UI32 PageNum)         // Destination page
```

This function creates a go-to-remote action. A go-to-remote action opens an external PDF file and jumps to the page defined by the parameter *PageNum*. If the destination page does not exist the first page will be displayed.

Remarks:

The path to the external PDF file should be defined as relative path. Actions must be added to a PDF object with AddActionToObj(). This function is also implemented in a wide version that accepts an Unicode file path. However, Unicode file path are supported since PDF 1.7 (Acrobat 8). Earlier Acrobat versions will fail to open the file if the path contains non-Ansi characters.

Return value:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateGoToRActionEx

Syntax:

```
SI32 pdfCreateGoToRActionEx(
    const PPDF* IPDF,       // Instance pointer
    const char* FileName,   // File path to another PDF file
    char* DestName,         // Named Destination to access
    LBOOL NewWindow)        // If true, the file is opened in a new window
```

The function creates a go-to-remote action that opens a named destination in an external PDF file. The parameter *DestName* must be the name of a named destination located in the external PDF file. Note that this is a binary string which must be specified exactly and case-sensitive. If the name was originally defined as an Unicode string, the go-to remote action must use the same string format. Otherwise the destination cannot be found. See also CreateNamedDest().

Remarks:

The path to the external PDF file should be defined as relative path. Actions must be added to a PDF object with AddActionToObj(). This function is also implemented in a wide version that accepts an Unicode file path (CreateGoToRActionExU()). However, Unicode file path are supported since PDF 1.7 (Acrobat 8). Earlier Acrobat versions will fail to open the file if the path contains non-Ansi characters.

Return value:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateGroupField

Syntax:

```
SI32 pdfCreateGroupField(
   const PPDF* IPDF, // Instance pointer
   const char* Name, // Name of the field
   SI32 Parent)      // Parent group field if any or -1
```

This function creates a group field. A group field is a simple array of fields which can be used to separate fields into several groups. The field itself has no appearance, it is invisible.

For example, if an interactive form contains fields for private personal data and fields for company data in any kind, the fields can be structured into two base parts: *PersonalData* and *CompanyData*. The field names of each part must be unique inside its own hierarchy. However, duplicate field names of fields of the same type are still allowed.

Field names and Group fields

The name of a specific interactive form field is the partial field name. The fully qualified name is not explicitly defined, but is constructed from the partial field name of the field and all of its ancestors.

For a field with no parent, the partial field name and the fully qualified name are the same; for a field that is the child of a parent (group) field, the fully qualified name is formed by appending the child field's partial name to the parent's fully qualified name, separated by a period (.):

```
parent_field_name.child_partial_name
```

For example, if a group field with the partial field name *PersonalData* has a child whose partial field name is *Address*, which in turn has a child with the partial field name *ZipCode*, then the fully qualified name of this last field would be

```
PersonalData.Address.ZipCode
```

Thus all fields descended from a common ancestor will share the ancestor's fully qualified field name as a common prefix in their own fully qualified names.

The naming scheme must be taken into account when accessing fields of a group in JavaScript functions.

Remarks:

Acrobat 4 does not support group fields inside a hide or form action. For example, if a group field will be added to a hide action, the entire group becomes invisible if the action is executed in Acrobat 5 or higher. Acrobat 4 ignores the group field; each field of the group must be manually added to the hide action.

Return values:

If the function succeeds the return value is the field handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateHideAction

Syntax:

```
SI32 pdfCreateHideAction(
   const PPDF* IPDF, // Instance pointer
   UI32 AField,      // Field handle of a field that should be hidden
   LBOOL Hide)       // Hide or unhide the fields?
```

A hide action hides or shows one or more interactive form fields on screen by setting or clearing their hidden flags. The parameter *AField* must be a valid field handle. If the parameter *Hide* is true, the fields become invisible. If *Hide* is false, the fields become visible if their hidden flag was set before.

Remarks:

To add more fields to the action use the function AddFieldToHideAction(). Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateICCBasedColorSpace

Syntax:

```
SI32 pdfCreateICCBasedColorSpace(
   const PPDF* IPDF,      // Instance pointer
   const char* ICCProfile) // File path of an ICC profile
```

The function creates an ICC-based color space. ICC-based color spaces (PDF 1.3) are based on a cross-platform color profile as defined by the International Color Consortium (ICC). Unlike the CalGray, CalRGB, and Lab color spaces, which are characterized by entries in the color space dictionary, an ICC-based color space is characterized by a sequence of bytes in a standard format.

An ICC profile describes the color characteristics of a particular device and it provides necessary information to convert color data between native device color spaces and device independent color spaces. The ICC specification classifies color devices according to their use as input or output devices such as monitors, scanners, cameras or printers. Depending on the output PDF format different device classes are supported.

Supported device classes in PDF 1.3 or higher and PDF/A-1:

- scnr -> Scanners
- mntr -> Monitors
- prtr -> Printers
- spac -> Color conversion profile

Supported device classes in PDF/X-1 and PDF/X-3:

- `prtr -> Printers`

ICC profiles are available in different versions. Which profile versions are allowed to use depends on the output PDF version:

PDF Version	ICC Specification Version	ICC Profile Version Number
PDF 1.3	3.3 or earlier	2.10
PDF 1.4	ICC.1:1998-09 and its addendum ICC.1A:1999-04	2.20
PDF 1.5	ICC.1:2001-12	4.00
PDF 1.6	ICC.1:2003-09	4.10
PDF 1.7	ICC.1:2004-10	4.20

Please note that neither PDF/A-1 nor PDF/X-1 and PDF/X-3 support ICC profile major versions higher than 2. The functions CreateICCBasedColorSpace() and AddRenderingIntent() check the profile version only if the output version was set to a PDF/A or PDF/X compatible version (see SetPDFVersion()).

ICC profiles are available for different input color spaces such as DeviceGray, DeviceRGB, DeviceCMYK, or Lab. Color values must be defined in the very same as for the corresponding device color spaces.

Notice:

Non-device color spaces cannot be used for interactive objects such as annotations or form fields. The active color space must always be changed to a device color space before creating interactive objects; see SetColorSpace(). Note also that annotations support DeviceRGB only. Form fields support DeviceGray, DeviceRGB, and DeviceCMYK.

Remarks:

Use the function SetExtColorSpace() to activate the color space in the graphics state. To set a color of an ICC-based color space convert the color value to a 32 bit integer in the very same way as for DeviceGray, DeviceRGB, or DeviceCMYK depending on the base color space and pass the resulting color value to SetFillColor(), SetStrokeColor(), or SetColors().

Return values:

If the function succeeds the return value is the color space handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateICCBasedColorSpaceEx

Syntax:

```
SI32 CreateICCBasedColorSpaceEx(
    const PPDF* IPDF,
    const void* Buffer,
    UI32 BufSize)
```

The function creates an ICC-based color space exactly like CreateICCBasedColorSpace() but accepts a file buffer as input. See CreateICCBasedColorSpace() for further information.

Return values:

If the function succeeds the return value is the color space handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateImage

Syntax:

```
LBOOL pdfCreateImage(
    const PPDF* IPDF,      // Instance pointer
    const char* FileName, // Ouput file name or NULL
    TImageFormat Format)  // Output image format
typedef enum
{
    ifmTIFF,      // RGB, CMYK, Gray, B&W -> CCITT 3/4, JPEG, Flate, LZW
    ifmJPEG,      // RGB, CMYK, Gray -> JPEG compression
    ifmPNG,       // Gray, RGB, B&W  -> Flate compression
    ifmReserved, // Reserved for future extensions
    ifmBMP,       // Gray, RGB, B&W  -> Uncompressed
    ifmJPC        // RGB, CMYK, Gray -> JPEG2000 compression
}TImageFormat;
```

The function creates an empty image in the specified format. This is a helper function to create image files from the rasterizer or from raw images returned by the content parser. See ParseContent() or RenderPage() for further information. After the image was created one or more images can be added to it with AddImage(). Note that TIFF is the only supported multi-page format. All other formats support exactly one image.

If the parameter *FileName* is set to NULL the image is created in memory. In this case the image buffer is available after CloseImage() was called. To get the file buffer call the function GetImageBuffer(). Note that the internal resources of memory based images must be manually freed with FreeImageBuffer() in this case.

When the function is used to create images from the rasterizer, the supported pixel formats of the output image format must be considered. The following table specifies which combinations are valid:

Image Format	Supported Pixel Formats
BMP (Bitmap)	pxf1Bit, pxfGray, pxfBGR, pxfBGRA
JPEG	pxfGray, pxfRGB, pxfRGBA
JPEG 2000 (JPC)	pxf1Bit, pxfGray, pxfRGB
PNG	pxf1Bit, pxfGray, pxfRGB, pxfRGBA
TIFF	pxf1Bit, pxfGray, pxfRGB, pxfRGBA

Remarks:

CreateImage() can be used in combination with ParseContent() or RenderPage() and AddImage().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

CreateImportDataAction

Syntax:

```
SI32 pdfCreateImpDataAction(
    const PPDF* IPDF,      // Instance pointer
    const char* DataFile) // File path to FDF file
```

An import data action imports FDF data into the document's interactive form. FDF data files are normally created by Adobe's Acrobat when submitting a form to a web server.

Note that the XML based format XFDF is not supported by Adobe's Acrobat.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. However, because Acrobat does not support Unicode strings as file paths, the path is converted back to Ansi. A Unicode string must not contain characters outside of the code page 1252.

Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateIndexedColorSpace

Syntax:

```
SI32 pdfCreateIndexedColorSpace(
    const PPDF* IPDF,          // Instance pointer
    TExtColorSpace Base,       // Base color space
    SI32 Handle,               // Color space handle or -1 depending on Base
    const void* ColorTable,    // Array of colors
    UI32 NumColors)            // Number of colors in the array
```

The function creates an indexed color space which can be used for vector graphics and text output. The parameter *Base* defines the underlying base color space. It can be any device or ICC-based color space or separation space, but not a pattern space or another indexed space.

The parameter *Handle* must be a handle of the underlying base color space if no device color space is used. If a device color space is used the parameter *Handle* is ignored.

The color table must be defined as an array of color values defined in the corresponding base color space. For example, if the base color space is DeviceRGB, the values in the color table are to be interpreted as red, green, and blue components; if the base color space is a CIE-based *ABC* space such as a CalRGB or Lab space, the values are to be interpreted as *A*, *B*, and *C* components.

The number of components is taken from the base color space. The parameter *NumColors* defines the number of color values defined in the array. An indexed color space cannot contain more than 256 colors.

The length of the color table in bytes must be *NumColors * NumComponents*.

Notice:

Non-device color spaces cannot be used for interactive objects such as annotations or form fields. The active color space must always be changed to a device color space before creating interactive objects, see SetColorSpace(). Note also that annotations support DeviceRGB only. Form fields support DeviceGray, DeviceRGB, and DeviceCMYK.

Remarks:

This function does not activate the color space in the graphics state. Use the function SetExtColorSpace() to activate the color space in the graphics state. To set a color of the indexed color space pass the color index to SetFillColor(), SetStrokeColor(), or SetColors().

Return values:

If the function succeeds the return value is a color space handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateJSAction

Syntax:

```
SI32 pdfCreateJSAction(
   const PPDF* IPDF,   // Instance pointer
   const char* Script) // Must be one or more JavaScript functions
```

This function creates a JavaScript action. A JavaScript Action causes a script to be compiled and executed by the JavaScript interpreter of Adobe's Acrobat. Depending on the nature of the scripts, this can cause various interactive form fields in the document to update their values or change their visual appearances.

The parameter *Script* must be a valid JavaScript. The function does not check whether the script is valid.

A JavaScript action can be added to a field, page, or to the Catalog object with AddActionToObj().

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Because JavaScript 1.2 is not Unicode compatible, Unicode encoded scripts are translated to a platform specific encoding prior to interpretation by the JavaScript engine. This conversion is done by Adobe's Acrobat. Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateLaunchAction

Syntax:

```
SI32 pdfCreateLaunchAction(
   const PPDF* IPDF,      // Instance pointer
   TFileOP OP,            // Kind of operation (open or print the file)
   const char* FileName,  // File name (required)
   const char* DefDir,    // Default directory (can be NULL)
   const char* Param,     // Optional parameters (can be NULL)
   LBOOL NewWindow)       // Open the file in a new window?

typedef enum
{
   foOpen,
   foPrint
}TFileOP;
```

A launch action launches an application or opens or prints a document. Acrobat passes the parameters of a launch action directly to the API function ShellExecute(). The parameter *FileName* can be an absolute or relative file path but relative file paths are preferred.

The parameter *DefDir* defines a default directory in which the file should be searched. The parameter *Param* defines an optional parameter string which can be passed to the application. This parameter should be omitted if the *FileName* designates a document. The parameters *DefDir* and *Param* are both optional, they can be NULL. See also ShellExecute() of your Windows API reference for further information.

The last parameter *NewWindow* specifies whether the file should be opened in a new window or not. This parameter is ignored if the file is not a PDF file.

Remarks:

Adobe's Acrobat displays a warning before executing a launch action. Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateLaunchActionEx

Syntax:

```
SI32 pdfCreateLaunchActionEx(
   const PPDF* IPDF,     // Instance pointer
   const char* FileName, // Should be a relative path
   LBOOL NewWindow)      // Meaningful for PDF files only
```

The function creates a launch action. This type of a launch action can be used to open an arbitrary file or application on the users system. It supports no parameters like CreateLaunchAction() but it is also available in an Unicode compatible version. Unicode paths are supported since PDF 1.7 (Acrobat 8 or higher).

Remarks:

Adobe's Acrobat displays a warning before executing a launch action. Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateListBox

Syntax:

```
SI32 pdfCreateListBox(
    const PPDF* IPDF,  // Instance pointer
    const char* Name,  // Name of the list box
    LBOOL Sort,        // Sort the choice values?
    SI32 Parent,       // Parent group field if any or -1
    double PosX,       // X-Coordinate of the list box
    double PosY,       // Y-Coordinate of the list box
    double Width,      // Width of the list box
    double Height)     // Height of the list box
```

This function creates a list box. The choice values of the list box must be added with the function AddValToChoiceField(). This function requires the field handle that was returned by this function.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the list box. If the coordinate system is top-down it defines the upper left corner.

Like all form fields, the width and height is measured incl. the line width of the border. The size of normal vector graphics is measured without the line width; this must be taken into account when calculating the width or height of a form field. The line width of the border is derived from the current graphics state (see SetLineWidth()), it should be either 0, 1, 2, or 3 units (no border, thin, medium, or thick). The border style can be changed with the functions SetBorderStyle() or SetFieldBorderStyle().

The values of the list box are sorted if the parameter *Sort* is true. List boxes support multiple selected values. To enable multi selection add the flag ffMultiSelect to the field (see SetFieldFlags() for further information).

A list box requires a font. If no font is active the standard font Helvetica is used. Note that if the form must be compatible to Acrobat 4, the used fonts by the form fields must be restricted to the 14 standard fonts (see SetFont() for further information). Acrobat 4 does not support interactive forms that use other fonts than the 14 standard fonts. Such a form requires Acrobat 5 or higher.

Form fields support natively the code pages 1252 and MacRoman. However, other code pages and Unicode can be used too as long as a few external cmaps are available. See SetFieldFont() for further information.

The font size of a list box is set to "auto" if either the font size of the active font is set to 1.0 unit or if no font is active when the field is created.

A text field can be formatted and the allowed input values can be restricted to specific data formats. See SetDateTimeFormat(), SetNumberFormat() for further information.

Specific flags supported by list boxes:

- `ffSorted` `// Change the sort flag if necessary`
- `ffMultiSelect // Enable multi selection`

Remarks:

Interactive form fields can be structured into several groups by passing a handle of a group field to the parameter *Parent*. See CreateGroupField() for further information.

Return values:

If the function succeeds the return value is the handle of the combo box, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateNamedAction

Syntax:

```
SI32 pdfCreateNamedAction(
    const PPDF* IPDF,     // Instance pointer
    TNamedAction Action) // Kind of action that should be executed

typedef enum
{
    naFirstPage,     // PDF 1.2 Go to the first page of the document
    naLastPage,      // PDF 1.2 Go to the last page of the document
    naNextPage,      // PDF 1.2 Go to the next page
    naPrevPage,      // PDF 1.2 Go to the previous page
    naGoBack,        // Go back to last page and position
    naOpenDlg,       // Display the file open dialog
    naPrintDlg,      // Display the print dialog
    naGeneralInfo,   // Display the general info tab
    naFontsInfo,     // Display the fonts info tab
    naSaveAs,        // Display the save as dialog (requires Acrobat)
    naSecurityInfo, // Display the security settings
    naFitPage,       // Fit the page into the window
    naDeletePages,   // Delete one or more pages
    naQuit,          // Quit the application
    naUserDefined   // Internal value to store unknown imported values
}TNamedAction;
```

The function creates a named action. Only PDF 1.2 compatible actions are documented in the PDF Reference. So, all other actions may still work but there is no guarantee that these actions will be supported in future versions of Adobe's Acrobat or Reader. Actions must be added to a PDF object with AddActionToObj() so that they can be executed.

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateNamedDest

Syntax:

```
SI32 pdfCreateNamedDest(
   const PPDF* IPDF,   // Instance pointer
   const char* Name,   // Name of the destination (byte string)
   UI32 DestPage,      // Destination page
   TDestType DestType, // Destination type
   double a,           // Various, depends on destination type
   double b,           // Various, depends on destination type
   double c,           // Various, depends on destination type
   double d)           // Various, depends on destination type
```

The function creates a named destination that can be accessed from external PDF files. Named destinations are useful if the destination must be accessed from another PDF file. For example, a link to the beginning of Chapter 3 in another document might refer to the destination by a name, such as Chap3.begin, instead of an explicit page number in the other document. This makes it possible to change the destination in the document without invalidating the external link.

Named destinations can be used in the same document by bookmarks, page links, and go-to actions. Named destinations which are located in another document can be accessed with a go-to-remote action (see CreateGoToRActionEx() for further information).

The different destination types are described in detail at CreateGoToAction().

Remarks:

Make sure that the same string format is used in a named destination and in a go-to remote action that tries to access it. Note that the destination cannot be found if the string format is different, e.g. Ansi in the named destination and Unicode in the go-to remote action.

To avoid unnecessary issues it is usually best to use 7 bit Ansi strings for named destinations.

Return values:

If the function succeeds the return value is the destination handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateNewPDF

Syntax:

```
LBOOL pdfCreateNewPDF(
   const PPDF* IPDF,    // Instance pointer
   const char* OutPDF) // File name of the output file or NULL
```

This function creates a new PDF file. If the parameter *OutPDF* is an empty string the PDF file is created in memory. This function resets also the internal error flag if it was set before.

A memory based PDF file is available after CloseFile() or CloseFileEx() was called. To get the file buffer call the function GetBuffer(). Note that the internal used resources must be freed manually if a PDF was created in memory. See also GetBuffer().

It is also possible to open the output file with OpenOutputFile() right before CloseFile() is called. The advantage is that the entire PDF file can be processed before opening the output file. If an error occurred during processing, the creation of the output file can be discarded so that no empty file will be left on disk.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. On Windows operating systems Unicode file names are supported on NT based systems only (Win NT, 2000, XP, and so on). Non-Windows operating systems use UTF-8 to encode Unicode file paths. Therefore, the Ansi version should be used to open the file. If the Unicode version is used instead the file path is converted to UTF-8 and passed to the Ansi version of the function.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Examples (C):

Example 1: Create a PDF file in memory:

```
// First we declare an error callback function that is called if an
// error occurred.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}

int main(int argc, char* argv[])
{
   FILE* f; char* buffer; UI32 bufSize;
   PPDF* pdf = pdfNewPDF();
   if (!pdf) return 2; // Out of memory?
   // No need to check return values.
```

```c
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfSetDocInfo(pdf, diAuthor, "Jens Boschulte");
   pdfSetDocInfo(pdf, diCreator, "C sample project");
   pdfSetDocInfo(pdf, diSubject, "How to create a PDF file in memory?");
   pdfCreateNewPDF(pdf, NULL);

   pdfAppend(pdf);
   pdfSetFont(pdf, "Arial", fsItalic, 40.0, true, cp1252);
   pdfWriteFText(pdf, taCenter, "My first memory based PDF...");
   pdfEndPage(pdf);

   pdfCloseFile(pdf);
   // The file must be opened in binary mode!
   if ((f = fopen("c:/cout.pdf", "wb")) == NULL)
   {
      pdfDeletePDF(pdf);
      printf("Cannot open output file!");
      return 3;
   }

   buffer = pdfGetBuffer(pdf, &bufSize); // Get the file buffer
   if (buffer)
   {
      // We write the buffer as it is to the file.
      fwrite(buffer, 1, bufSize, f);
      fclose(f);
   }
   // We created a new PDF instance inside the function. When the
   // instance is deleted all used resources are freed. However, in most
   // cases it is a better way to use one PDF instance for multiple PDF
   // files to improve processing speed. In the latter case we must call
   // pdfFreePDF(pdf); to free the internal used resources.
   pdfDeletePDF(pdf); // Delete the PDF instance
   return 0;
}
```

Example 2: Usage of OpenOutputFile()

```c
int main(int argc, char* argv[])
{
   PPDF* pdf = pdfNewPDF();
   if (!pdf) return 2; // Out of memory?
   // No need to check return values.
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfSetDocInfo(pdf, diSubject, "Usage of OpenOutputFile()...");
   pdfCreateNewPDF(pdf, NULL);

   pdfAppend(pdf);
   pdfSetFont(pdf, "Arial", fsItalic, 40.0, true, cp1252);
```

```c
   pdfWriteFText(pdf, taCenter, "We use OpenOutputFile() now...");
   pdfEndPage(pdf);

   // No fatal error occurred?
   if (pdfHaveOpenDoc(pdf))
   {
      // OK, now we can open the output file.
      /* Note that the function can also be called in a while statement
       * e.g. to display a file open dialog if the file could not be
       * opened...
       */
      if (!pdfOpenOutputFile(pdf, "c:/test.pdf"))
      {
         pdfDeletePDF(pdf);
         printf("Cannot open output file!\n");
         return -1;
      }
      if (pdfCloseFile(pdf))
      {
         printf("PDF file successfully created!\n");
      }
   }
   pdfDeletePDF(pdf);
```

CreateOCG

Syntax:

```
SI32 pdfCreateOCG(
  const PPDF* IPDF,   // Instance pointer
  const char* Name,   // Layer name (required)
  LBOOL DisplayInUI,  // Display the OCG in the user interface?
  LBOOL Visible,      // Intitial state of the OCG
  TOCGIntent Intent)  // see below

typedef enum
{
   oiDesign = 2,
   oiView   = 4, // Default
   oiAll    = 8,
   oiEmpty  = 16 // Internal (refers to oiView if used)
}TOCGIntent;
```

The function creates an Optional Content Group (OCG). Content that belongs to an OCG can be made visible or invisible dynamically by users of conforming readers. OCGs are mostly called Layers because Optional Content and Layers share the same basic functionality. A layer is a piece of content that is associated with an OCG.

The parameter *DisplayInUI* specifies whether the OCG should be displayed in the user interface of a PDF viewer. The state of invisible layers (the ones which are not displayed in the user interface) can only be changed programmatically, e.g. with a SetOCGState or Javascript actions. Also if *DisplayInUI* is set to false, the parameter *Name* is still required.

If *DisplayInUI* is set to true the layer will be added to the root of the layer tree. To reflect nesting levels, layer groups and so on, it is also possible to configure the layer tree manually with AddLayerToDisplTree(). The parameter *DisplayInUI* should be set to false in this case so that the layer will not already be added to the tree.

The parameter *Visible* specifies the initial state of the OCG. If set to false, contents that belongs to this OCG remains invisible until the state will be changed due to interaction with the user interface or programmatically with a SetOCGState or Javascript action.

The parameter *Intent* specifies the intended use of the graphics in the group. It provides a way to distinguish between different intended uses of optional content. For example, many document design applications, such as CAD packages, offer layering features for collecting groups of graphics together and selectively hiding or viewing them for the convenience of the author. However, this layering may be different (at a finer granularity, for example) than would be useful to consumers of the document. Therefore, it is possible to specify different intents for optional content groups within a single document. A conforming reader may decide to use only groups that are of a specific intent.

The *Intent* entry can be used to create different configurations for viewing and design applications. However, this functionality is not yet implemented in DynaPDF.

How to make content optional?

Page contents, such as text, images, and vector graphics can be associated with an OCG with BeginLayer() / EndLayer(). Anything that is drawn between these function calls becomes part of the layer or OCG.

Interactive objects such as Annotations and Form Fields can be added to a layer with AddObjectToLayer(). However, interactive objects can be associated with exactly one layer at time. In cases where the visibility should depend on more than one OCG it is possible to connect the object with an Optional Content Membership Dictionary (OCMD). OCMDs accept an array of OCG handles and a visibility expression that defines when the associated content should become visible or invisible (see CreateOCMD() for further information). OCMDs can also be used with BeginLayer(), e.g. if too many nested BeginLayer() calls would be required to achieve the same result.

Images and templates can be associated with an OCG or OCMD too. This can be useful if the object must be drawn outside of the parent layers or to achieve more complex visibility expressions.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Unused OCGs will not be written to the PDF file.

Return values:

If the function succeeds the return value is the OCG handle, a value greater or equal zero. If the function fails the return value is a negative error code.

Example (C++):

```cpp
...
pdfCreateNewPDF(pdf, "test_layer.pdf");

// We use three layers in this example
SI32 oc1 = pdfCreateOCG(pdf, "Anything", true, true, oiAll);
SI32 oc2 = pdfCreateOCG(pdf, "Text and Annotations", true, true, oiAll);
SI32 oc3 = pdfCreateOCG(pdf, "Images", true, true, oiAll);

pdfSetPageCoords(pdf, pcTopDown);

pdfAppend(pdf);
// The main layer controls the visibility of all three layers in this
// example.
pdfBeginLayer(pdf, oc1);

   pdfBeginLayer(pdf, oc2);
      pdfSetFont(pdf, "Arial", fsRegular, 12.0, true, cp1252);
      char someText[] = "Some text with a link!!!";
      pdfWriteText(pdf, 50.0, 50.0, someText);
      double tw = pdfGetTextWidth(pdf, someText);
```

```
        // To reflect the same nesting as the text layer we must
        // use an OCMD for the annotation because the visibility of the
        // layer oc2 depends on oc1 at this position.
        pdfSetBorderStyle(pdf, bsUnderline);
        pdfSetStrokeColor(pdf, PDF_BLUE);
        SI32   annot = pdfWebLink(pdf, 50, 51, tw, 12, "www.dynaforms.com");
        UI32 ocgs[2] = {oc1, oc2};
        SI32     ocmd = pdfCreateOCMD(pdf, ovAllOn, ocgs, 2);
        pdfAddObjectToLayer(pdf, ocmd, ooAnnotation, annot);
    pdfEndLayer(pdf);

    pdfBeginLayer(pdf, oc3);
        InsertImageEx(pdf, 50.0, 70.0, 300.0, 200.0, "c:/Imgs/test.tif", 1);
    pdfEndLayer(pdf);

pdfEndLayer(pdf);

pdfWriteText(pdf, 50.0, 300.0, "This text is not part of a layer!");

pdfEndPage(pdf);

pdfCloseFile(pdf);
...
```

CreateOCMD

Syntax:

```
SI32 pdfCreateOCMD(
  const PPDF* IPDF,         // Instance pointer
  TOCVisibility Visibility, // When should the OCMD be visible?
  UI32* OCGs,               // Array of OCG handles (required)
  UI32 Count)               // Number of handles in the array (required)

typedef enum
{
   ovAllOff,
   ovAllOn,
   ovAnyOff, // Default
   ovAnyOn,
   ovNotSet  // Internal (defaults to ovAnyOn if used)
}TOCVisibility;
```

The function creates an Optional Content Membership Dictionary (OCMD). OCMDs can be used to create visibility expressions which depend on more than one optional content group.

OCMDs are mostly used with Annotations and Form Fields because these objects can only be associated with one OCG or one OCMD. In cases where the visibility should depend on more than one OCG or if the object should be used in different layers, an OCMD must be used.

The parameter *Visibility* specifies when the associated objects should become visible. The parameter *OCGs* must be an array of valid OCG handles. A viewer application inspects the array of OCGs and the visibility parameter to determine whether the associated content should be visible or invisible.

Remarks:

Unused OCMDs will not be written to the PDF file.

Return values:

If the function succeeds the return value is the OCMD handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateRadialShading

Syntax:

```
SI32 pdfCreateRadialShading(
   const PPDF* IPDF, // Instance pointer
   double sX,        // X-Coordinate of the inner circle
   double sY,        // Y-Coordinate of the inner circle
   double R1,        // Radius of the inner circle
   double eX,        // X-Coordinate of the outer circle
   double eY,        // Y-Coordinate of the outer circle
   double R2,        // Radius of the outer circle
   double SCenter,   // Shading center
   UI32 SColor,      // Start color
   UI32 EColor,      // End color
   LBOOL Extend1,    // Extend the inner circle
   LBOOL Extend2)    // Extend the outer circle
```

Radial shadings define a color blend or gradient that varies between two circles. The shading may optionally be extended beyond the starting or ending circles by continuing the boundary colors indefinitely. Shadings of this type are commonly used to depict three-dimensional spheres and cones.

The shading center defines the point from where the first color will blend into the other. A value of 1 determines the exact center between the start and end point of the shading. Smaller values shift the shading center in direction to the start circle, greater values to the end circle.

Radial shadings can be drawn into a clipping path to restrict painting into this path. If the shading is drawn outside of a clipping path it is applied to the entire page. Not that extended shadings are opaque, objects behind the shading become invisible if they are overprinted by the shading.

Shadings are drawn by using the current coordinate system. It is recommended to understand that shadings have its own dimension like a normal shape. The parameters *Extend1* and *Extend2* extend the shading beyond its dimension. If the shading is extended it must normally be drawn into a clipping path to avoid overprinting of other objects.

Shadings support any device, ICCBased, and special color spaces like DeviceN or Separation. The color values of the start and end color must be defined in the current color space. See also SetColorSpace(), SetExtColorSpace().

This function creates a shading object but it doesn't draw it on the page. The shading can be drawn with ApplyShading().

Return values:

If the function succeeds the return value is the shading handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateRadioButton

Syntax:

```
SI32 pdfCreateRadioButton(
    const PPDF* IPDF,      // Instance pointer
    const char* Name,      // Name of the radio button
    const char* ExpValue, // Export value of the first check box
    SI32 Checked,          // If true, the check box appears checked
    SI32 Parent,           // Parent group field if any or -1
    double PosX,           // X-Coordinate of the first check box
    double PosY,           // Y-Coordinate of the first check box
    double Width,          // Width of the first check box
    double Height)         // Height of the first check box
```

A radio button field is a set of related toggle controls (check boxes), at most one of which may be on at any given time; selecting any one of the button automatically deselects all others.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the first check box that is automatically created by this function. If the coordinate system is top-down it defines the upper left corner.

Like all form fields, the width and height is measured incl. the line width of the border. The size of normal vector graphics is measured without the line width; this must be taken into account when calculating the width or height of a form field. The line width of the border is derived from the current graphics state (see SetLineWidth()), it should be either 0, 1, 2, or 3 units (no border, thin, medium, or thick). The border style can be changed with the functions SetBorderStyle() or SetFieldBorderStyle().

This field type supports two additional flags: ffNoToggleToOff and ffRadioIsUnion. The flag ffNoToggleToOff has no more any effect in Adobes Acrobat or Reader. The radio button behaves always as if the flag ffNoToggleToOff is set.

If the flag ffRadioIsUnion is set, and if two or more check boxes contain the same export value, then these check boxes will be selected in unison.

Remarks:

It is not possible to add an action to a radio button field. Actions must be added to the check boxes which are included in the radio button. The handle of the first check box can be calculated by adding 1 to the handle value of the radio button field. For example, if the handle of the radio button is 1, then the handle of the first check box is 2.

Return values:

If the function succeeds the return value is the handle of the radio button, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateRasterizer (Rendering Engine)

Syntax:

```
IRAS* rasCreateRasterizer(
   const PPDF* IPDF,      // PDF Instance pointer
   BYTE** Rows,           // Pointer to a row buffer or alternatively
   BYTE*  Buffer,         // Pointer to an image buffer that was allocated
                          // as one large buffer.
   UI32   Width,          // Width in Pixel
   UI32   Height,         // Height in Pixel
   SI32   ScanlineLen,    // Scanline length in bytes
   TPDFPixFormat PixFmt)  // Pixel format in which the buffer is defined

typedef enum
{
   pxf1Bit, // Defined for future use, not supported yet.
   pxfGray, // 8 Bit gray
   pxfRGB,  // RGB
   pxfBGR,  // BGR -> Pixel format for Windows Bitmaps
   pxfRGBA, // RGBA
   pxfBGRA, // BGRA -> Pixel format for Windows Bitmaps
   pxfARGB, // ARGB
   pxfABGR  // ABGR
   pxfCMYK, // CMYK
   pxfCMYKA // CMYKA
}TPDFPixFormat;
```

The function creates a new instance of a rasterizer. The caller is responsible to release the rasterizer when finish with DeleteRasterizer().

The rasterizer renders objects, e.g. a PDF page (see RenderPage()), into the supplied image buffer. Once the object was rendered, the result can be displayed on screen or saved on disk in an appropriate image format.

The image buffer can be changed if necessary with AttachImageBuffer(). This can be useful when the size of the image must be changed, e.g. when the user changes the window size. The overhead to attach a larger or smaller image buffer is minimal, and hence, very fast. So, it is mostly not required

to allocate the image in the largest size that a user can request, e.g. in a viewer application. The image can be reallocated in the required size and attached to the rasterizer whenever necessary.

The pixel format is directly connected with the rasterizer and cannot be changed at runtime. When the pixel format must be changed then delete the rasterizer and create a new one in that pixel format.

The image buffer can be either defined as one large memory block or as an array of scanlines. When it was allocated as one large block then set the parameter *Rows* to NULL and pass the buffer to the parameter *Buffer*. If the scanlines were allocated separately or in blocks then use the parameter *Rows* instead. One of these parameters must be set but never both. The function checks first whether the parameter *Rows* is present.

The rasterizer requires no special alignment of the scanlines, the scanlines can be byte aligned.

However, the scaline length can be longer than necessary, e.g. due to aligment requirements of certain image formats like Windows Bitmaps. A negative value of the scanline length mirrors the image vertically.

The pointer of a PDF instance is required because the rasterizer uses the error log of the PDF instance to output warnings and errors.

The return value of the function is a pointer to the rasterizer object. This pointer is required in functions like RenderPage().

Remarks:

The rasterizer uses the error log of the PDF instance to output warnings and error messages but it does never raise PDF exceptions. So, an error that occurs during rendering does not affect the PDF file in memory.

Return values:

When the function succeeds the return value is the pointer of the rasterizer object. If the function fails the return value is NULL.

CreateRasterizerEx (Rendering Engine)

Syntax:

```
IRAS* rasCreateRasterizerEx(
   const PPDF* IPDF,      // Instance pointer
   const void* DC,        // Device Context (HDC)
   UI32 Width,            // Width in Pixel
   UI32 Height,           // Height in Pixel
   TPDFPixFormat PixFmt)  // Wished pixel format
```

The function creates a new instance of a rasterizer. The caller is responsible to release the rasterizer when finish with DeleteRasterizer().

The difference in comparison to CreateRasterizer() is that this version creates a DIB Section in the specified size and pixel format that is compatible to a device context. A DIB Section is required to render into a device context with RenderPageEx(). The image buffer is released when the rasterizer is deleted.

The function supports the pixel formats pxf1Bit, pxfGray, pxfBGR, and pxfBGRA. The pixel format pxfBGRA should only be used when the page is rendered over an arbitrary background image. The default pixel format on Windows is pxfBGR.

The pixel format is directly connected with the rasterizer and cannot be changed at runtime. When the pixel format must be changed then delete the rasterizer and create a new one in that pixel format.

The size of the image buffer can be changed at runtime with ResizeBitmap().

Remarks:

At time of publication this function can be used on Windows only.

The rasterizer uses the error log of the PDF instance to output warnings and error messages but it does never raise PDF exceptions. So, an error that occurs during rendering does not affect the PDF file in memory.

Return values:

When the function succeeds the return value is the pointer of the rasterizer object. If the function fails the return value is NULL.

CreateRectilinearMeasure

Syntax:

```
SI32 pdfCreateRectilinearMeasure(
   const PPDF* IPDF,              // Instance pointer
   UI32 Viewport,                 // Viewport handle (required)
   double Scale,                  // Scaling factor (required)
   const char* UnitDescription,   // Unit description, e.g. "Inch to mm"
   const char* LabelDistance,     // Label for the distance tool, e.g. "mm"
   const char* LabelArea)         // Label for the area tool, e.g. "mm²"
```

The function creates a rectilinear measure dictionary. This kind of measure dictionary can be used to convert PDF units to an arbitrary other unit like millimeters, meters, feets, and so on.

A measure dictionary is used by the measure tool of Adobe's Acrobat and by other PDF viewers which contain a PDF compatible measure tool.

The parameter *Viewport* must be a viewport handle that was returned by CreateViewport(). *Scale* is the required scaling factor to convert a PDF unit (1/72 inch) to the wished destination unit. The parameters *UnitDescription*, *LabelDistance*, and *LabelArea* are all required and cannot be NULL.

UnitDescription is a description of the coordinate conversion, e.g. "Inch to mm".

LabelDistance is the label for the distance tool, e.g. "mm".

LabelArea is the label for the area tool, e.g. "mm²".

Example (C++):

```cpp
PPDF* pdf = pdfNewPDF();
if (!pdf) return -1; // Out of memory?
pdfCreateNewPDF(pdf, "e:/cppout.pdf");
   // We use top down coordinates in this example
   pdfSetPageCoords(pdf, pcTopDown);
   pdfAppend(pdf);
      // Just a test object of a known size that can be measured...
      pdfSetLineWidth(pdf, 0.5);
      pdfRectangle(pdf, 50.0, 50.0, 144, 72.0, fmStroke);

      TPDFRect bbox;
      pdfGetBBox(pdf, pbMediaBox, bbox);
      SI32 vp = pdfCreateViewport(
         pdf,               // Instance pointer
         NULL,              // Optional viewport name
         bbox.Left,         // Lowever left corner
         bbox.Bottom,       // Lowever left corner
         bbox.Right,        // Upper right corner
         bbox.Top);         // Upper right corner

      pdfCreateRectilinearMeasure(
         pdf,               // Instance pointer
         vp,                // Viewport handle
         25.4/72.0,         // Conversion factor from 1/72 inch to mm
         "inch to mm",      // Description of the measure dictionary
         "mm",              // Unit label for the distance tool
         "mm²");            // Unit label for the area tool
   pdfEndPage(pdf);
pdfCloseFile(pdf);
pdfDeletePDF(pdf);
```

Remarks:

This function is implemented in an Ansi and Unicode compatible variant.

Return values:

If the function succeds the return value is a measure handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateResetAction

Syntax:

```
SI32 pdfCreateResetAction(
    const PPDF* IPDF) // Instance pointer
```

A reset form action resets all or specific fields of an interactive form to their default values. If only a few specific fields should be reset then add these fields with AddFieldToFormAction() to the action. It is possible to exclude or include only specific fields when resetting the form. However, if no specific fields are added to the action, all fields are reset to their default values, this is the normal case.

Remarks:

Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateSeparationCS

Syntax:

```
SI32 pdfCreateSeparationCS(
  const PPDF* IPDF)          // Instance pointer
  const char* Colorant,      // Colorant name -> required
  TExtColorSpace Alternate,  // Alternate color space
  SI32 Handle,               // Alternate color space handle or -1
  UI32 Color)                // Color value defined in the alt. color
space
```

The function creates a Separation color space.

Color output devices produce full color by combining primary or process colorants in varying amounts. On an additive color device such as a display, the primary colorants consist of red, green, and blue phosphors; on a subtractive device such as a printer, they typically consist of cyan, magenta, yellow, and sometimes black inks. In addition, some devices can apply special colorants, often called spot colorants, to produce effects that cannot be achieved with the standard process colorants alone. Examples include metallic and fluorescent colors and special textures.

When printing a page, most devices produce a single composite page on which all process colorants (and spot colorants, if any) are combined. However, some devices, such as imagesetters, produce a separate, monochromatic rendition of the page, called a separation, for each colorant. When the separations are later combined—on a printing press, for example—and the proper inks or other colorants are applied to them, the result is a full-color page.

A Separation color space (PDF 1.2) provides a means for specifying the use of additional colorants or for isolating the control of individual color components of a device color space for a subtractive

device. When such a space is the current color space, the current color is a single-component value, called a tint, that controls the application of the given colorant or color components only.

The parameter *Colorant* specifies the colorant name which this Separation color space is intended to represent. The special colorant name *All* refers collectively to all colorants available on an output device, including those for the standard process colorants. When a Separation space with this colorant name is the current color space, painting operators apply tint values to all available colorants at once. This is useful for purposes such as painting registration targets in the same place on every separation.

Such marks are typically painted as the last step in composing a page to ensure that they are not overwritten by subsequent painting operations.

The special colorant name *None* never produces any visible output. Painting operations in a Separation space with this colorant name have no effect on the current page.

The parameter *Alternate* specifies the alternate color space in which the color should be rendered if the device does not support the specified colorant. The alternate color space can be any Device or CIE based color space but not in turn a special color space like Separation, Indexed, or DeviceN.

The parameter *Handle* specifies the handle of the alternate color space if a CIE based color space should be used. If the alternate color space is a device color space the parameter *Handle* is ignored.

The parameter *Color* specifies the alternate color value, defined in the alternate color space, which is used if the device does not support the colorant.

A color value in a Separation color space consists of a single tint component in the range 0 to 255. The value 0 represents the minimum amount of colorant that can be applied; 255 represent the maximum. Tints are always treated as subtractive colors, even if the device produces output for the designated component by an additive method. Thus, a tint value of 0 denotes the lightest color that can be achieved with the given colorant, and 255 is the darkest. This convention is the same as for DeviceCMYK color components but opposite to the one for DeviceGray and DeviceRGB.

Encoding of Colorant Names

Colorant names are interpreted in the code page 1252 (WinAnsi) by default. Because colorant names are stored in UTF-8 Unicode format in PDF, it is also possible to pass UTF-8 encoded Unicode strings to the function. However, the function must be able to distinguish between both string formats. To achieve this, the parameter *Handle* accepts the special flag 0x10000000 that specifies that the parameter *Colorant* contains a UTF-8 encoded string. The flag must be combined with the handle as follows:

```
handle |= 0x10000000;          // C/C++, C#
handle  = handle Or &H10000000 // Visual Basic
handle := handle or $10000000  // Delphi
```

If the alternate color space is a device color space, simply set the parameter to 0x10000000 to specify that the colorant name is a UTF-8 string.

Notice:

Non-device color spaces cannot be used for interactive objects such as annotations or form fields. The active color space must always be changed to a device color space before creating interactive objects; see SetColorSpace(). Note also that annotations support DeviceRGB only. Form fields support DeviceGray, DeviceRGB, and DeviceCMYK.

Remarks:

Use the function SetExtColorSpace() to activate the color space in the graphics state. To set a color of a Separation space pass the wished color value to SetFillColor(), SetStrokeColor(), or SetColors().

Return values:

If the function succeeds the return value is the color space handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateSetOCGStateAction

Syntax:

```
SI32 pdfCreateSetOCGStateAction(
  const PPDF* IPDF, // Instance pointer
  UI32* On,         // Array of OCG handles which should be set to On
  UI32  OnCount,    // Number of OCGs handles in the On array
  UI32* Off,        // Array of OCG handles which should be set to Off
  UI32  OffCount,   // Number of OCG handles in the Off array
  UI32* Toggle,     // Array of OCG handles which should toggle the state
  UI32  ToggleCount,// Number of OCG handles in the Toggle array
  LBOOL PreserveRB) // Preserve radio button relationships if any?
```

The function creates a SetOCGState action that can be used to change the visibility state of certain Optional Content Groups (OCGs). All arrays of OCG handles are optional but at least one array must be provided. The parameter *PreserveRB* should be set to true to preserve radio button state relashionships, if any. If set to false, radio button state relashionships will be ignored when setting the visibility state of affected OCGs.

Actions must be added to a PDF object so that they can be executed. See AddActionToObj() for further information and the example on the next page.

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

Example (C++):

This example creates a document in two languages. The text for the different languages is placed into separate layers. A JavaScript Action selects the correct layer depending on the viewer language when opening the file. It is also possible to change the language with two bookmarks. The

bookmarks are connected with a SetOCGState action that simply toggles the layer states from on to off or vice versa.

```c
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode,
                       const char* ErrMessage,
                       SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0; // We try to continue
}

int main(int argc, char* argv[])
{
   SI32 oc1, oc2, bmkDE, bmkEN, act, retval = -2;
   PPDF* pdf = pdfNewPDF();
   if (!pdf) return -1; // Out of memory?

   // Pass warnings and errors to this error callback function
   pdfSetOnErrorProc(pdf, NULL, PDFError);

   pdfCreateNewPDF(pdf, NULL);

   oc1 = pdfCreateOCG(pdf, "Deutsch", true, false, oiAll);
   oc2 = pdfCreateOCG(pdf, "English", true, true, oiAll);

   bmkDE = pdfAddBookmark(pdf, "Deutsch", -1, 1, false);
   bmkEN = pdfAddBookmark(pdf, "English", -1, 1, false);

   UI32 toggle[2] = {oc1, oc2};
   // Simply toggle the state from On to Off or vice versa
   act = pdfCreateSetOCGStateAction(pdf,NULL,0,NULL,0,toggle,2,true);

   // Add the action to the bookmarks
   pdfAddActionToObj(pdf, otBookmark, oeOnMouseUp, act, bmkDE);
   pdfAddActionToObj(pdf, otBookmark, oeOnMouseUp, act, bmkEN);

   pdfSetPageCoords(pdf, pcTopDown);

   pdfAppend(pdf);

      pdfSetFont(pdf, "Arial", fsRegular | fsItalic, 12.0, true, cp1252);

      // Text for the German layer
      pdfBeginLayer(pdf, oc1);
         pdfWriteFTextEx(
            pdf,
            50.0,
            50.0,
            pdfGetPageWidth(pdf)  - 100.0,
```

```
            pdfGetPageHeight(pdf) - 100.0,
            taLeft,
            "Dieses Beispiel zeigt wie ein mehrsprachiges Dokument "
            "erzeugt werden kann.\n\n"
            "Zunächst wird beim Öffnen des Dokuments die Sprache mit "
            "einer JavaScript Aktion eingestellt.\n\n"
            "Zusätzlich kann die Sprache auch über zwei Lesezeichen "
            "ausgewählt werden. Hierbei wird lediglich ein Layer ein- "
            "bzw. ausgeblendet.\n\n"
            "Der Seiteninhalt muss natürlich zweimal erzeugt werden, "
            "einmal in Deutsch und einmal in Englisch in diesem "
            "Beispiel, jeweils in unterschiedlichen Layern.");

    pdfEndLayer(pdf);

    // Text for the English layer
    pdfBeginLayer(pdf, oc2);
       pdfWriteFTextEx(pdf,
            50.0,
            50.0,
            pdfGetPageWidth(pdf)  - 100.0,
            pdfGetPageHeight(pdf) - 100.0,
            taLeft,
            "This example shows how a multi-language document can be "
            "created.\n\n"
            "The language is initially selected with a JavaScript "
            "Action when opening the file.\n\n"
            "Additionally, the wished language can be selected with two "
            "bookmarks. The bookmarks simply hide or unhide a layer.\n\n"
            "The page contents must of course be created twice, one time "
            "in English and one time in German in this example, but in "
            "different layers.");

    pdfEndLayer(pdf);

pdfEndPage(pdf);

// This script displays the correct layer depending on the viewer
// language.
SI32 actLang = pdfCreateJSAction(pdf,
    "if (app.viewerVersion >= 6.0)\n"
    "{\n"
    "    var ocgArray = this.getOCGs();\n"
    "    var de = (app.language == \"DEU\");\n"
    "    for (var i = 0; i < ocgArray.length; i++)\n"
    "    {\n"
    "        if(ocgArray[i].name==\"English\")\n"
    "        {\n"
    "            ocgArray[i].state = !de;\n"
```

```
    "          }else\n"
    "          {\n"
    "              ocgArray[i].state = de;\n"
    "          }\n"
    "      }\n"
    "}");

    pdfAddActionToObj(pdf, otCatalog, oeOnOpen, actLang, -1);

    if (pdfHaveOpenDoc(pdf))
    {
        if (pdfOpenOutputFile(pdf, "test_layer.pdf"))
        {
            retval = pdfCloseFile(pdf);
        }
    }
    pdfDeletePDF(pdf);
    return retval;
}
```

CreateSigField

Syntax:

```
SI32 pdfCreateSigField(
    const PPDF* IPDF, // Instance pointer
    const char* Name, // Name of the signature field (required)
    SI32 Parent,      // Parent group field if any or -1
    double PosX,      // X-Coordinate of the field
    double PosY,      // Y-Coordinate of the field
    double Width,     // Width of the field box
    double Height)    // Height of the field
```

This function creates an empty signature field which can be used to digitally sign the PDF file. If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the signature field. If the coordinate system is top-down it defines the upper left corner.

Like all form fields, the width and height is measured incl. the line width of the border. The size of normal vector graphics is measured without the line width; this must be taken into account when calculating the width or height of a form field. The line width of the border is derived from the current graphics state (see SetLineWidth()), it should be either 0, 1, 2, or 3 units (no border, thin, medium, or thick). The border style can be changed with the functions SetBorderStyle() or SetFieldBorderStyle().

If the PDF file should be digitally signed by DynaPDF call the function CloseAndSignFile() or CloseAndSignFileEx() after all pages has been created. The first signature field is used for signing if multiple signature fields exist. In this case it is also possible to create a user defined appearance for the signature field. See CreateSigFieldAP() for further information.

A signature field can be visible or invisible depending on your requirements. To create a hidden signature field set the flag ffHidden to the field (see also SetFieldFlags()).

How to lock an Interactive Form after signing?

If an Interactive Form should be filled in and signed within the full version of Adobe's Acrobat 4 or higher then it is possible to create an empty signature field which executes a special JavaScript Action to lock the form after it has been signed. This can be done with a JavaScript Action which contains the JavaScript function *AFSignature_Format()*. This function can be called within the *OnFormat* event of the signature field. Note that this is the one and only event that signature fields support.

The function supports two parameters, the first parameter defines whether all or only specific fields should be locked, the second parameter contains either an empty array or an array of full qualified field names delimited by a comma:

- `AFSignature_Format("ALL", new Array ("")");`
- `AFSignature_Format("THESE", new Array ("Field1, Field2")");`
- `AFSignature_Format("EXCEPT", new Array ("Field1, Field2")");`

Notice: If a form contains a button with a Submit Form Action then it is recommended to exclude the button field from locking so that it is still possible to submit the form data. Otherwise, the entire form is locked, incl. the submit button, and it is impossible to submit the form data to a web server.

Example (C++):

In this example we create a simple form with only one text field and an empty signature field. The form should be signed by the user after it has been filled in and then submitted to a web server. The submit button is hidden when the form is created; it becomes visible after the document has been signed. Note that this example works only with the full version of Adobe's Acrobat 5 or higher.

```cpp
#include "dynapdf.h"
using namespace DynaPDF;
// Error callback function.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return -1; // We break processing if an error occurs
}
int main(int argc, char* argv[])
{
   PPDF* pdf = pdfNewPDF();
   if (!pdf) return 2; // Out of memory?
   char outFile[] = "c:/cppout.pdf";
   // Error messages and warnings are passed to the callback function.
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   if (!pdfCreateNewPDF(pdf, outFile))
   {
      pdfDeletePDF(pdf);
      (void)getch();
      return 1;
   }
   pdfAppend(pdf);
   pdfSetPageCoords(pdf, pcTopDown);
   pdfCreateTextField(pdf, "Test", -1, false, 0, 50.0, 50.0, 200.0, 16.0);
   SI32 sig = pdfCreateSigField(pdf,"Signature",-1,50.0,70.0,200.0,30.0);
   // We exclude the button from locking, otherwise it would be impossible
   // to execute the submit form action. The submit button becomes
   // visible after the document has been signed.
   char s[] = "AFSignature_Format(\"EXCEPT\", new Array(\"Submit\"));\n"
   "var f = this.getField(\"Submit\");\nf.hidden=false;\n";
   SI32 act = pdfCreateJSAction(pdf, s);
   pdfAddActionToObj(pdf, otField, oeOnFormat, act, sig);
   pdfSetBorderStyle(pdf, bsBevelled);
   pdfSetFieldBackColor(pdf, PDF_SILVER);
   SI32 btn = pdfCreateButton(pdf,"Submit","Submit",1,50,110,100, 20);
   pdfSetFieldFlags(pdf, btn, ffHidden, false);
```

```
    // The entire PDF file should be submitted to our web server.
    act = pdfCreateSubmitAction(pdf,sfPDF,"http://www.test.com/pdf.php");
    pdfAddActionToObj(pdf, otField, oeOnMouseUp, act, btn);
    pdfEndPage(pdf);
    pdfCloseFile(pdf);
    pdfDeletePDF(pdf);
    printf("PDF file %s successfully created!\n", outFile);
    return 0;
}
```

Remarks:

The height of a signature field should be large enough to display at least 3 lines of text. If the optional strings "Location" and "Reason" of the function CloseAndSignFile() should be set, set the field height to a value so that up to 5 lines of text can be drawn into the field. The resulting font size depends on the height of the field divided by the number of text lines.

Interactive form fields can be structured into several groups by passing a handle of a group field to the parameter *Parent*. See CreateGroupField() for further information.

Return values:

If the function succeeds the return value is the field handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateSigFieldAP

Syntax:

```
SI32 pdfCreateSigFieldAP(
    const PPDF* IPDF) // Instance pointer
    UI32 SigField)    // Handle of a signature field
```

The function creates a signature appearance template in the exact size of the base signature field. The created template is already opened if the function succeeds and you can draw arbitrary contents into it. After the appearance has been defined the template must be closed with EndTemplate().

Note that the template is reserved for the signature field. It must not be placed on pages or other templates.

When creating a user defined appearance for a signature field you should also place the validation icon properly so that it does not overprint the entire contents. See also PlaceSigFieldValidateIcon().

Remarks:

An example application that demonstrates the creation of a user defined signature appearance is delivered with DynaPDF (see example signature_ap).

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateSoftMask

Syntax:

```
void* pdfCreateSoftMask(
   const PPDF* IPDF,    // Instance pointer
   UI32 TranspGroup,    // Handle of the transparency group (required)
   TSoftMaskType Type,  // Soft mask type, see description
   UI32 BackColor)      // Background color if the type is smtLuminosity

typedef enum
{
   smtAlpha,
   smtLuminosity
}TSoftMaskType;
```

The function creates a soft mask from a transparency group, see BeginTransparencyGroup() for further information. A soft mask represents an alpha channel that can be applied on vector graphics, texts, and images.

The creation of a soft mask begins with the creation of a transparency group. The rendered result of the transparency group represents finally the soft mask or alpha channel.

The color space of the transparency group should be device gray or a gray ICC based color space. It is also possible to define the group in RGB or CMYK but this is not really meaningful since the result must always be converted to gray.

Soft mask types

The parameter *Type* specifies the type of the soft mask. If the type is smtAlpha the mask is computed from the group's alpha channel, disregarding any color in the group. This type of soft mask works like a clipping path. Any pixel that is drawn in the group makes the mask transparent, uncovered areas remain invisible. There are not many cases for which this kind of masking can be useful. It is mostly used for images if the source image format did not support an alpha channel or if the mask was stored in a separate image file.

If the type is smtLuminosity the group is composited with a fully opaque backdrop as specified by the parameter *BackColor* (the background color must be specified in the group's color space). The mask is finally computed from the luminosity of the resulting color at each point in the group. This enables the definition of the mask with arbitrary vector graphics, text, shadings, and images. The underlying transparency group must be isolated since a non-isolated group would be initialized with the current backdrop and this conflicts with the properties of the soft mask.

This is the most common type of a soft maks since it works like a real alpha channel with the advantage that it can be created with vector graphics and especially shadings.

Possible rendering issues

A soft mask is designed to mask one object at time. Although it is possible to draw arbitrary objects when a soft mask is active, the result is maybe not what is intended because the effect on overlapping objects is as if the mask would be applied twice. If multiple objects must be drawn then these objects are usually placed into a transparency group since a transparency group is rendered as a hole and hence avoids issues with overlapping objects.

How to activate a soft mask?

Once the soft mask was created it can be activated and deactivated with an extended graphics state:

Example:

```
// Create the transpareny group
...
SI32 grp = pdfBeginTransparancyGroup(pdf, ...);
...
pdfEndTemplate(pdf);

// A soft mask can only be activated with an extended graphics state.
TPDFExtGState gs;
pdfInitExtGState(&gs);

// Create the soft mask from the group and set it in the graphics state
// object. You can change more settings of the graphics state if
// necessary...
gs.SoftMask = pdfCreateSoftMask(pdf, grp, smtLuminosity, 0);

// Create the extended graphics state now
SI32 extGState = pdfCreateExtGState(pdf, &gs);
// And activate it...
pdfSetExtGState(pdf, extGState);
// The soft mask is now active, draw something on the page
pdfInsertImage(pdf, ...);

// The only way to deactivate a soft mask is to set a second extended
// graphics state that disables the soft mask.
gs.SoftMask     = NULL;
gs.SoftMaskNone = true;
SI32 restoreSoftMask = pdfCreateExtGState(pdf, &gs);
pdfSetExtGState(pdf, restoreSoftMask);
...
```

Return values:

If the function succeeds the return value is a pointer to the soft mask object. This pointer is required to create an extended graphics state so that the mask can be activated. If the function fails the return value is NULL.

CreateStdPattern

Syntax:

```
SI32 pdfCreateStdPattern(
   const PPDF* IPDF,    // Instance pointer
   TStdPattern Pattern, // Kind of pattern
   double LineWidth,    // Line width to draw the pattern
   double Distance,     // Distance between the lines
   UI32 LineColor,      // Color of the lines
   UI32 BackColor)      // Background color or NO_COLOR if transparent

typedef enum
{
   spHorizontal, /* ----- */
   spVertical,   /* ||||| */
   spRDiagonal,  /* \\\\\ */
   spLDiagonal,  /* ///// */
   spCross,      /* +++++ */
   spDiaCross    /* xxxxx */
}TStdPattern;
```

This function creates a hatch pattern. Several types of hatch pattern can be created with this function. The line width, line distance, and line color can be set individually. If the pattern should be transparent set the background color to NO_COLOR.

The function creates a colored tiling pattern. See BeginPattern() for further information.

A pattern can be used like a color, it is not required to save the graphics state before applying a pattern. A pattern remains active until the corresponding color is changed with SetFillColor() or SetStrokeColor().

Remarks:

Patterns are invisible as long as they are not applied by the function ApplyPattern().

Return values:

If the function succeeds the return value is the pattern handle, a value greater or equal zero. If the function fails the return value is a negative error code.

Example (C):

```
// First we declare an error callback function.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}
```

```cpp
int main(int argc, char* argv[])
{
   SI32 pat1, pat2, pat3;
   PPDF* pdf = pdfNewPDF();
   if (!pdf) return 2; // Out of memory?
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfSetDocInfo(pdf, diSubject, "Standard hatch patterns");
   pdfSetDocInfo(pdf, diCreator, "C example test project");
   pdfSetDocInfo(pdf, diTitle, "Standard hatch patterns");
   pdfSetPageCoords(pdf, pcTopDown);

   pdfCreateNewPDF(pdf, "c:/cppout.pdf");

   pdfAppend(pdf);

   pat1 = pdfCreateStdPattern(pdf, spCross,2,8,PDF_RED,NO_COLOR);
   pat2 = pdfCreateStdPattern(pdf, spDiaCross,1,4,PDF_BLUE,PDF_YELLOW);
   pat3 = pdfCreateStdPattern(pdf, spDiaCross,1,4,PDF_BLUE,NO_COLOR);

   pdfApplyPattern(pdf, pat1, cmFill, 0);
   pdfApplyPattern(pdf, pat2, cmStroke, 0);
   pdfSetLineWidth(pdf, 10.0);
   pdfRectangle(pdf, 50, 50, 50, 50, fmFillStroke);
   pdfApplyPattern(pdf, pat3, cmStroke, 0);
   pdfRectangle(pdf, 120, 50, 50, 50, fmFillStroke);

   pdfEndPage(pdf);
   pdfCloseFile(pdf);
   pdfDeletePDF(pdf); // Do not forget to delete the PDF instance
}
```

Output:

Opaque background (border)   **Transparent background (border)**
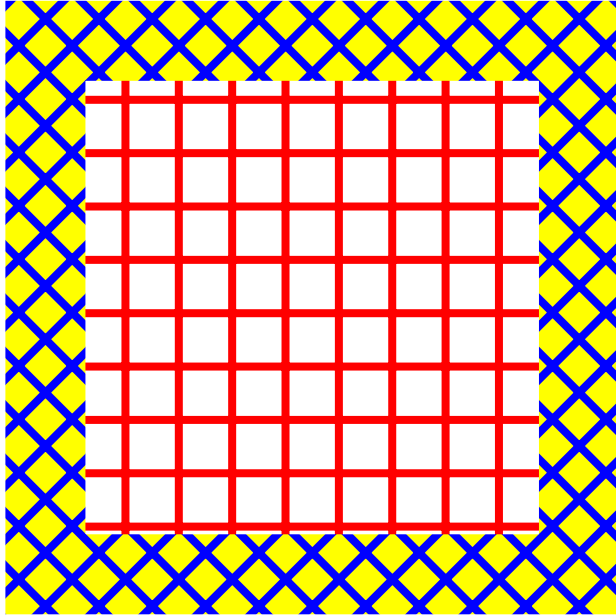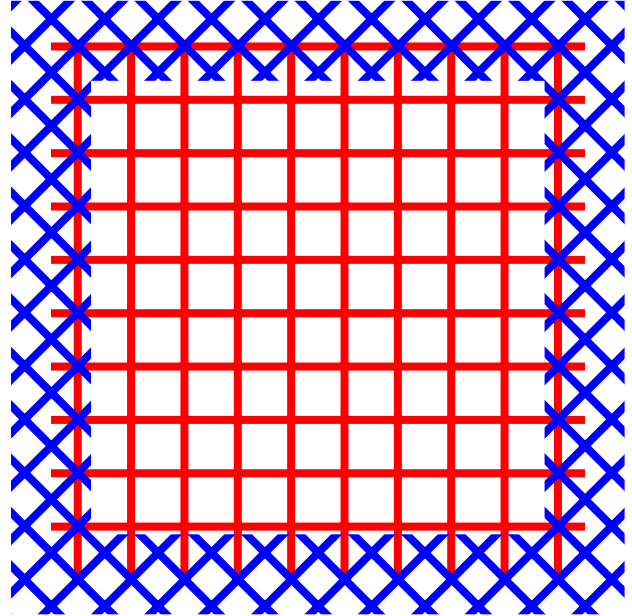
CreateStructureTree

Syntax:

```
LBOOL pdfCreateStructureTree(
  const PPDF* IPDF) // Instance pointer
```

The function creates a global structure tree that is required to create Tagged PDF files. Tagged PDF files require a root node of type Art, Div, Document, Part, or Sect. This function creates a root node of type Document if nothing else was already imported. Other types can be created with the function CreateStructureTreeEx().

Notice:

DynaPDF is able to extend an existing structure tree that was imported from an external PDF file. However, when editing external PDF files the following rules must be considered:

- The structure information of a PDF file will only be imported when the entire PDF file is imported with ImportPDFFile(). When importing single pages with ImportPage() or ImportPageEx() of a PDF file that contains structure information then tagging will be disabled because DynaPDF is not able to import the structure information of a PDF file on a per page basis.

- Import **first** the PDF file and call then CreateStructureTree(). The reverse order causes that the structure tree will not be imported and tagging will be disabled!

- Only one PDF file with structure information can be imported without invalidating the structure information.

- Keep in mind that the structure tree of a PDF file is a complex global structure that is difficult to edit. Because of this, probably millions of PDF files exist that contain damages in the structure tree, mostly due to editing actions in certain viewer applications. When opening an existing page with EditPage() the function tries to find the corresonding StructParents array of the page in the ParentTree of the document's Structure Tree. When this action fails then tagging will be disabled for this page. OpenTag() and CloseTag() do not produce further warnings in this case. The function GetIsTaggingEnabled() can be used to determine whether tagging is still enabled after an existing page was opened for editing.

When creating Tagged PDF files it is important to create the PDF file in the logical reading order.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is zero.

CreateStructureTreeEx

Syntax:

```
LBOOL pdfCreateStructureTreeEx(
    const PPDF* IPDF,      // Instance pointer
    TPDFBaseTag RootTag) // btArt, btDiv, btDocument, btPart, or btSect.
                          // Any other value will be replaced by btDocument.
```

The function creates the global structure tree as described at CreateStructureTree(). The only difference is that the type of the root node can be specified. See CreateStructureTree() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is zero.

CreateSubmitAction

Syntax:

```
SI32 pdfCreateSubmitAction(
    const PPDF* IPDF,    // Instance pointer
    TSubmitFlags Flags, // Additional flags see below
    const char* URL)     // URL of the web server
typedef SI32 TSubmitFlags;
#define sfNone           0x00000000 // No flags -> FDF format
#define sfExlude         0x00000001 // Excl. the fields of the action
#define sfInclNoValFields 0x00000002 // Include empty fields
#define sfHTML           0x00000004 // HTML format
#define sfGetMethod      0x00000008 // Use Get instead of Post
#define sfSubmCoords     0x00000010 // Mouse coordinates
#define sfXML            0x00000024 // XFDF format PDF 1.4
#define sfInclAppSaves   0x00000040 // Include append saves PDF 1.4
#define sfInclAnnots     0x00000080 // Include annotations PDF 1.4
#define sfPDF            0x00000100 // PDF 1.4
#define sfCanonicalFormat 0x00000200 // Canonical date format PDF 1.4
#define sfExlNonUserAnnots 0x00000400 // PDF 1.4
#define sfExlFKey        0x00000800 // PDF 1.4
#define sfEmbedForm      0x00002000 // PDF 1.5 Embed the form into FDF
```

A submit form action submits the field values of an interactive form to a web server. It is also possible to submit the entire PDF file; however, this feature requires the full version of Adobe's Acrobat (version 5 or higher).

Instead of simply submitting all field values to a web server it also possible to submit the values of specific fields only. The fields must be added to the action with the function AddFieldToFormAction(). The fields of the action can be included or excluded depending whether the flag sfExclude was set or not.

Remarks:

Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

Flag	Description
sfNone	Use default settings. Field values are transmitted in Forms Data format (FDF).
sfExlude	Exclude the fields in the submit form action if any. These fields must be added to the form action with AddFieldToFormAction().
sfInclNoValFields	If set, all fields designated by the fields array of the form action are submitted, regardless whether they have a value, only the field name is transmitted. If no fields are added to form action this flag is applied to all fields included in the interactive form.
sfHTML	If set, the field names and values are submitted in HTML format. If clear, they are submitted in Forms Data Format (FDF). This flag is meaningful only when the flag sfXML is clear; if set, this flag is ignored.
sfGetMethod	Use HTTP GET request instead of POST.
sfSubmCoords	If set, the coordinates of the mouse click that caused the submitform action are transmitted as part of the form data. The coordinate values are relative to the upper-left corner of the field 's bounding box. They are represented in the data in the format name.x = xval & name.y = yval where name is the field name. This flag is meaningful only when the sfHTML flag is set; if sfHTML is clear, this flag must also be clear.
sfXML	*(PDF 1.4)* If set, field names and values are submitted in XFDF format.
sfInclAppSaves	*(PDF 1.4)* Meaningful only when the form is being submitted in Forms Data Format (that is, when both the sfXML and sfHTML flags are clear). If set, the submitted FDF file includes the contents of all incremental updates to the underlying PDF document, as contained in the Differences entry in the FDF dictionary; if clear, the incremental updates are not included.

Flag	Description
sfInclAnnots	*(PDF 1.4)* Meaningful only when the form is being submitted in Forms Data Format (that is, when both the sfXML and sfHTML flags are clear). If set, the submitted FDF file includes all annotations in the underlying PDF document; if clear, the annotations are not included.
sfPDF	*(PDF 1.4)* If set, the document is submitted in PDF format, using the MIME content type application/pdf (described in Internet RFC 2045, Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies). If this flag is set, all other flags are ignored except sfGetMethod. This flag is not supported by Adobe's Acrobat Reader.
sfCanonicalFormat	*(PDF 1.4)* If set, any submitted field values representing dates are converted to a standard format (see next page). The interpretation of a form field as a date is not specified explicitly in the field itself, but only in the JavaScript code that processes it.
sfExlNonUserAnnots	*(PDF 1.4)* Meaningful only when the form is being submitted in Forms Data Format (that is, when both the sfXML and sfHTML flags are clear) and the sfInclAnnots flag is set. If set, will include only those annotations whose title matches the name of the current user, as determined by the remote server to which the form is being submitted. The title, which specifies the text label to be displayed in the title bar of the annotation's pop-up window, is assumed to represent the name of the user authoring the annotation. This allows multiple users to collaborate in annotating a single remote PDF document without affecting one another's annotations.
sfExlFKey	*(PDF 1.4)* Meaningful only when the form is being submitted in Forms Data Format (that is, when both the sfXML and sfHTML flags are clear). If set, the submitted FDF will exclude the "F" entry (the PDF document file that this FDF file was exported from).

Flag	Description
sfEmbedForm	*(PDF 1.5)* Meaningful only when the form is being submitted in Forms Data Format (that is, when both the sfXML and sfHTML flags are clear). If set, the F entry of the submitted FDF will be a file specification containing an embedded file stream representing the PDF file from which the FDF is being submitted. This flag is not supported by Adobe's Acrobat Reader.

The standard date format

PDF defines a standard date format, which closely follows that of the international standard ASN.1 (Abstract Syntax Notation One), defined in ISO/IEC 8824.

A date string is defined as follows:

(D:YYYYMMDDHHmmSSOHH'mm')

where

YYYY is the year
MM is the month
DD is the day (01 –31)
HH is the hour (00 –23)
mm is the minute (00 –59)
SS is the second (00 –59)
O is the relationship of local time to Universal Time (UT), denoted by one of the characters +, -, or Z (see below)
HH followed by ' is the absolute value of the offset from UT in hours (00 –23)
mm followed by ' is the absolute value of the offset from UT in minutes (00 –59)

The apostrophe character (') after HH and mm is part of the syntax. All fields after the year are optional. The default values for MM and DD are both 01; all other numerical fields default to zero values. A plus sign (+) as the value of the O field signifies that local time is later than UT, a minus sign (-) that local time is earlier than UT, and the letter Z that local time is equal to UT. If no UT information is specified, the relationship of the specified time to UT is considered to be unknown. Whether or not the time zone is known, the rest of the date should be specified in local time.

For example, December 23, 2004, at 7:52 PM, U.S. Pacific Standard Time, is represented by the string:

    D:200412231952–08'00'

CreateTextField

Syntax:

```
SI32 pdfCreateTextField(
   const PPDF* IPDF, // Instance pointer
   const char* Name, // Name of the text field
   SI32 Parent,      // Parent group field if any or -1
   LBOOL Multiline,  // Enable multi line text output
   SI32 MaxLen,      // Maximum length of the text or 0
   double PosX,      // X-Coordinate of the text field
   double PosY,      // Y-Coordinate of the text field
   double Width,     // Width of the text field
   double Height)    // Height of the text field
```

This function creates a text field. A text field is a box or space in which the user can enter text from the keyboard. The text is may be restricted to a single line or may be allowed to span multiple lines, depending whether the parameter *Multiline* is true or false. The parameter *MaxLen* specifies if the length of the text should be restricted; if zero, the length is not restricted.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the text field. If the coordinate system is top-down it defines the upper left corner.

Like all form fields, the width and height is measured incl. the line width of the border. The size of normal vector graphics is measured without the line width; this must be taken into account when calculating the width or height of a form field.

The line width of the border is taken from the current graphics state (see SetLineWidth()), it should be either 0, 1, 2, or 3 units (no border, thin, medium, or thick). The border style can be changed with the functions SetBorderStyle() or SetFieldBorderStyle().

A text field requires a font. If no font is active the standard font Helvetica is used. Note that when the form must be compatible to Acrobat 4, the used fonts by the form fields must be restricted to the 14 standard fonts (see SetFont() for further information). Acrobat 4 does not support interactive forms that use other fonts; such a form requires Acrobat 5 or higher.

Form fields support natively the code pages 1252 and MacRoman. However, other code pages and Unicode can be used too as long as a few external cmaps are available. See SetFieldFont() for further information.

The alignment of the text can be changed with the function SetFieldTextAlign().

The font size of a text field is set to "auto" if the flag ffMultiline is not set and if either the font size of the active font is set to 1.0 unit or if no font is active when the field is created.

A text field can be formatted and the allowed input values can be restricted to specific data formats. See SetDateTimeFormat(), SetNumberFormat() for further information.

Return values:

If the function succeeds the return value is the field handle, a value greater or equal zero. If the function fails the return value is a negative error code.

Specific flags supported by text fields:

Flag	Description
ffMultiline	If set, the field may contain multiple lines of text; if clear, the field's text is restricted to a single line.
ffPassword	If set, the field is intended for entering a secure password that should not be echoed visibly to the screen. Characters typed from the keyboard should instead be echoed in some unreadable form, such as asterisks or bullet characters. To protect password confidentiality, the value of the text field is not stored in the PDF file if this flag is set.
ffFileSelect	*(PDF 1.4)* If set, the text entered in the field represents the pathname of a file whose contents are to be submitted as the value of the field.
ffDoNotSpellCheck	*(PDF 1.4)* If set, the text entered to the field will not be spell-checked.
ffDoNotScroll	*(PDF 1.4)* If set, the field will not scroll (horizontally for single-line fields, vertically for multi-line fields) to accommodate more text than will fit within its annotation rectangle. Once the field is full, no further text will be accepted.
ffComb	*(PDF 1.5)* Meaningful only if MaxLen is set and if the flags ffMultiline, ffPassword, and ffFileSelect flags are clear. If set, the field is automatically divided up into as many equally spaced positions, or combs, as the value of *MaxLen*, and the text is laid out into those combs.

CreateURIAction

Syntax:

```
SI32 pdfCreateURIAction(
    const PPDF* IPDF, // Instance pointer
    const char* URL)  // URL of a website
```

A uniform resource identifier (URI) is a string that identifies (resolves to) a resource on the internet - typically a file that is the destination of a hyperlink, although it can also resolve to a query or other entity. A URI action causes a URI to be resolved. The parameter URL must be 7-bit ASCII string.

Remarks:

Actions must be added to a PDF object with AddActionToObj().

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateViewport

Syntax:

```
SI32 pdfCreateViewport(
    const PPDF* IPDF, // Instance pointer
    const char* Name, // Optional name of the viewport
    double X1,        // Lower left corner
    double Y1,        // Lower left corner
    double X2,        // Upper right corner
    double Y2)        // Upper right corner
```

The function creates a viewport. A viewport is a rectangular region of a page that can be associated with a measure dictionary. A measure dictionary achieves a coordinate conversion so that the distance and area can be calculated in other units like meters, feets, or millimeters, for example.

This is especially useful for CAD drawings which are not defined in PDF units. A PDF unit represents 1/72 inch.

A page can contain an arbitrary number of viewports and every viewport can be assigned with its own measure dictionary, e.g. to calculate coordinates in different units on the same page.

Viewports should be created in drawing order. Since viewports might overlap a PDF viewer examines the array starting with the last one and itereates in reverse order. The first one whose bounding box contains the point or cursor coordiante is chosen.

See also CreateRectilinearMeasure().

Remarks:

This function is implemented in an Ansi and Unicode compatible variant.

Return values:

If the function succeeds the return value is the viewport handle, a value greater or equal zero. If the function fails the return value is a negative error code.

CreateXFAStream

Syntax:

```
SI32 pdfCreateXFAStream(
   const PPDF* IPDF,    // Instance pointer
   const char* Name,    // Stream name, e.g. config, xdp:xdp, and so on
   const void* Buffer,  // XFA buffer
   UI32 BufSize)        // Buffer size in bytes
```

The function creates an XFA stream and adds it to the global XFA resource array. The parameter *Name* must be the name of the XFA resource. *Buffer* must be a valid XFA stream buffer or NULL, if the buffer should be set later with SetXFAStream().

Remarks:

This function is implemented in an Ansi and Unicode compatible variant.

Return values:

If the function succeeds the return value is the stream handle, a value greater or equal zero. If the function fails the return value is a negative error code.

DecryptPDF

Syntax:

```
SI32 pdfDecryptPDF(
   const PPDF* IPDF,     // Instance pointer
   const char* FileName, // PDF file to be decrypted
   TPwdType PwdType,     // Password type used to decrypt the file
   const char* Password) // Password
```

This function decrypts a PDF file by using the supplied password and password type. If the file is not encrypted the parameter password is ignored. The file is recompressed during import; this reduces the file size in most cases.

By using specific import flags (see SetImportFlags()), it is also possible to remove unwanted objects from the PDF file, such as annotations, form fields, bookmarks and so on.

If the input file contains a compressed object structure, it will be converted back to a normal PDF file.

The input PDF file will be replaced with the new one if no error occurred during import. If an error occurred the file is left unchanged.

If document information entries are set before calling the function, existing entries in the PDF file will be replaced with the new values, see the example below.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Unicode version is supported by Windows NT systems only. The file path is converted back to Ansi if the Unicode version is used under Linux or UNIX.

Return values:

If the function succeeds the return value is 0. If the function fails the return value is a negative error code. The returned error code can be used to determine whether a wrong password was supplied to the function. This can be done with the macro PDF_WRONG_PWD(). This macro returns true if the password was wrong. In Delphi, the C macro is defined as a normal function.

Example (C++):

```cpp
#include "dynapdf.h"
using namespace DynaPDF;
// First we declare an error callback function.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}
int main(int argc, char* argv[])
{
   pdfSetOnErrorProc(NULL, PDFError);
   // The document info entries are changed by the function if set
   pdfSetDocInfo(diSubject, "Decrypt PDF files");
   pdfSetDocInfo(diCreator, "C++ example test project");
   pdfSetDocInfo(diTitle, "Changed title");

   pdfDecryptPDF("c:/test.pdf", ptOpen, NULL);
   return 0;
}
```

DeleteAcroForm

Syntax:

```
void pdfDeleteAcroForm(
  const PPDF* IPDF) // Instance pointer
```

The function deletes a maybe existing Interactive Form. If the document contains no interactive form the function does nothing.

DeleteActionFromObj

Syntax:

```
SI32 pdfDeleteActionFromObj(
    const PPDF* IPDF, // Instance pointer
    TObjType ObjType, // Object type
    UI32 ActHandle,   // Action handle
    UI32 ObjHandle)   // Object handle

typedef enum
{
    otAction,
    otAnnotation,
    otBookmark,
    otCatalog,
    otField,
    otPage,
    otPageLink
}TObjType;
```

This function deletes an action from a PDF object. If the object type is page, then use the page number as handle. If the handle of the action is not known, because the object was may be imported from an external PDF file, then use the function DeleteActionFromObjEx() instead.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteActionFromObjEx

Syntax:

```
SI32 pdfDeleteActionFromObjEx(
    const PPDF* IPDF, // Instance pointer
    TObjType ObjType, // Object type
    UI32 ObjHandle,   // Object handle
    UI32 ActIndex)    // Action index (see below)
```

This function can be used to delete an action from an object without using an action handle. The action handle is often not known because the object was may be imported from an external PDF file. In this case the actions of the object can be enumerated in a *while* statement and deleted by this function if necessary (see example below).

The maximum index is decremented each time an action was deleted.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (C):

```
// In this example, all actions are removed from the object. The action index can be zero because the
// next action gets the position of the deleted one.
SI32 actCount, objHandle;
...
actCount = pdfGetObjActionCount(pdf, otBookmark, objHandle);
```

```
while (actCount > 0)
{
   pdfDeleteActionFromObjEx(pdf, otBookmark, objHandle, 0);
   --actCount;
}
// Now, we want to delete all actions except JavaScript actions from the
// object.
SI32 i, actCount, actType;
...
actCount = pdfGetObjActionCount(pdf, otPage, 2); // Page 2
for (i = 0; i < actCount; i++)
{
   // Check whether an error occurred
    if ((actType = pdfGetActionTypeEx(pdf, otPage, 2, i)) < 0) break;
   if (actType != (SI32)atJavaScript)
   {
       pdfDeleteActionFromObjEx(pdf, otPage, objHandle, i);
       break;
   }
}
```

DeleteAltFontList

Syntax:

```
LBOOL pdfDeleteAltFontList(
   const PPDF* IPDF, // Instance pointer
   SI32 Handle)      // List handle or -1 to delete all lists
```

The function deletes an alternate font list that was created by CreateAltFontList(). If the parameter Handle was set to -1 all available lists will be deleted.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteAnnotation

Syntax:

```
SI32 pdfDeleteAnnotation(
   const PPDF* IPDF, // Instance pointer
   UI32 AHandle)     // Annotation handle
```

This function deletes an annotation. The parameter *Handle* must be a valid annotation handle.

Annotations are global objects. To delete specific annotations of a PDF file use the functions GetAnnotCount(), GetAnnotType() and DeleteAnnotation() (see example below).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (C++):

```
// This example deletes all web links contained in the document
using namespace DynaPDF;
...
```

```
SI32 aType, annotCount = pdfGetAnnotCount(pdf);
for (SI32 i = 0; i < annotCount; i++)
{
    if ((aType = pdfGetAnnotType(pdf, i)) < 0) break;
    if ((TAnnotType)aType == atWebLink)
        pdfDeleteAnnotation(pdf, i);
}
...
```

DeleteAnnotationFromPage

Syntax:

```
LBOOL pdfDeleteAnnotationFromPage(
    const PPDF* IPDF, // Instance pointer
    UI32 PageNum,     // Page number from which to delete the annotation
    UI32 Handle)      // Annotation handle
```

The function deletes an annotation from a page. Typical annotations which are shared among pages are Watermark and Stamp annotations. *PageNum* is the page number from which the annotation should be deleted. The first page is denoted by 1.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteAppEvents

Syntax:

```
SI32 pdfDeleteAppEvents(
    const PPDF* IPDF,  // Instance pointer
    LBOOL ApplyEvent,  // Apply the visibility state Event before deletion?
    TOCAppEvent Event) // Will be ignored if ApplyEvent is false
```

The function deletes all application events if any. The function can optionally apply the visibility state of an event before the application events will be deleted. The new visibility state will be stored in the PDF file. To apply the visibility state of a specific event without deletion call ApplyAppEvent() instead.

Return values:

If the function succeeds and if application events were deleted the return value is 1. If the no application events were defined the return value is 0. If the function fails the return value is a negative error code.

DeleteBookmark

Syntax:

```
SI32 pdfDeleteBookmark(
    const PPDF* IPDF, // Instance pointer
    UI32 ABmk)        // Bookmark handle
```

This function deletes a bookmark. The parameter *ABmk* must be a valid bookmark handle. Please note that this function invalidates all bookmark handles which are numerically greater than ABmk. DynaPDF supports several functions to find or enumerate bookmarks. It is safe to delete a bookmark within a search run. See the following functions for further information:

- GetBookmarkCount()
- GetBookmark()
- FindBookmark(), FindNextBookmark()

Return values:

If the function succeeds the return value is the remaining number of bookmarks. If the function fails the return value is a negative error code.

DeleteDPartNode

Syntax:

```
LBOOL pdfDeleteDPartNode(
    const PPDF* IPDF, // Instance pointer
    SI32 Handle)      // DPart node handle or -1
```

The deletes a dpart node or the entire dpart structure if *Handle* was set to -1. *DPart* stands for Document Parts, a feature of PDF/VT. If the dpart node contains children then all child nodes will be deleted too.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteEmbeddedFile

Syntax:

```
LBOOL pdfDeleteEmbeddedFile(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle)      // Embedded file handle or array index.
```

The function deletes an embedded file. The parameter *Handle* must be a valid embedded file handle. Such a handle is a simple array index. GetEmbeddedFileCount() returns the remaining number of embedded files. If all embedded files should be deleted then delete the files from top to bottom by decrementing the loop variable. This is faster because it is not required to reorganize the array in which the files are stored in this case. All handles above the deleted file index become invalid after the file was deleted.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteField

Syntax:

```
SI32 pdfDeleteField(
   const PPDF* IPDF, // Instance pointer
   UI32 AField)      // Field handle
```

This function deletes an interactive form field. The parameter *AField* must be a valid field handle.

If the field is a radio button or group field, the child fields used are also deleted. There is no need to delete each field separately.

A field is never physically deleted. All field handles are still valid after a field was deleted, but the "Used" flag of the field is set to false (see GetField() for further information). If a field contained an action that was not used by another object then the action is also deleted.

Fields can also be deleted by using its name instead of the handle. See DeleteFieldEx() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteFieldEx

Syntax:

```
SI32 pdfDeleteFieldEx(
   const PPDF* IPDF, // Instance pointer
   const char* Name) // Full qualified field name
```

This function deletes an interactive form field by using its full qualified name, that is the name of any parent group field separated by a period (.) plus the field name.

Remarks:

Check boxes of a radio button have no name. It is not possible to delete such a check box by using this function, use DeleteField() instead. However, the entire radio button can be deleted with this function.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteJavaScripts

Syntax:

```
void pdfDeleteJavaScripts(
  const PPDF* IPDF,             // Instance pointer
  LBOOL DelJavaScriptActions) // Delete Javascript Actions?
```

The function deletes all global Javascripts and optionally all Javascript Actions from the document.

DeleteNamedDest

Syntax:

```
PDF_EXP LBOOL pdfDeleteNamedDest(
   const PPDF* IPDF, // Instance pointer
   const UI16* Name) // Name of the named destination or NULL
```

The function deletes a named destination. The way how named destinations are defined in PDF was changed in PDF 1.2. In earlier versions named destinations were accessible through byte strings only. Since PDF 1.2 named destinations can also be defined as name tree which use either Ansi or Unicode strings.

To avoid issues with different string formats it is strongly recommended to use the Unicode version of this function. The Unicode version is able to find named destinations independent of the string format that was used in the PDF file.

If *Name* is set to NULL or to an empty string all named destinations will be deleted.

Remarks:

All non C/C++ interface include the Unicode version of the function only.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteNamedDestByIndex

Syntax:

```
LBOOL pdfDeleteNamedDestByIndex(
   const PPDF* IPDF, // Instance pointer
   SI32 Index)       // Array index of the named destination or -1
```

The function deletes a named destination. If *Index* is set to a negative value all named destinations will be deleted.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteOCGFromAppEvent

Syntax:

```
LBOOL pdfDeleteOCGFromAppEvent(
   const PPDF* IPDF,                  // Instance pointer
   UI32 Handle,                       // OCG Handle
   TOCAppEvent Events,                // Bitmask, one or more events
   TOCGUsageCategory Categories,      // Bitmask, if DelCategoryOnly is true
   LBOOL DelCategoryOnly);            // If true, delete only the category from
                                      // the events
```

The function deletes an OCG or layer from one or more application events, or it deletes only one or more categories from an application event in which the OCG was found (if *DelCategoryOnly* is true).

If *DelCategoryOnly* is true, and if no more categories are left in the application event, then the event will be deleted. See also AddOCGToAppEvent() and SetOCGContUsage() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteOCGFromDisplayTree

Syntax:

```
LBOOL pdfDeleteOCGFromDisplayTree(
   const PPDF* IPDF, // Instance pointer
   UI32 OCGHandle,   // OCG handle
   LBOOL Recursive)  // Delete all occurrences?
```

The function deletes a layer or OCG (Optional Content Group) from the display tree of the currently active layer configuration.

Notice:

If the PDF file was not imported with ImportPDFFile() the function ImportOCProperties() must be called in order to import the global Optional Content Properties.

It is also required to load a layer configuration with LoadLayerConfig() before the function can be called.

The display tree is a linked list that contains UI nodes. UI stands for User Interface. Every node in this tree can contain an OCG or an array of OCGs with an optional label.

This function can be used to delete an OCG from the display tree without traversing every node of it with GetOCUINode() beforehand. Therefore, the function is suitable for console applications but GUI based applications should use DeleteOCUINode() instead. DeleteOCUINode() can also be used to delete the entire display tree with just one function call.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteOCUINode

Syntax:

```
LBOOL pdfDeleteOCUINode(
    const PPDF* IPDF, // Instance pointer
    IOCN* Node)       // Node to delete or NULL to delete all UI nodes
```

The function deletes an *OCUINode* (Optional Content User Interface Node) from the display tree of the currently active layer configuration or the entire tree if *Node* is set to NULL.

Notice:

If the PDF file was not imported with ImportPDFFile() the function ImportOCProperties() must be called in order to import the global Optional Content Properties.

It is also required to load a layer configuration with LoadLayerConfig() before the function can be called.

Note that the deletion of the display tree does not delete any layer or OCG (Optional Content Group). All OCGs still exists and are fully functional. However, users are not able to change the visibility state of OCGs which are not included in the display tree of the corresponding layer configuration.

If the parameter *Node* is set then it must be a valid pointer of an *OCUINode* that was returned by GetOCUINode(). Note that the deletion of a node invalidates all pointers which were previously returned by GetOCUINode().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteOutputIntent

Syntax:

```
SI32 pdfDeleteOutputIntent(
    const PPDF* IPDF, // Instance pointer
    SI32 Index)       // Array index or -1 to delete all intents
```

The function can be used to delete a specific or all output intents. A PDF file can contain more than one output intent. Call GetOutputIntentCount() to determine the number of available output intents. If the parameter *Index* is set to -1 all output intents will be deleted.

Return values:

If the function succeeds the return value is the remaining number of output intents. If the function fails the return value is a negative error code.

DeletePage

Syntax:

```
SI32 pdfDeletePage(
    const PPDF* IPDF, // Instance pointer
    UI32 PageNum)     // Page number
```

This function deletes a page. The parameter *PageNum* must be the page number that should be deleted. The first page has the number 1, the second 2 and so on. A page is physically deleted by this function but no objects used by the page. All other pages and objects are still valid.

Return values:

If the function succeeds the return value is the remaining number of pages. If the function fails the return value is a negative error code.

DeletePageLabels

Syntax:

```
void pdfDeletePageLabels(
    const PPDF* IPDF) // Instance pointer
```

The function deletes all page labels contained in the current open document, if any.

DeletePDF

Syntax:

```
SI32 pdfDeletePDF(
    PPDF* IPDF) // Instance pointer
```

The function deletes a PDF instance that was previously created with the function NewPDF().

This function is automatically called in the wrapper classes for Visual Basic, Visual Basic .Net and Delphi.

DeleteRasterizer (Rendering Engine)

Syntax:

```
void rasDeleteRasterizer(
    IRAS** RasPtr) // Address of the rasterizer pointer
```

The function deletes a rasterizer object and sets the variable to NULL. The address of the variable must be passed to the function, e.g. rasDeleteRasterizer(&ras);

DeleteSeparationInfo

Syntax:

```
LBOOL pdfDeleteSeparationInfo(
   const PPDF* IPDF, // Instance pointer
   LBOOL AllPages)   // Delete the separation info of all pages?
```

The function deleted the separation of the current open page or of all pages if the parameter AllPages is set to true.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteTemplate

Syntax:

```
SI32 pdfDeleteTemplate(
   const PPDF* IPDF, // Instance pointer
   UI32 Handle)      // Template handle
```

This function deletes a template. The parameter *Handle* must be a valid template handle. A template is not physically deleted by this function, only the content stream is cleared. Templates can be references in other content stream so that it is quite complex to delete a template physically at runtime. However, this behaviour is maybe changed in a future version of DynaPDF.

Imported PDF pages are converted to templates by default. The handles of such templates are often not known, however, it is possibly to delete them by using the function DeleteTemplateEx().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DeleteTemplateEx

Syntax:

```
SI32 pdfDeleteTemplateEx(
   const PPDF* IPDF, // Instance pointer
   UI32 Index)       // Index of the template inside the page's array
```

This function deletes a template by using an index instead of a template handle. The parameter *Index* represents an index into the array of templates used by the current open page. The number of templates used by a page is returned by the function GetTemplCount().

It is not easy to identify a specific template, all templates used by a page can be deleted easily, but it is much more complex to delete a specific template if the page contains more than one.

However, if a template contains text, then the functions EditTemplate() and GetPageText() can be used to identify a template, but in most cases it is easier to delete a template and check the file with Acrobat or Reader to determine whether it was the right one.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (C++):

```cpp
// In this example, we have a document with 10 imported pages.
// The entire document was imported with the function ImportPDFFile()
// so that all interactive features are also imported. However, the
// third page contains invalid contents that should be removed. The
// imported pages were converted to templates during import (it is also
// possible to import pages without conversion). The template handles
// of the imported pages are not known so that we use the indexes instead.
...
pdfEditPage(3);
SI32 tmplCount = pdfGetTemplCount();
for (SI32 i = 0; i < tmplCount; i++)
   pdfDeleteTemplateEx(i);
pdfEndPage();
...
```

DeleteWatermark

Syntax:

```cpp
SI32 pdfDeleteWatermark(
   const PPDF* IPDF, // Instance pointer
   SI32 PageNum,     // Page number or -1 for all pages
   LBOOL InclAnnots) // Delete watermark annotations too?
```

The function can be used to delete watermarks from a single page or from the entire document if *PageNum* is set to -1. If *InclAnnots* is true, watermark annotations will be deleted too, if any.

A watermark in PDF terms is a regular template that has a ADBE_CompoundType property of type Watermark. Such templates can be created with the Watermark tool of Adobes Acrobat, or with DynaPDF, for example. See MarkTemplateAsWatermark() for further information.

Return values:

If the function succeeds the return value is the number of deleted watermarks. Note that deleted watermark annotations do not increment the return value. If the function fails, the return value is a negative error code.

DeleteXFAForm

Syntax:

```
void pdfDeleteXFAForm(
  const PPDF* IPDF) // Instance pointer
```

The function deletes a maybe existing XFA form (XML based form) if any. This can be useful if the PDF file contains a hybrid form because DynaPDF does not allow editing form fields as long as an XFA form is present.

DrawArc

Syntax:

```
SI32 pdfDrawArc(
  const PPDF* IPDF,   // Instance pointer
  double PosX,        // X-Coordinate of the midpoint of the arc
  double PosY,        // Y-Coordinate of the midpoint of the arc
  double Radius,      // Radius
  double StartAngle,  // Start angle
  double EndAngle)    // End angle
```

This function draws an arc by using a start and end angle. A full circle is drawn if the start and end angles coincide. The path is not closed, stroked or filled so that it will be invisible as long it is not stroked, filled or both (see ClosePath() or StokePath() for further information).

The current point is connected with the start point and then updated to the end point of the arc (see example below).

The draw direction can be changed with the function SetDrawDirection(). The start and end angles are always measured counter clockwise independent of the drawing direction.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (Delphi):

```
implementation
uses dynapdf;
procedure TForm1.Button1Click(Sender: TObject);
var pdf: TPDF;
begin
    pdf := nil;
    try
        pdf := TPDF.Create;
        // Declaration of the error callback (see SetOnErrorProc())
        pdf.SetOnErrorProc(nil, @ErrProc);

        pdf.CreateNewPDF('c:\dout.pdf');

        pdf.SetDocInfo(diCreator, 'Delphi sample project');
        pdf.SetDocInfo(diSubject, 'How to draw an arc segment');
        pdf.SetDocInfo(diTitle, 'Vector graphics');
        pdf.Append;
            pdf.SetStrokeColor(clBlue);
            pdf.SetFillColor($00FFCFCF);
            pdf.SetDrawDirection(ddClockwise);
            pdf.MoveTo(250, 500);
            pdf.DrawArc(250, 500, 50, 30, 60);
            pdf.ClosePath(fmFillStroke);
            pdf.SetDrawDirection(ddCounterClockwise);
            pdf.SetStrokeColor($0000A600);
            pdf.SetFillColor($00DDFFDD);
            pdf.MoveTo(255, 505);
            pdf.DrawArc(250, 500, 50, 33, 57);
            pdf.ClosePath(fmFillStroke);
        pdf.EndPage;
        pdf.CloseFile;
    except
        on E: Exception do MessageDlg(E.Message, mtError, [mbOK], 0);
    end;
    if pdf <> nil then pdf.Free;
end;
```

Output:

DrawArcEx

Syntax:

```
SI32 pdfDrawArcEx(
   const PPDF* IPDF,  // Instance pointer
   double PosX,       // X-Coordinate of the midpoint of the arc
   double PosY,       // Y-Coordinate of the midpoint of the arc
   double Width,      // Width of the bounding rectangle
   double Height,     // Height of the bounding rectangle
   double StartAngle, // Start angle
   double EndAngle)   // End angle
```

This function draws an elliptical arc. A full ellipse is drawn if the start and end angles are the same. The path is not closed, stroked or filled so that it will be invisible as long it is not stroked, filled or both (see ClosePath() or StokePath() for further information).

The current point is connected with the start point and then updated to the end point of the ellipse.

The draw direction can be changed with the function SetDrawDirection(). The start and end angles are always measured counter clockwise independent of the drawing direction.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DrawChord

Syntax:

```
SI32 pdfDrawChord(
   const PPDF* IPDF,        // Instance pointer
   double PosX,             // X-Coordinate of the midpoint of the chord
   double PosY,             // Y-Coordinate of the midpoint of the chord
   double Width,            // Width of the bounding rectangle
   double Height,           // Height of the bounding rectangle
   double StartAngle,       // Start angle
   double EndAngle,         // End angle
   TPathFillMode FillMode)  // Fill mode
```

This function draws an elliptical chord (a region bounded by the intersection of an ellipse and a line segment, called a secant).

The draw direction can be changed with the function SetDrawDirection(). The start and end angles are always measured counter clockwise independent of the drawing direction.

A chord is a closed path that can be filled, stroked or both. It is also possible to draw a chord invisible to apply the filling rules nonzero winding number or even-odd. The filling rules are described under ClipPath(). The parameter *FillMode* is ignored if the chord is drawn inside a clipping path. The fill modes are described under ClosePath().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DrawCircle

Syntax:

```
SI32 pdfDrawCircle(
   const PPDF* IPDF,      // Instance pointer
   double PosX,           // X-Coordinate of the midpoint of the circle
   double PosY,           // X-Coordinate of the midpoint of the circle
   double Radius,         // Radius
   TPathFillMode FillMode) // Fill mode
```

This function draws a circle. The draw direction can be changed with the function SetDrawDirection().

A circle is a closed path that can be filled, stroked or both. It is also possible to draw a circle invisible to apply the filling rules nonzero winding number or even-odd. The filling rules are described at ClipPath(). The parameter *FillMode* is ignored if the circle is drawn inside a clipping path. The fill modes are described at ClosePath().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DrawNGon

Syntax:

```
LBOOL pdfDrawNGon(
   const PPDF* IPDF,      // Instance pointer
   double PosX,           // X-Coordinate of the midpoint of the ngon
   double PosY,           // Y-Coordinate of the midpoint of the ngon
   double Radius,         // Radius
   double Alpha,          // Rotation angle in degrees
   UI32 NumSides,         // Number of sides. Must be greater 2.
   TPathFillMode FillMode) // Fill mode
```

The function draws a n-gon. A n-gon is a polygon with "n" number of sides. The minimum number of sides is three.

A n-gon is a closed path that can be filled, stroked or both. It is also possible to draw a n-gon invisible to apply the filling rules nonzero winding number or even-odd. The filling rules are described at ClipPath(). The parameter *FillMode* is ignored if the circle is drawn inside a clipping path. The fill modes are described at ClosePath().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

DrawPie

Syntax:

```
SI32 PDF_CALL pdfDrawPie(
    const PPDF* IPDF,        // Instance pointer
    double PosX,             // X-Coordinate of the midpoint of the pie
    double PosY,             // Y-Coordinate of the midpoint of the pie
    double Width,            // Width of the bounding rectangle
    double Height,           // Height of the bounding rectangle
    double StartAngle,       // Start angle
    double EndAngle,         // End angle
    TPathFillMode FillMode)  // Fill mode
```

The function draws a pie-shaped wedge bounded by the intersection of an ellipse and two angles.

The draw direction can be changed with the function SetDrawDirection(). The start and end angles are always measured counter clockwise independent of the drawing direction.

A pie is a closed path that can be filled, stroked or both. It is also possible to draw a pie invisible to apply the filling rules nonzero winding number or even-odd. The filling rules are described under ClipPath(). The parameter *FillMode* is ignored if the pie is drawn inside a clipping path. The fill modes are described under ClosePath().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (Delphi):

```
implementation

uses DynaPDF;
// First we declare an error callback function
function ErrProc(const Data: Pointer; ErrCode: Integer; const ErrMessage:
PAnsiChar; ErrType: Integer): Integer; stdcall;
var s: String;
begin
    s := Format('%s'#13'Abort processing?', [ErrMessage]);
    if MessageDlg(s, mtError, [mbYes, mbNo], 0) = mrYes then
       Result := -1 // break processing
    else
       Result := 0; // try to continue
end;
```

```pascal
procedure TForm1.Button1Click(Sender: TObject);
var pdf: TPDF;
begin
   pdf := nil;
   try
      pdf := TPDF.Create;
      pdf.SetOnErrorProc(nil, @ErrProc);
      pdf.CreateNewPDF('c:\dout.pdf');
      pdf.SetDocInfo(diCreator, 'Delphi sample project');
      pdf.SetDocInfo(diSubject, 'Simple shapes');
      pdf.SetDocInfo(diTitle, 'Vector graphics');

      pdf.Append;
         pdf.SetPageCoords(pcTopDown);
         pdf.SetFont('Arial', fsNone, 12, true, cp1252);
         // default draw direction counter clockwise
         pdf.SetColors(clRed);
         pdf.WriteText(380, 143, 'DrawArcEx');
         pdf.DrawArcEx(360, 150, 200, 100, 330, 30);
         pdf.StrokePath;
         pdf.SetColors(clBlue);
         pdf.WriteText(380, 253, 'DrawPie');
         pdf.DrawPie(360, 260, 200, 100, 330, 30, fmStroke);
         pdf.SetColors(clGreen);
         pdf.WriteText(380, 363, 'DrawChord');
         pdf.DrawChord(360, 370, 200, 100, 330, 30, fmStroke);
         // new draw direction
         pdf.SetDrawDirection(ddClockWise);
         pdf.SetColors(clRed);
         pdf.WriteText(200, 143, 'DrawArcEx');
         pdf.DrawArcEx(250, 150, 200, 100, 330, 30);
         pdf.StrokePath;
         pdf.SetColors(clBlue);
         pdf.WriteText(200, 253, 'DrawPie');
         pdf.DrawPie(250, 260, 200, 100, 330, 30, fmStroke);
         pdf.SetColors(clGreen);
         pdf.WriteText(200, 363, 'DrawChord');
         pdf.DrawChord(250, 370, 200, 100, 330, 30, fmStroke);
      pdf.EndPage;
      pdf.CloseFile;
   except
      on E: Exception do MessageDlg(E.Message, mtError, [mbOK], 0);
   end;
   if pdf <> nil then pdf.Free;
end;
```

Output:

Draw direction counter clockwise   **Draw direction clockwise**

DrawArcEx   DrawArcEx

DrawPie   DrawPie

DrawChord   DrawChord
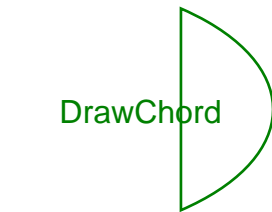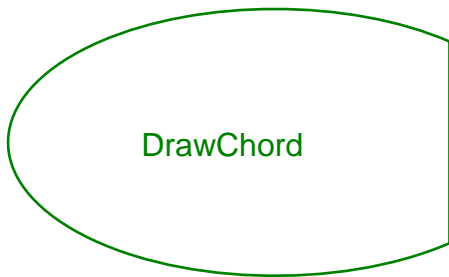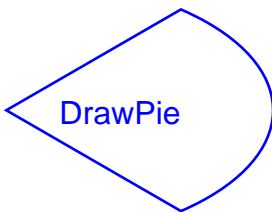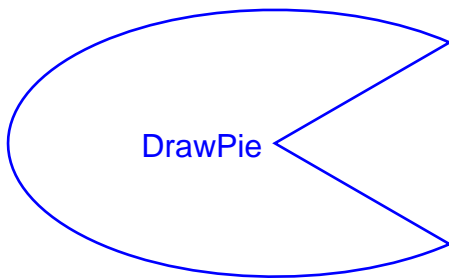
EditPage

Syntax:

```
SI32 pdfEditPage(
   const PPDF* IPDF, // Instance pointer
   SI32 PageNum)     // Page number
```

This function prepares a page for editing. If the page does exist (*PageNum* is greater than the number of pages in the document), new pages are appended until the number of pages is equal *PageNum*. An open page must be closed with EndPage() after the required changes are made.

When opening an existing page for editing DynaPDF does the following:

- When the page is opened the first time, DynaPDF parses the content stream to determine the graphics state at the end of the stream.
- When tagging is enabled and if the page contains already tagging information then it tries also to find the StructParents array of the page in the ParentTree of the global structure tree. If this action fails then tagging will be disabled for this page. The function GetIsTaggingEnabled() can be used to determine whether tagging is still enabled.
- Errors in the content stream will be repaired if possible.
- If the content stream contains non-repairable errors then editing is still possible but the result is of course undefined. Errors are written to the error log (see GetErrLogMessage()).
- When closing the page, the graphics state is stored in a compact format so that it is not required to parse the page again when it will be edited again.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

EditTemplate

Syntax:

```
SI32 pdfEditTemplate(
   const PPDF* IPDF, // Instance pointer
   UI32 Index)       // Array index of the template inside the page
```

This function prepares a template for editing. The parameter *Index* is the array index inside the page that contains the template. To get the number of templates used by a page call the function GetTemplCount().

The current graphics state is saved entirely before the template will be opened. This graphics state is restored when the template is closed with EndTemplate(). That means, the current font, line width, fill color and so on are all restored to its values before entering the template.

EndTemplate() requires an open page that must be opened with EditPage() beforehand. After the changes are made the template must be closed with EndTemplate().

Remarks:

EditTemplate() is mostly used to extract text strings from imported PDF files because imported pages are converted to templates by default. See GetPagetext() for an example application.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

EditTemplate2

Syntax:

```
SI32 pdfEditTemplate2(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle)      // Template handle
```

Templates can be edited multiple times in the same way as normal PDF pages. Therefore, the function prepares the template for editing. The current graphics state is saved entirely before the template will be opened and restored when the template will be closed with EndTemplate().The current font, line width, fill color and so on are all restored to its values before entering the template.

The function depends not on an open page.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Ellipse

Syntax:

```
SI32 pdfEllipse(
    const PPDF* IPDF,        // Instance pointer
    double PosX,             // X-Coordinate of the bounding rectangle
    double PosY,             // Y-Coordinate of the bounding rectangle
    double Width,            // Width of the bounding rectangle
    double Height,           // Height of the bounding rectangle
    TPathFillMode FillMode)  // Fill mode
```

The function draws an ellipse. If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the bounding rectangle. If the coordinate system is top-down it defines the upper left corner.

The draw direction can be changed with the function SetDrawDirection().

An ellipse is a closed path that can be filled, stroked or both. It is also possible to draw an ellipse invisible to apply the filling rules nonzero winding number or even-odd. The filling rules are described under ClipPath(). The parameter *FillMode* is ignored if the ellipse is drawn inside a clipping path. The fill modes are described under ClosePath().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

EnableImageCache (Rendering engine)

Syntax:

```
LBOOL pdfEnableImageCache(
    const PPDF* IPDF,     // Instance pointer
    UI32 MaxImageCount, // Maximum number of images to be cached
    UI32 Size)            // Maximum cache size in bytes
```

The function creates an image cache that is used when rendering PDF pages. The maximum number of images and the maximum cache size can be set independently of each other.

Caching can speed up rendering dramatically but caching can also slow down rendering if too many images are in the cache. Therefore, the maximum number of images must be restricted. The value should be between 256 and 1024 but not much higher.

The optimal cache size depends on the number of pages and whether the app is 32 or 64 bit. The cache size can be large, e.g. up to 1 GB on 32 bit or 2 GB on 64 bit systems, if a document contains only a few pages. Images are cached only if the system contains enough memory.

The size of the image cache should be around 30% of the entire cache size for rendered images. If the full cache size is 1 GB, for example, then around 300 MB should be used for the image cache. Keep all caches dynamic depending on the currently available memory.

The cache size is automatically reduced if the system has not sufficient memory.

The cache is always cleared when the PDF file in memory was closed or when FreePDF() was called.

In order to disable or delete the cache set *MaxImageCount* or *Size* to zero.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

EncryptPDF

Syntax:

```
SI32 pdfEncryptPDF(
    const PPDF* IPDF,        // Instance pointer
    const char* FileName,   // File to encrypt
    const char* OpenPwd,    // Open password (user password)
    const char* OwnerPwd,   // Owner password
    TKeyLen KeyLen,          // Key length for RC4 encryption
    TRestrictions Restrict) // Restrictions
```

This function encrypts a PDF file. The input file must be unencrypted or no open password must be set and the property SetUseExactPwd() must be set to false. The file is recompressed during import; this reduces the file size in most cases.

By using specific import flags (see SetImportFlags()), it is also possible to remove unwanted objects from the PDF file, such as annotations, form fields, bookmarks and so on.

If the input file contains a compressed object structure, it will be converted back to a normal PDF file.

The input PDF file will be replaced by the new one if no error occurred during import. If an error occurred the file is left unchanged.

It is not possible to change the file except removing specific objects by using specific import flags. However, the document info entries are changed to new values if they were set beforehand. See SetDocInfo() for further information.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Unicode version is supported by Windows NT systems only. The file path is converted back to Ansi if the Unicode version is used under Linux or UNIX.

Return values:

If the function succeeds the return value is 0. If the function fails the return value is a negative error code. The returned error code can be used to determine whether a password is required to open the file. Pass the return value to the macro PDF_WRONG_PWD(). If the return value of the macro is true, use the function ReEncryptPDF() instead.

EndContinueText (obsolete)

Syntax:

```
LBOOL pdfEndContinueText(
    const PPDF* IPDF) // Instance pointer
```

It is no longer required to finish a text block with this function. The function returns always true.

EndLayer

Syntax:

```
LBOOL pdfEndLayer(
    const PPDF* IPDF) // Instance pointer
```

The function closes a layer that was opened by BeginLayer().

If the function succeeds the return value is 1. If the function fails the return value is 0.

EndPage

Syntax:

```
SI32 pdfEndPage(
    const PPDF* IPDF) // Instance pointer
```

This function closes an open page that was opened by Append() or EditPage().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

EndPattern

Syntax:

```
SI32 pdfEndPattern(
    const PPDF* IPDF) // Instance pointer
```

This function closes an open pattern that was opened by BeginPattern() or CreateStdPattern().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

EndTemplate

Syntax:

```
SI32 pdfEndTemplate(
    const PPDF* IPDF) // Instance pointer
```

This function closes an open template that was opened by BeginTemplate(), EditTemplate() or EditTemplate2().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

EnumDocFonts

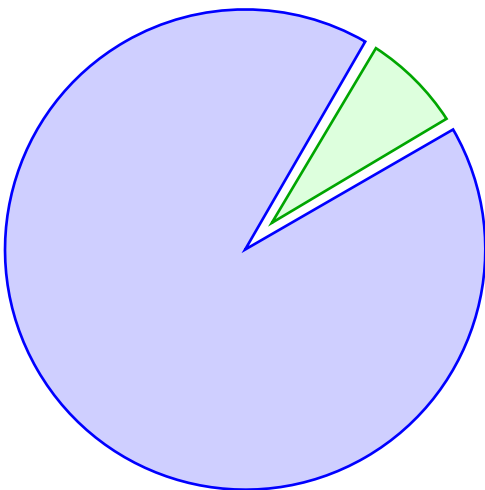Syntax:

```
SI32 pdfEnumDocFonts(
   const PPDF* IPDF,           // Instance pointer
   const void* Data,           // User defined pointer
   TEnumFontProc2* EnumProc)   // Callback function see below

typedef SI32 PDF_CALL TEnumFontProc2(
   const void* Data,        // User defined pointer
   const void* PDFFont,     // Pointer to font object for use with GetFont()
   TFontType Type,          // PDF Font type
   const char* BaseFont,    // Base font (PostScript or Family Name)
   const char* FontName,    // Font name taken from the font descriptor
   LBOOL Embedded,          // If true, the font is embedded
   LBOOL IsFormFont,        // It true, the font is reserved for form fields
   UI32 Flags)              // Various font properties

#define PDF_CALL __stdcall // Windows only, otherwise empty

typedef enum
{
   ftMMType1  = 0, // Multiple Master
   ftTrueType = 1, // TrueType font
   ftType0    = 2, // CID font (multi-byte font)
   ftType1    = 3, // Type1 font
   ftType3    = 4  // Type3 font (kostructed with PDF objects
}TFontType;
```

The function enumerates all fonts used in the current PDF document. If the parameter *EnumProc* is set to NULL the function returns the number of fonts used in the document. The callback function provides the most important properties of the font as well as a pointer to the font object that can be used to get further information about the font with GetFont(). The return value of the callback must be 0, every other value breaks processing.

The parameters *BaseFont* and *FontName* represent the original values stored in the font object or font descriptor. Both strings are provided because many PDF drivers set incorrect or different values to the font name and base font.

However, font names returned by the callback function begin often with a prefix like "AOMDMK+". Such a prefix is used to mark a font as a subset. In general, the parameter *BaseFont* contains either the family name or the postscript name of the font.

The real font name can be extracted with the function fntBuildFamilyNameAndStyle(). The function inspects the font names and other variables of the font object and font descriptor to determine what kind of name is provided and to extract the font style.

The parameter *Flags* provides further information about the font characteristics:

Bit	Value	Description
1	1	Fixed pitch.
2	2	Serif font.
3	4	Symbolic font.
4	8	Script. Glyphs resemble cursive handwriting.
6	32	Non-symbolic font. Either bit 3 or bit 6 is set at time.
7	64	Italic. Glyphs have dominant vertical strokes that are slanted.
19	262144	Force Bold. Indication for a bold font. (Ignore this flag for non Type1 fonts)

DynaPDF supports a special import flag to enable high performance access on the font objects of PDF files. This makes it possible to enumerate the fonts of a large number of PDF files as fast as possible.

The usage is as follows (C++):

```
...
pdfCreateNewPDF(pdf, NULL);
pdfSetImportFlags(pdf, ifEnumFonts); // Import font objects only
pdfOpenImportFile(pdf, "c:/test.pdf", ptOpen, NULL);
pdfImportPDFFile(pdf, 1, 1.0, 1.0);
// Now we can enumerate all PDF fonts
pdfEnumDocFonts(pdf, NULL, pdf_EnumDocFontProc);
// Only the font objects were imported. Delete the PDF file now.
pdfFreePDF(pdf);
...
```

The import flag ifEnumFonts causes that only font objects are imported. Anything else that can be discarded is not imported. After the fonts were enumerated the PDF file must be deleted. Do not try to write the file on disk, this would cause errors because certain required PDF objects are not available. To speed up processing make sure that the PDF instance is not always deleted and newly created when a PDF file is processed. This is especially important if a large number of PDF files must be processed. One PDF instance can be used on an arbitrary number of PDF files; there is no need to delete it every time a PDF file was processed.

Return values:

If the function succeeds the return value is number of fonts enumerated. If the parameter EnumProc is set to NULL the function returns the number of used fonts in the document.

EnumHostFonts

Syntax:

```
SI32 pdfEnumHostFonts(
    const PPDF* IPDF,          // Instance pointer
    const void* Data,          // User defined pointer
    TEnumFontProc* EnumProc) // Callback function see below

typedef SI32 PDF_CALL TEnumFontProc(
    const void* Data,          // User defined pointer
    const UI16* FamilyName,     // Family name of the font (Unicode)
    const char* PostScriptName, // PostScript name of the font
    SI32 Style);               // Font style (bold, italic, bold+italic)

#define PDF_CALL __stdcall     // Windows only, otherwise empty
```

This function enumerates all fonts found in the search directories by passing the font names to a callback function. If the parameter *EnumProc* is NULL the function returns the number of available font files which are found in the search directories. This is may be not the number of font files which are returned by the callback function!

Fewer fonts can be enumerated if the search directories contain invalid font files. However, it is also possible that more fonts are enumerated as font files were found!

This can happen if TrueType Collection font files are stored in the search directories. Such fonts contain more than one font in a font file, so that the number of fonts can be greater than the number of font files. If the font names should be stored in an array you must make sure that memory can be reallocated if the number of fonts exceeds the number of font files.

The font names passed to the callback function are null-terminated and sorted in ascending order by PostScript name. Note that the family name is in Unicode format. The user defined pointer *Data* is passed unchanged to the callback function. If this pointer is not required set it to NULL.

Enumeration stops immediately if the return value of the callback function is nonzero. This function depends not on an open PDF file.

Remarks:

The function does not display error messages or warnings. The internal exception handling of DynaPDF is not used by this function.

Return values:

If the function succeeds the return value is the number of available font files or zero if no font files were found. If the function fails the return value is a negative error code. At time of publication negative error codes are not returned.

Example (Delphi):

```
// In this example we have a combo box on the form that should be
// filled with the available fonts when the form is created.
// The font names passed to the callback function are already sorted in
// ascending order by PostScript name, so that we do not need to sort
// them again.

// Note that the font selection mode must be set to smPostScriptName
// if you want to use postscript names for font selection.
function EnumFontProc(const Data: Pointer; const FamilyName: PWideChar;
const PostScriptName: PAnsiChar; Style: TFStyle): Integer; stdcall;
var cb: TComboBox;
begin
   // Data holds the pointer of our combo box cbFonts.
   cb := Data;
   cb.AddItem(PostScriptName, nil);
   Result := 0; // Any other return value break enumeration!
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
   // First, we create an instance of the class TPDF. This is our main
   // instance used until the form will be destroyed. cbFonts is the
   // combo box on our form that should be used for font selection.
   PDF := TPDF.Create;
   PDF.EnumHostHonts(cbFonts, @EnumFontProc);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
   PDF.Free;
end;
```

EnumHostFontsEx

Syntax:

```
SI32 pdfEnumHostFontsEx(
   const PPDF* IPDF,          // Instance pointer
   const void* Data,          // User defined pointer
   TEnumFontProcEx* EnumProc) // Callback function see below

typedef SI32 PDF_CALL TEnumFontProcEx(
   const void* Data,         // User defined pointer
   const UI16* FamilyName,    // Family name
   const char* PostScriptName, // PostScript name
   SI32 Style,               // Font Style
   TFontBaseType BaseType,    // Base font type
   TEnumFontProcFlags Flags,  // See below
   const char* FilePath)      // File path to font file

#define PDF_CALL __stdcall    // Windows only, otherwise empty
```

```
typedef enum
{
    efpAnsiPath    = 0, // Code page 1252 on Windows, UTF-8 otherwise
    efpUnicodePath = 1, // FilePath is in Unicode format (UTF-16).
    efpEmbeddable  = 2, // The font has embedding rights.
    efpEditable    = 4  // The font has editing rights (important for form
                        // fields).
}TEnumFontProcFlags;
```

The function enumerates all fonts found in the search directories in the same way as EnumHostFonts(). However, the callback function provides further information about the font such as the font type, file path, and whether the font is embeddable or editable.

Form fields which can be edited, e.g. like text fields, require a font that is editable if the font will be embedded. If a font has no editable rights then the fields read only flag should be set since a PDF viewer application must replace the font if such a field would be edited. This can lead to unwanted side effects and should be avoided. To avoid unnecessary issues, fonts with no editable rights should not be embedded if used with form fields.

Note that the file path can be returned in Unicode or Ansi format. On Windows the string format can be different for every font. The default string format on Windows is Unicode but the function can also return Ansi strings, i.e. if further fonts were loaded with the Ansi version of AddFontSearchPath().

On non-Windows operating systems the function returns always UTF-8 Unicode strings.

Remarks:

The function does not produce error messages or warnings. The internal exception handling of DynaPDF is not used by this function.

Return values:

If the function succeeds the return value is the number of available font files or zero if no font files were found. If the function fails the return value is a negative error code. At time of publication negative error codes are not returned.

ExchangeBookmarks

Syntax:

```
LBOOL pdfExchangeBookmarks(
   const PPDF* IPDF, // Instance pointer
   SI32 Bmk1,        // Handle of first bookmark
   SI32 Bmk2)        // Handle of second bookmark
```

This function exchanges two bookmarks.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ExchangePages

Syntax:

```
LBOOL pdfExchangePages(
   const PPDF* IPDF, // Instance pointer
   UI32 First,       // First page number
   UI32 Second)      // Second page number
```

The function exchanges two pages. Page links, bookmarks, go-to actions, and named destinations are changed so that the destination page still refers to the correct page. Note that page numbering starts at 1. It is also possible to move a page to another position in the file (see MovePage() for further information).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ExtractText

Syntax:

```
LBOOL pdfExtractText(
   const PPDF* IPDF,            // Instance pointer
   UI32 PageNum,                // Page number
   TTextExtractionFlags Flags,  // See below
   struct TFltRect* Area,       // Optional -> Get the text of this area
   UI16** Text,                 // Out (required)
   UI32* TextLen)               // Out (required)
typedef enum
{
   tefDefault            = 0, // Create text lines in the original order.
   tefSortTextX          = 1, // Sort text records in x-direction.
   tefSortTextY          = 2, // Sort text records in y-direction.
   tefSortTextXY         = tefSortTextX | tefSortTextY,
   tefDeleteOverlappingText = 4  // See description
}TTextExtractionFlags;
```

The function extracts the text of the page *PageNum*. The first page is denoted by 1.

Text lines can be sorted in x- and y-direction. The flag *tefDeleteOverlappingText* causes that identical text records which are placed on the same position (with a tolerance of 2 units) will be deleted. The records must occur one after the other in order to detect them.

The optional parameter *Area* can be set to restrict the text extraction to that rectangle. The rectangle must be defined according to the current coordinate system. That means either in bottom up or top down coordinates, see SetPageCoords() for further information. Note also that the function considers the orientation of the page. The width and height of the rectangle must be exchanged if the orientation is 90, -90, 270, or -270 degrees.

The function SetSpaceWidthFactor() can be used to adjust the space width factor that is used to identify emulated space characters. This can be useful if too many or too few spaces occur in extracted text.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

FileAttachAnnot

Syntax:

```
SI32 pdfFileAttachAnnot(
   const PPDF* IPDF,       // Instance pointer
   double PosX,            // X-Coordinate for the annotation icon
   double PosY,            // Y-Coordinate for the annotation icon
   TFileAttachIcon Icon,   // Annotation icon
   const char* Author,     // Optional author
   const char* Desc,       // Optional description
   const char* AFile,      // File to be embedded (required)
   LBOOL Compress)         // If true, the embedded file will be compressed

typedef enum
{
   faiGraph,
   faiPaperClip,
   faiPushPin,
   faiTag
}TFileAttachIcon;
```

The function inserts a file attachment annotation on the current open page. If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left point of the annotation icon. If the coordinate system is top-down it defines the upper left point. The width and height of the icon is not changeable.

The embedded file is compressed if the parameter *Compress* is set to true. Otherwise it is left uncompressed. It is not always useful to compress embedded files especially if the file is already compressed, e.g. Zip files or already compressed image formats require no further compression. In the worst case the compressed files becomes larger. However, text files and most document formats should be compressed to reduce the file size.

File attachment annotations are supported since PDF 1.3 (Acrobat 4). File attachments are supported since PDF 1.4 (Acrobat 5). The difference between both attachment types is that a file attachment annotation is used as a comment that has usually a visible icon on a page, e.g. to provide additional information about a specific object.

File attachments are used to add arbitrary files to the PDF file, e.g. job options, the original document from which the file was created, or any other file (see also AttachFile()).

Note that Acrobat 7 or higher restricts access to executable files and compressed file formats such as zip, or rar. If such files must be embedded then change the file extension, e.g. to .dat or .bin. The user must then change the extension again when extracting the file.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Note that Unicode file paths are not supported under Linux and UNIX. The file path is converted back to Ansi on these operating systems before trying to open the file. This conversion can cause problems if the file name contains special characters which are not convertible to Ansi.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

FileAttachAnnotEx

Syntax:

```
SI32 pdfFileAttachAnnotEx(
   const PPDF* IPDF,        // Instance pointer
   double PosX,             // X-Coordinate for the annotation icon
   double PosY,             // Y-Coordinate for the annotation icon
   TFileAttachIcon Icon,    // Annotation icon
   const char* FileName,    // File name including extension (required)
   const char* Author,      // Optional author
   const char* Desc,        // Optional description
   const void* Buffer,      // File buffer (required)
   UI32 BufSize,            // Buffer size in bytes (required)
   LBOOL Compress)          // If true, the embedded file will be compressed
```

The function creates a file attachment annotation exactly in the same way as FileAttachAnnot() but accepts a file buffer as input. See FileAttachAnnot() for further information. The parameter *FileName* is required. It should contain the file name including extension, e.g. "MyImage.jpg".

Remarks:

The function is available in an Ansi and Unicode version.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

FileLink

Syntax:

```
SI32 pdfFileLink(
   const PPDF* IPDF,       // Instance pointer
   double PosX,            // X-Coordinate of bounding rectangle
   double PosY,            // Y-Coordinate of bounding rectangle
   double Width,           // Width of bounding rectangle
   double Height,          // Height of bounding rectangle
   const char* AFilePath)  // File path
```

The function creates a file link annotation. A file link annotation opens a file by using the application that is connected with the file extension (MIME type). Viewer applications maybe display a warning before the file is opened.

The file path can be defined as relative or absolute path. However, the usage of an absolute makes only sense if it can be guaranteed that the path exist on an arbitrary system.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the bounding rectangle. If the coordinate system is top-down it defines the upper left corner.

The border of the link annotation is drawn by using the current line width, stroke color and line dash pattern. If the link should appear without a border set the line width to zero beforehand.

When clicking on a link annotation the rectangle is highlighted, that is a simple visual effect. Several highlight modes are supported, see SetLinkHighlightMode() for further information.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. On Windows operating systems the path can be defined as Ansi or Unicode string depending on whether the Ansi or Unicode version of the function was called.

Like all file paths, the path of the Ansi version must be an UTF-8 encoded Unicode string on non-Windows operating systems.

The function does not check whether the path is valid.

Return values:

If the function succeeds the return value is a annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

FindBookmark

Syntax:

```
SI32 pdfFindBookmark(
    const PPDF* IPDF,   // Instance pointer
    SI32 DestPage,      // Destination page used by the bookmark or -1
    const char* Title)  // The title of the bookmark or a part of it
```

This function searches for a bookmark in the document outline tree. If the parameter *DestPage* is greater zero the function returns the handle of the first bookmark that uses this page number as destination page.

If *DestPage* is smaller 1 the function returns the handle of the first bookmark that contains the substring *Title* in the bookmark title. The substring *Title* is compared non-case-sensitive.

The parameter *Title* can be NULL, the function searches for bookmarks with no title in this case. To search for the next bookmark with the same search parameters call FindNextBookmark().

It is safe to delete a found bookmark with DeleteBookmark() and continue the search run with FindNextBookmark().

Remarks:

This function is implemented in an Ansi and Unicode compatible variant. Bookmarks can be stored in Unicode or Ansi string format. The function searches always by using both encodings, so that a bookmark can also be found if the Ansi variant of FindBookmark() was used, but the bookmark was stored in Unicode or vice versa.

Return values:

If a bookmark can be found the return value is the bookmark handle, a value greater or equal zero. If no bookmark can be found the return value is **-1**. If an error occurred, the return value is a negative error code.

FindEmbeddedFile

Syntax:

```
SI32 pdfFindEmbeddedFile(
    const PPDF* IPDF, // Instance pointer
    const char* Name) // Name of the embedded file
```

The function searches for the embedded file *Name* and returns the handle of it when it can be found. The file can then be extracted with GetEmbeddedFile().

Return values:

If the function succeeds the return value is an embedded file handle, a value greater or equal zero. If the file cannot be found the return value is -1.

FindField

Syntax:

```
SI32 pdfFindField(
   const PPDF* IPDF, // Instance pointer
   const char* Name) // Fully qualified field name
```

This function searches for an interactive form field by using the fully qualified field name. The fully qualified field name is constructed from the partial field name of the field and all of its ancestors. For a field with no parent group field, the partial and fully qualified names are the same. For a field that is the child of another field, the fully qualified name is formed by appending the child field's partial name to the parent's fully qualified name, separated by a period, e.g. "Company.Employee.Name".

Remarks:

Field names are case-sensitive; the name must be specified exactly. This function is available in an Ansi and Unicode compatible version. Both versions do not depend on the string format in which the field name is defined, e.g. Unicode or Ansi string.

Return values:

If the function succeeds the return value is the field handle, a value greater or equal zero. If the function fails the return value is a negative error code, a value smaller -1, or -1 if the field cannot be found.

FindLinkAnnot

Syntax:

```
SI32 pdfFindLinkAnnot(
   const PPDF* IPDF, // Instance pointer
   const char* URL)  // URL or file path
```

This function searches for a file link or web link annotation. The parameter *URL* must be the URL of a web link annotation or the file path of a file link annotation, exactly defined in a case sensitive manner.

Return values:

If the annotation can be found the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

FindNextBookmark

Syntax:

```
SI32 pdfFindNextBookmark(
    const PPDF* IPDF) // Instance pointer
```

The function searches for the next bookmark with the same search parameters which were used by a previous call of FindBookmark(). FindBookmark() must be called beforehand.

It is safe to delete a found bookmark with DeleteBookmark() and to continue the search run.

Return values:

If a bookmark can be found the return value is the bookmark handle, a value greater or equal zero. If no bookmark can be found the return value is **-1**. This function cannot fail; other return values are impossible.

FinishSignature

Syntax:

```
LBOOL pdfFinishSignature(
    const PPDF* IPDF,       // Instance pointer
    const void* PKCS7Obj,   // PKCS#7 object buffer
    UI32 Length)            // Length of the PKCS#7 object in bytes
```

The function writes the PKCS#7 signature object to the PDF file and writes finally the finish PDF file to disk and frees all used resources if the file was not created in memory. If the file was created in memory GetBuffer() can now be called to obtain the finish PDF buffer.

CloseAndSignFileExt() must be called prior this function can be called.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

FlattenAnnotOrField

Syntax:

```
LBOOL pdfFlattenAnnotOrField(
    const PPDF* IPDF,              // Instance pointer
    UI32 Index,                    // Annotation index
    TAnnotFlattenFlags Flags)      // See FlattenAnnots()
```

The function draws an annotation or form field on the current open page. Form fields are annotations of type atWidget. The annotations of a page can be enumerated with GetPageAnnotEx() / GetPageAnnotCount().

The annotation or form field is deleted from the page's annotation array after it was flattened.

Remarks:

Annotations and form fields are stored in the same array but handled differently depending on the function that is used to enumerate them. GetPageFieldEx() / GetPageFieldCount() use a pseudo indexing scheme by skipping regular annotations between form fields (Widget Annotations).

In order to access a form field with GetPageField() or GetPageFieldEx(), the function must be informed that an annotation index is passed to the function. This can be achieved by combining the array index with the flag PDF_ANNOT_INDEX as follows (i represents the array index):

- C/C++:   `pdfGetPageFieldEx(pdf, i | PDF_ANNOT_INDEX, &f);`
- VB 6:   `Call pdf.GetPageFieldEx(i Or PDF_ANNOT_INDEX, f)`
- VB .Net: `pdf.GetPageFieldEx(i Or CPDF.PDF_ANNOT_INDEX, f)`
- C#:   `pdf.GetPageFieldEx(Convert.ToInt32(i | CPDF.PDF_ANNOT_INDEX), ref f);`

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

FlattenAnnots

Syntax:

```
SI32 pdfFlattenAnnots(
  const PPDF* IPDF,         // Instance pointer
  TAnnotFlattenFlags Flags) // See below

typedef UI32 TAnnotFlattenFlags;
#define affNone              0x00000000 // Printable annotations
#define affUseViewState      0x00000001 // Visible annotations
#define affMarkupAnnots      0x00000002 // Markup annotations only
#define affNonPDFA_1         0x00000004 // Flatten annotations which are unsupported in PDF/A 1.
#define affNonPDFA_2         0x00000008 // Flatten annotations which are unsupported in PDF/A 2/3.
#define affNonPDFA_4e        0x00000080 // Flatten annotations which are unsupported in PDF/A 4e
#define affFormFields        0x00000010 // If set, form fields will be flattened too.
#define affUseFieldViewState 0x00000020 // Meaningful only if affFormFields is set. If set, flatten
                                        // the view state of form fields. Use the print state
                                        // otherwise.
#define affSigFields         0x00000040 // Meaningful only if affFormFields is not set. If set,
                                        // signed signature fields will be flattened.
#define affKeepLinkAnnots    0x00001000 // If set, link annotations will be kept.
#define affKeepFileAttach    0x00002000 // If set, file attachment annotations will be kept.
#define affKeepTextAnnots    0x00004000 // If set, text annotations will be kept.
#define affPreserveZOrder    0x00400000 // If set, preserve the z-order under any curcumstances.
                                        // That means visible overlapping annotations or form fields
                                        // will be flattened, regardless of the type.

#define affKeepEmptySigFields 0x00008000 // If set, signature fields which are not already signed
                                         // will be kept.
```

The function draws annotations and optionally form fields on the corresponding pages and deletes the annotations when finish. The flatten flags can be combined. The flag *affMarkupAnnots* causes that non-markup annotations will be left intact (all kinds of 3D, link, and sound annotations, for example).

The function is able to create appearance streams for most annotation types but not all. Missing appearance streams of 3D, Movie, RichMedia, Redact, PrinterMark, Projection, and Screen annotations cannot be created on demand.

If form fields should be flattened too then it is usually best to flatten the print state (default) since push puttons, for example, are usually excluded from printing.

Return values:

If the function succeeds the return value is the number of annotations which are still in memory. If the function fails the return value is a negative error code.

FlattenForm

Syntax:

```
SI32 pdfFlattenForm(
   const PPDF* IPDF) // Instance pointer
```

The function draws all form fields on the corresponding pages and deletes the form fields, incl. corresponding JavaScripts and JavaScript actions. The resulting PDF pages look after flattening as if the form were printed.

Note that this function does not support XFA forms. If the form is a hybrid form, the PDF form fields will be flattened. A maybe existing XFA form will be deleted but not flattened.

Fields, which are invisible for printing, due to an absent print flag, or if the hidden flag was set, are excluded from flattening. These fields are deleted without drawing them on the page.

This function is especially useful if a large amount of Interactive Forms (which are already filled out by the user) should be prepared for printing, or if the forms should be archived without allowing further changes. Flattened forms require less disk space and can be printed faster.

Annotations and form fields can be flattened with FlattenAnnots(). This function preserves also the z-order of annotations and form fields. This is not possible when flatting form fields and annotations separetly.

Remarks:

Pure XFA Forms which are created by Adobe's Designer are not supported. Only hybrid forms can be flatted.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

FlushPageContent

Syntax:

```
SI32 pdfFlushPageContent(
    const PPDF* IPDF,         // Instance pointer
    struct TPDFStack* Stack)  // Operation stack
```

The function replaces the content stream of a page or template that was changed with the function ReplacePageText() or ReplacePageTextEx() beforehand.

The function must be called after all changes are made. See GetPageText() for an example application.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

FlushPages

Syntax:

```
LBOOL pdfFlushPages(
    const PPDF* IPDF,        // Instance pointer
    TFlushPageFlags Flags)   // see below

typedef enum
{
    fpfDefault      = 0, // Write anything to the file that is possible
    fpfImagesOnly   = 1, // If set, only images are written to the PDF
                         // file. All pages are still kept in memory and
                         // can be modified with EditPage(). Flushed images
                         // can still be referenced in other pages. The
                         // image handles remain valid.
    fpfExclLastPage = 2  // If set, the last page is not flushed
}TFlushPageFlags;
```

The function writes the pages in memory to the PDF file. The function can be called every time a new page was created or whenever the pages in memory are no longer required.

Calling the function on a memory based PDF file is not meaningful. The output file must already be open before the function can be called. If no output file was set in CreateNewPDF() then open the output file with OpenOutputFile() or OpenOutputFileEncrypted() beforehand. The latter one can be used to create an encrypted PDF file.

Flushed pages can no longer be accessed with EditPage() but it is of course possible to add further pages to the file.

By default, the function writes the content streams of all pages and referenced templates, patterns, and images to the output file. The content streams of these objects will be released but the objects remain in memory. So, all handles remain valid and it is still possible to use already flushed objects in subsequent pages. For example, a flushed image or template can still be inserted in other pages with PlaceImage() or PlaceTemplate() because the content streams or image buffers are not required for this action.

The flag fpfImagesOnly can be used to flush the images in memory only. This can be useful if further objects must be added to the pages in memory or if a large image was inserted.

It is allowed to call the function within an open page. In this case the function does not flush the content stream of the current open page but anything else depending on the used flags.

Calling a function like DeletePage(), DeleteField() or any other function that deletes a flushed object results in a damaged PDF file.

Remarks:

PDF/A 4 output requires a cleanup run when closing the file. Links to color space resources must might be changed in this run. This is only possible if parts of the document have not already been written to the output file. Therefore, this function cannot be used for PDF/A 4 files.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

FlushPagesEx

Syntax:

```
LBOOL pdfFlushPagesEx(
   const PPDF* IPDF,      // Instance pointer
   TFlushPageFlags Flags, // Various flags
   UI32 LastPage)         // Last page that should be flushed or -1
```

The function writes the pages in memory to the PDF file exactly like FlushPages(). The only difference is that the last page can be specified. The function calls FlushPages() if *LastPage* is smaller 1.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

FreeImageBuffer

Syntax:

```
void pdfFreeImageBuffer(
   const PPDF* IPDF) // Instance pointer
```

The function frees the buffer of a memory image that was created with the function CreateImage().

FreeImageObj

Syntax:

```
LBOOL pdfFreeImageObj(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle)      // Image handle
```

The function releases memory that was allocated by GetImageObj() to decompress the image.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

FreeImageObjEx

Syntax:

```
LBOOL pdfFreeImageObjEx(
    const PPDF* IPDF,      // Instance pointer
    const void* ImagePtr) // Image pointer
```

The function releases memory that was allocated by GetImageObjEx() to decompress the image.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

FreePDF

Syntax:

```
SI32 pdfFreePDF(
    const PPDF* IPDF) // Instance pointer
```

The function frees all used resources with exception of the font and external CMap cache. The font and external CMap caches will be only freed when unloading the DLL or deleting the current PDF instance to improve processing speed. However, the font cache can be also be freed manually with the function ClearHostFonts().

There is normally no need to free the used resources by DynaPDF manually except for memory PDF files. When creating PDF files in memory the internal resources are not freed automatically after CloseFile() or CloseFileEx() was called. After the PDF buffer was retrieved by GetBuffer() call FreePDF() to free all internal used resources.

FreePDF() can be safely called at any time so that a PDF file can be deleted if necessary.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0. This function can normally not fail; a return value of zero indicates that an unknown fatal error occurred.

FreeTextAnnot

Syntax:

```
SI32 pdfFreeTextAnnot(
   const PPDF* IPDF,     // Instance pointer
   double PosX,          // X-Coordinate of the bounding rectangle
   double PosY,          // Y-Coordinate of the bounding rectangle
   double Width,         // Width of the bounding rectangle
   double Height,        // Height of the bounding rectangle
   const char* Author,   // Author of the annotation (can be NULL)
   const char* AText,    // The visible text of the annotation
   TTextAlign Align)     // The text alignment see below

typedef enum
{
   taLeft,
   taCenter,
   taRight,
   taJustify
}TTextAlign;
```

This function creates a Free Text annotation. The text of a Free Text annotation appears directly on screen such as normal text of a page. However, Free Text annotations are used to add comments. Comments can be excluded from printing if necessary (see SetAnnotFlags() for further information) and the contents of a Free Text annotation can be edited in Adobe's Acrobat.

The text of the annotation is printed into the bounding box by applying a formatting algorithm so that the text appears left aligned, right aligned, centered, or justified. Justified text is not directly supported by Adobe's Acrobat. However, the annotation appears correctly on screen but the alignment is set to left aligned text if the text will be changed in Acrobat.

The border of the annotation is drawn by using the current line width, stroke color and line dash pattern. If the annotation should appear without a border set the line width to zero beforehand.

The border appears completely inside the bounding box of the annotation. This must be considered because normal vector graphics such as rectangles, ellipses and so on are measured without the line width. For instance, the real width of a rectangle is the width plus the line width.

A Free Text annotation requires a font. If no font is set beforehand Helvetica is used.

Text positioning issues:

Acrobat 7 or higher use a very unusual way to place text into a Free Text annotation. While the baseline of text is very well defined in practically all typographics systems, Acrobat uses a dynamic position depending on the height of the largest character in the first line. The baseline is placed at the annotation's height - border width - cap height as long as the first line contains no character with an accent (e.g. Ä, Ö, Ü, À, È, Ê, Ñ). Because the cap height is not large enough for accented characters Acrobat adjusts the position of the first line if necessary. So, the text position depends on the contents of the first text line.

DynaPDF places the text at the annotation's height - border width - font size. This is makes it possible to change the text of a Free Text annotation without changing its position. However, if the annotation will be edited with Adobe's Acrobat the text position will slightly change. If the text position is important then it is mostly better to use text fields instead (see CreateTextField()).

Remarks:

This function is implemented in an Ansi and Unicode compatible variant.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

FreeUniBuf

Syntax:

```
SI32 pdfFreeUniBuf(
    const PPDF* IPDF) // Instance pointer
```

The function frees the internal buffer used by Unicode conversion functions such as ConvToUnicode(), ToUTF16() and so on. The conversion is automatically freed by all conversion functions before a new buffer is allocated. However, the buffer can be freed manually to reduce memory usage.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0. This function can normally not fail; a return value of 0 indicates that an unknown fatal error occurred.

Get3DAnnotStream

Syntax:

```
LBOOL pdfGet3DAnnotStream(
    const PPDF* IPDF, // Instance pointer
    UI32 Annot,       // 3D annotation handle
    BYTE** Data,      // Pointer to PRC or U3D file buffer
    UI32* Size,       // Data size in bytes
    char** SubType)   // U3D or PRC
```

The function retrieves the data stream of a 3D annotation. At time of publication, PDF supports U3D and PRC files. Check the value of the string *SubType* to determine the file type. The default format is U3D. *SubType* can be NULL in this case. For PRC files, *SubType* is required to be present.

The function returns pointers of the original values. Do not free or change the values.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetActionCount

Syntax:

```
SI32 pdfGetActionCount(
    const PPDF* IPDF) // Instance pointer
```

The function returns the number of actions contained in a document. However, this value does not include actions (mostly GoTo actions) which are defined inline in bookmarks or link annotations, for example. Such actions are not stored in separate objects since this would slow down PDF processing significantly.

GetActionHandle

Syntax:

```
SI32 pdfGetActionHandle(
    const PPDF* IPDF, // Instance pointer
    TObjType ObjType, // Object type which contains uses the action
    UI32 ObjHandle,   // Object handle
    UI32 ActIndex)    // Array index -> GetObjActionCount()
```

The function returns the handle of an action. To determine the number of available actions of a specific object call GetObjActionCount(). Since the *Catalog* object contains no handle, the parameter *ObjHandle* will be ignored for this object type.

Return values:

If the function succeeds the return value is the action handle, a value greater or equal zero. If the function fails the return value is a negative error code.

GetActionType

Syntax:

```
SI32 pdfGetActionType(
    const PPDF* IPDF, // Instance pointer
    UI32 ActHandle)   // Action handle
typedef enum
{
    atGoTo,         // Go-to action
    atGoToR,        // Go-to remote action
    atHide,         // Hide action
    atImportData,   // Import data action
    atJavaScript,   // JavaScript action
    atLaunch,       // Launch action
    atMovie,        // Movie action
    atNamed,        // Named action
    atRendition,    // Rendition action
    atReset,        // Reset form action
    atSetOCGState,  // Set OCG (optional content group) action
    atSound,        // Sound action
    atSubmit,       // Submit form action
    atThread,       // Thread action
    atTransition,   // Transition action
    atURI           // URI action
```

```
    atGoTo3DView,    // Go-to 3D view action
    atGoToE,         // Go-to embedded file action
    atRichMediaExec  // Rich Media Execution, PDF 1.7 Extension Level 3
}TActionType;
```

The function returns the type of an annotation. The parameter *ActHandle* must be a valid action handle. To determine the used action types of a specific object use GetActionTypeEx() instead.

Return values:

If the function succeeds the return value is the type of the annotation (make a type cast to TActionType to determine the action type). If the function fails the return value is a negative error code.

GetActionTypeEx

Syntax:

```
SI32 pdfGetActionTypeEx(
    const PPDF* IPDF, // Instance pointer
    TObjType ObjType, // Object type
    UI32 ObjHandle,   // Object handle
    UI32 ActIndex)    // Action index
typedef enum
{
    otAction,
    otAnnotation,
    otBookmark,
    otCatalog,
    otField,
    otPage,
    otPageLink
}TObjType;
```

The function returns the action type of an action used by a specific PDF object. The parameter *ObjType* specifies the type of object that identifies the parameter *ObjHandle*. If the object type is a page use the page number as handle.

The parameter *ActIndex* is an index into the objects action array. Use the function GetObjActionCount() to get the number of actions used by the object. This number can be used in a *for* statement to loop over the object actions. See DeleteActionFormObj() for an example application.

Return values:

If the function succeeds the return value is the type of the annotation (make a type cast to TActionType to determine the action type). If the function fails the return value is a negative error code.

GetActiveFont

Syntax:

```
SI32 pdfGetActiveFont(
    const PPDF* IPDF) // Instance pointer
```

The function returns the handle of the active font or **-1** if no font is set.

GetAllocBy

Syntax:

```
SI32 pdfGetAllocBy(
    const PPDF* IPDF) // Instance pointer
```

The function returns the pre-allocated buffer size of page content streams in bytes. Memory allocation is slow, especially re-allocation of memory by using realloc(). Because of this all content streams are buffered by DynaPDF to avoid too much memory allocation calls. The property AllocBy defines the size of memory that is allocated each time when more memory must be allocated.

The processing speed depends heavily on memory allocation. The default size of pre-allocated memory for content streams is 16 KB. It can be set to a larger or smaller value if necessary to improve processing speed.

However, if too much memory is allocated at runtime, processing speed will be slower and an out of memory exception can occur. If the size is too small, processing speed is slower too because of the many memory allocation calls. In most cases, it is not required to change the property *AllocBy*.

GetAnnot (obsolete)

Syntax:

```
SI32 pdfGetAnnot(
    const PPDF* IPDF)                    // Instance pointer
    UI32 Handle,                         // Annotation handle
    struct TPDFAnnotation ADDR Annot)    // see below
struct TPDFAnnotation
{
    TAnnotType      Type;           // Annotation type
    LBOOL           Deleted;        // Is this annotation marked as deleted?
    struct TPDFRect BBox;           // Bounding box in bottom-up coordinates
    double          BorderWidth;    // Border width -> see comment below
    UI32            BorderColor;    // Border color -> see comment below
    TBorderStyle    BorderStyle;    // Border style
    UI32            BackColor;      // Background color -> see comment below
    UI32            Handle;         // Annotation handle
    char*           AuthorA;        // Annotation's author, Ansi format
    UI16*           AuthorW;        // Annotation's author, Unicode format
    char*           ContentA;       // Annotation's contents, Ansi format
    UI16*           ContentW;       // Annotation's contents, Unicode format
    char*           NameA;          // Annotation's name, Ansi format
    UI16*           NameW;          // Annotation's name, Unicode format
    char*           SubjectA;       // Annotation's subject, Ansi format
    UI16*           SubjectW;       // Annotation's subject, Unicode format
    UI32            PageNum;        // Page on which the annotation appears
```

```
    THighlightMode  HighlightMode; // Visual effect
};
```

This function retrieves the most important properties of an annotation. This function is marked as obsolete and should no longer be used. Please use GetAnnotEx() if possible.

All strings are pointers to the original string buffers or NULL if not set. The application must not change or free these values. Only one string can be set at time, either the Ansi string or Unicode string but not both. String values are always null-terminated.

Return values:

If the function succeeds the return value is 1 and the parameter *Annot* is filled with values. If the function fails the return value is 0 and the parameter *Annot* is left unchanged.

GetAnnotBBox

Syntax:

```
SI32 pdfGetAnnotBBox(
    const PPDF* IPDF)            // Instance pointer
    UI32 Handle,                 // Annotation handle
    struct TPDFRect ADDR BBox)   // see below

struct TPDFRect
{
    double Left;    // Lower x-coordinate
    double Bottom;  // Lower y-coordinate
    double Right;   // Upper x-coordinate
    double Top;     // Upper y-coordinate
};
```

This function retrieves the bounding box of an annotation measured in bottom-up coordinates. A bounding box is defined as rectangle giving the coordinates of a pair of diagonally opposite corners.

Remarks:

It is possible to change the bounding box of an annotation with the function ChangeAnnotPos().

Return values:

If the function succeeds the return value is 1 and the parameter *BBox* is filled with values. If the function fails the return value is 0 and the parameter *BBox* is left unchanged.

GetAnnotCount

Syntax:

```
SI32 pdfGetAnnotCount(
    const PPDF* IPDF) // Instance pointer
```

The function returns the number of annotations currently used in the document. Note: This value includes also annotations which were deleted by DeleteAnnotation() because annotations are not physically deleted at runtime. This value can be used to loop over all annotations. Annotation handles are simple array indexes.

GetAnnotEx

Syntax:

```
LBOOL pdfGetAnnotEx(
  const PPDF* IPDF,                        // Instance pointer
  UI32 Handle,                             // Annotation handle
  struct TPDFAnnotationEx ADDR Annot)      // Structure to be filled

struct TPDFAnnotationEx
{
    TAnnotType      Type;
    LBOOL           Deleted;        // Marked as deleted?
    struct TPDFRect BBox;           // Bounding box in bottom-up coordinates
    float           BorderWidth;
    UI32            BorderColor;
    TBorderStyle    BorderStyle;
    UI32            BackColor;
    UI32            Handle;
    char*           AuthorA;
    UI16*           AuthorW;
    char*           ContentA;
    UI16*           ContentW;
    char*           NameA;
    UI16*           NameW;
    char*           SubjectA;
    UI16*           SubjectW;
    UI32            PageNum;
    THighlightMode  HighlightMode;
    // Page link annotations only
    SI32            DestPage;
    struct TPDFRect DestPos;
    TDestType       DestType;
    char*           DestFile; // File link or web link annotations
    // The Icon type depends on the annotation type. If the annotation type
    // is atText the Icon is of type TAnnotIcon. If the annotation type is
    // atFileAttach then it is of type TFileAttachIcon. If the annotation
    // type is atStamp the Icon is the stamp type (TRubberStamp).
    // For any other annotation type this value is not set (-1).
    SI32            Icon;
    char*           StampName;    // Set only, if Icon == rsUserDefined
    UI32            AnnotFlags;    // See TAnnotFlags for available flags
    char*           CreateDate;    // Optional Creation Date
    char*           ModDate;       // Optional Modification Date
    // Grouped is meaningful only if Parent != -1 and Type != atPopUp. If
    // true, the annotation is part of an annotation group. Properties like
    // Content, CreateDate, ModDate, BackColor, Subject, and Open must be
    // taken from the parent annotation.
    LBOOL           Grouped;
    LBOOL           Open;   // If the annotation has a PopUp annotation.

    // The Parent annotation handle is set if the annotation is a PopUp
    // Annotation or if this annotation represents a state of a base
    // annotation. In this case, the annotation type is always atText and
    // only the following members should be considered:
    //    State       -> The current state
    //    StateModel  -> Marked, Review, and so on
    //    CreateDate  -> Creation Date
    //    ModDate     -> Modification Date
    //    Author      -> The user who has set the state
    //    Content     -> Not displayed in Adobe's Acrobat...
    //    Subject     -> Not displayed in Adobe's Acrobat...
    // The PopUp annotation of a text annotation which represent an
    // Annotation State must be ignored.
    SI32            Parent;
    SI32            PopUp;       // Handle of the PopUp annotation if any
```

```
    char*           State;       // The state of the annotation
    char*           StateModel; // State model (Marked, Review, ...)
    // FileAttach annotations only. This is a handle of an embedded file.
    // The embedded file can be accessed with GetEmbeddedFile().
    SI32            EmbeddedFile;
    char*           Subtype;     // Set only, if Type == atUnknown
    LBOOL           MarkupAnnot; // If true, the annotation is a markup
                                 // annotation. Markup annotations can be
                                 // flattened separately (FlattenAnnots()).
    float           Opacity;     // Opacity = 1.0 = Opaque,
                                 // Opacity < 1.0 = Transparent, Markup
                                 // annotations only

    float*          QuadPoints;      // Highlight, Link, and Redact annotations only
    UI32            QuadPointsCount; // Number of values in the array. Since a quadpoint requires
                                     // always four coordinate pairs, the number of QuadPoints is
                                     // QuadPointsCount divided by 8.

    float*          DashPattern;      // If BorderStyle == bsDashed
    UI32            DashPatternCount;// Number of values in the array

    char*           Intent;           // Markup annotations only. The intent allows to distinguish
                                      // between different uses of an annotation. For example, line
                                      // annotations have two intents: LineArrow and LineDimension.

    TLineEndStyle   LE1;              // Style of the start point -> Line and PolyLine annotations
only
    TLineEndStyle   LE2;              // Style of the end point -> Line and PolyLine annotations only

    float*          Vertices;         // Line, PolyLine, and Polygon annotations only
    UI32            VerticesCount;    // Number of values in the array. This is the raw number of
                                      // floating point values. Since a vertice requires always two
                                      // coordinate pairs, the number of vertices or points is
                                      // VerticeCount divided by 2.

    // Line annotations only. These properties should only be considered if the member Intent is set
    // to the string "LineDimension".
    LBOOL           Caption;          // If true, the annotation string Content is used as caption.
                                      // The string is shown in a PopUp annotation otherwise.
    float           CaptionOffsetX;   // Horizontal offset of the caption from its normal position
    float           CaptionOffsetY;   // Vertical offset of the caption from its normal position
    TLineCaptionPos CaptionPos;       // The position where the caption should be drawn if present
    float           LeaderLineLen;    // Length of the leader lines (positive or negative)
    float           LeaderLineExtend;// Optional leader line extend beyond the leader line
    float           LeaderLineOffset;// Amount of space between the endpoints of the annotation and
                                      // the leader lines.

    TBorderEffect   BorderEffect;     // Circle, Square, FreeText, and Polygon annotations.
    char*           RichStyle;        // Optional default style string. -> FreeText annotations.
    char*           RichText;         // Optional rich text string (RC key). -> Markup annotations.
    const ILST**    InkList;          // Ink annotations only. Array of array. The sub arrays can be
                                      // accessed with GetInkList().
    UI32            InkListCount;     // Number of ink arrays.
    SI32            OC;               // Handle of an OCG or OCMD or -1 if not set. See description.
};
```

The function returns the most important properties of an annotation. The parameter *Handle* must be a valid annotation handle. All string values are pointers to the original string buffer. You must not modify or free such a string. Reserved fields must be initialized with NULL.

The function returns also deleted annotations because DeleteAnnotation() does not physically delete annotations.

The members *QuadPoints* and *Vertices* contain the raw floating point values which are stored in the annotation. The corresponding values *QuadPointCount* and *VerticesCount* contain the number of floating point values and **not** the number of vertices or quad points which are stored in the arrays.

Note that the number of values can be odd if an imported annotation contains errors.

Optional Content

Annotations can be part of an Optional Content Group (OCG) or Optional Content Membership Dictionary (OCMD). If *OC* is greater -1 an OCG or OCMD handle was set. To determine whether an OCG handle was set, check whether the value is smaller 0x40000000:

```
bool IsOCG(SI32 Handle)
{
    return (Handle > -1 && Handle < 0x40000000);
}
```

Migration states

The migration state of a base annotation is stored in text annotations. The base annotation can be an arbitrary markup annotation, incl. text annotations. A text annotation represents a migration state if *Parent* > -1, if *Group* is false, and if *State* and *StateModel* are non-null.

The migration states of a base annotation are usually stored as single linked list. That means the last state is a just reply of the previous state and so on. The base annotation is the one that has no parent and this annotation must be a markup annotation.

Return values:

If the function succeeds the return value is 1 and the parameter *Annot* is filled with values. If the function fails the return value is 0 and the parameter *Annot* is left unchanged.

GetAnnotFlags

Syntax:

```
SI32 pdfGetAnnotFlags(
    const PPDF* IPDF) // Instance pointer
typedef UI32 TAnnotFlags;
#define afNone      0x00000000 // No flags are set
#define afInvisible 0x00000001 // see below
#define afHidden    0x00000002 // see below
#define afPrint     0x00000004 // Annotation is printable
#define afNoZoom    0x00000008 // Do not zoom the annotation
#define afNoRotate  0x00000010 // Do not rotate the annotation
#define afNoView    0x00000020 // see below
#define afReadOnly  0x00000040 // Changes are not allowed
```

The function returns the default flags used for newly created annotations. The return value is a bit mask; the flags must be mask out with the bitwise and operator.

The flags are described in detail on the next page.

Flag	Description
afNone	No flags are set.
afInvisible	If set, do not display the annotation if it does not belong to one of the standard annotation types and no annotation handler is available.
afHidden	*(PDF 1.2)* If set, do not display or print the annotation or allow it to interact with the user, regardless of its annotation type or whether an annotation handler is available.
afPrint	*(PDF 1.2)* If set, print the annotation when the page is printed. If clear, never print the annotation, regardless of whether it is displayed on the screen. This can be useful, for example, for annotations representing interactive pushbuttons, which would serve no meaningful purpose on the printed page.
afNoZoom	*(PDF 1.3)* If set, do not scale the annotation's appearance to match the magnification of the page. The location of the annotation on the page (defined by the upper-left corner of its annotation rectangle) remains fixed, regardless of the page magnification.
afNoRotate	*(PDF 1.3)* If set, do not rotate the annotation's appearance to match the rotation of the page. The upper-left corner of the annotation rectangle remains in a fixed location on the page, regardless of the page rotation.

Flag	Description
afNoView	*(PDF 1.3)* If set, do not display the annotation on the screen or allow it to interact with the user. The annotation may be printed (depending on the setting of the afPrint flag), but should be considered hidden for purposes of on-screen display and user interaction.
afReadOnly	*(PDF 1.3)* If set, do not allow the annotation to interact with the user. The annotation may be displayed or printed (depending on the settings of the afNoView and afPrint flags), but should not respond to mouse clicks or change its appearance in response to mouse motions.

GetAnnotLink

Syntax:

```
char* pdfGetAnnotLink(
   const PPDF* IPDF, // Instance pointer
   UI32 Handle)      // Annotation handle
```

This function returns the URL or file path of a file link annotation. The parameter *AHandle* must be a valid handle of a file link, page link, or web link annotation. The return value is a pointer to the original string value. The string must not be changed or freed within your application.

Remarks:

Imported file link or web link annotations are mostly defined as page link annotation because it is impossible to distinguish between these types during import. Use the function FindLinkAnnot() if you need to change an imported file link or web link annotation.

Return values:

If the function succeeds the return value is a pointer to the original string value. If the function fails the return value is NULL.

GetAnnotType

Syntax:

```
SI32 pdfGetAnnotType(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle)      // Annotation handle

typedef enum
{
    atCaret,       // Caret annotation
    atCircle,      // Circle annotation
    atFileLink,    // A Link annotation with an associated GoToR action
    atFreeText,
    atHighlight,   // Highlight annotation
    atInk,
    atLine,
    atPageLink,    // A Link annotation with an associated GoTo action
    atPolygon,
    atPolyLine,
    atPopUp,
    atSquare,
    atSquiggly,    // Highlight annotation
    atStamp,
    atStrikeOut,   // Highlight annotation
    atText,        // Also used as container to store the State Model
    atUnderline,   // Highlight annotation
    atWebLink,     // A Link annotation with an associated URI action
    atWidget,      // Form Fields are handled separately
    at3D,          // PDF 1.6
    atSoundAnnot,  // PDF 1.2
    atFileAttach,  // PDF 1.3
    atRedact,      // PDF 1.7
    atWatermark,   // PDF 1.6
    atUnknown,     // Unknown annotation type
    atMovieAnnot,  // PDF 1.2
    atPrinterMark, // PDF 1.4
    atProjection,  // PDF 1.7 Extension Level 3
    atRichMedia,   // PDF 1.7 Extension Level 3
    atScreen,      // PDF 1.5
    atTrapNet      // PDF 1.3
}TAnnotType;
```

The function returns the type of a specific annotation. The parameter *Handle* must be a valid annotation handle. If the function succeeds the return value is the annotation type (make a type cast to TAnnotType to determine the annotation type). If the function fails the return value is a negative error code.

GetAscent

Syntax:

```
double pdfGetAscent(
    const PPDF* IPDF) // Instance pointer
```

The function returns the ascender of the active font. The ascender is a typographic value that specifies the maximum extent to which characters rise above the baseline.

If no font is set the return value is a negative error code.

GetBarcodeDict

Syntax:

```
LBOOL pdfGetBarcodeDict(
    const PPDF* IBarcode,          // Pointer of a barcode dictionary
    struct TPDFBarcode* Barcode) // see below
```

```
struct TPDFBarcode
{
    UI32        StructSize;   // Must be set to sizeof(TPDFBarcode)
    const char* CaptionA;     // Optional
    const UI16* CaptionW;     // Optional
    float       ECC;          // 0..8 for PDF417, or 0..3 for QRCode
    float       Height;       // Height in inches
    float       nCodeWordCol; // Number of codewords per barcode coloumn
    float       nCodeWordRow; // Codewords per barcode row (PDF417)
    UI32        Resolution;   // Resolution
    const char* Symbology;    // PDF417, QRCode, or DataMatrix.
    float       Version;      // Version
    float       Width;        // Width in inches
    float       XSymHeight;   // Only needed for PDF417. The vertical
                              // distance between two barcode modules,
                              // measured in pixels. The ratio XSymHeight /
                              // XSymWidth shall be an integer value. For
                              // PDF417, the acceptable ratio range is from
                              // 1 to 4. For QRCode and DataMatrix, this
                              // ratio shall always be 1.
    float       XSymWidth;    // The horizontal distance, in pixels,
                              // between two barcode modules.
};
```

The function returns the properties of a barcode field. The member *StructSize* must be initialized to sizeof(TPDFBarcode) before the function can be called. The structure size is used to identify different versions of the structure. The value of the barcode is stored in the corresponding text field.

Return values:

If the function succeeds the return value is 1 and the structure is filled with values. If the function fails the return value is zero. Check whether the member StructSize was properly initialized in the latter case.

GetBBox

Syntax:

```
LBOOL pdfGetBBox(
    const PPDF* IPDF,          // Instance pointer
    TPageBoundary Boundary,    // see below
    struct TPDFRect ADDR BBox) // Bounding box

typedef enum
{
    pbArtBox,   // Art box
    pbBleedBox, // Bleed box
    pbCropBox,  // Crop box
    pbTrimBox,  // Trim box
    pbMediaBox  // Media box
}TPageBoundary;

struct TPDFRect
{
    double Left;
    double Bottom;
    double Right;
    double Top;
};
```

The function returns a bounding box of a PDF page. A PDF page may be prepared either for a finished medium, such as a sheet of paper, or as part of a prepress process in which the content of the page is placed on an intermediate medium, such as film or an imposed reproduction plate.

In the latter case, it is important to distinguish between the intermediate page and the finished page. The intermediate page may often include additional production-related content, such as bleeds or printer marks that falls outside the boundaries of the finished page. To handle such cases, a PDF page can define as many as five separate boundaries to control various aspects of the imaging process.

The function requires an open page because each page can use its own boundaries.

Return values:

If the function succeeds the return value is 1 and the structure *BBox* is filled with values. If the function fails the return value is 0.

Bounding boxes:
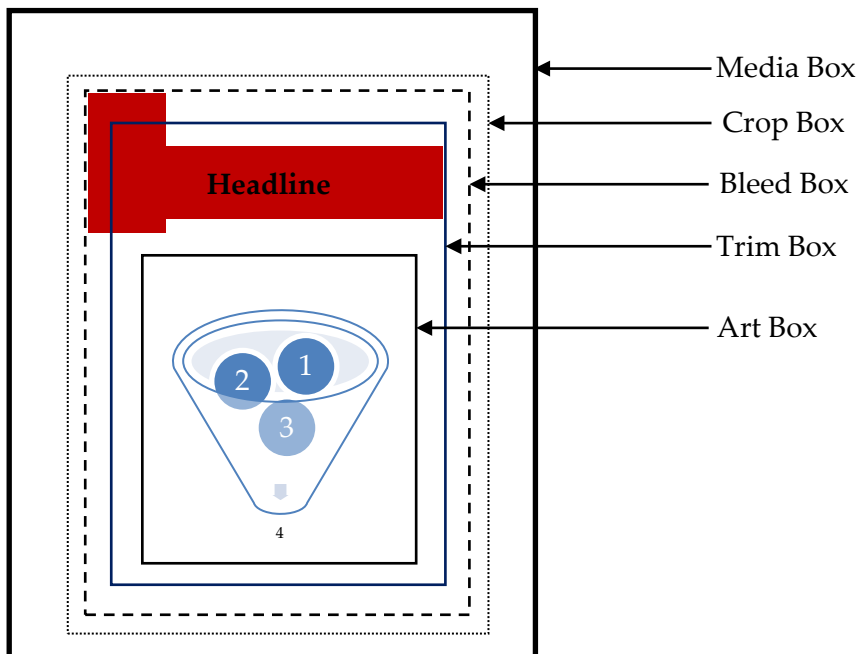
The media box represents the coordinate system or physical extend of a page. The crop box crops the media box and all other boxes. No box can be larger than the media box.

The media box represents the page format if no crop box is set, the crop box otherwise. All boxes are measured in PDF units and one PDF unit is 1/72 inch.

GetBidiMode

Syntax:

```
SI32 pdfGetBidiMode(
   const PPDF* IPDF) // Instance pointer
typedef enum
{
    bmLeftToRight = 0, // Apply the bidi algorithm in Left to Right layout
    bmRightToLeft = 1, // Apply the bidi algorithm in Right to Left layout
    bmNone        = 2  // Default -> do not apply the bidi algorithm
}TPDFBidiMode;
```

The function returns the current bidirectional mode. The possible return values are defined in the enum TPDFBidiMode. See also SetBidiMode().

GetBookmark (obsolete)

Syntax:

```
LBOOL pdfGetBookmark(
    const PPDF* IPDF,            // Instance pointer
    SI32 AHandle,                // Bookmark handle
    struct TBookmark ADDR Bmk)   // see below

struct TBookmark
{
    UI32 Color;               // Bookmark color (PDF 1.4)
    SI32 DestPage;            // Destination page
    struct TPDFRect DestPos;  // Destination position
    TDestType DestType;       // Destination type (see SetBookmarkDest())
    LBOOL Open;               // True if the bookmark appears open
    SI32 Parent;              // Parent bookmark if any or -1
    TBmkStyle Style;          // Text style, see below (PDF 1.4)
    const void* Title;        // Pointer to text string
    UI32 TitleLen;            // Title length in characters
    LBOOL Unicode;            // If true, Title is a pointer to UI16*
};

typedef enum
{
    bmsNormal = 0,
    bmsItalic = 1,
    bmsBold   = 2
}TBmkStyle;
```

This function returns the properties of a bookmark. This function is marked as obsolete. Please use
GetBookmarkEx() if possible.

GetBookmarkEx

Syntax:

```
LBOOL pdfGetBookmarkEx(
    const PPDF* IPDF,            // Instance pointer
    SI32 Handle,                 // Bookmark handle
    struct TPDFBookmark* Bmk)    // Structure to be filled

struct TPDFBookmark
{
    UI32          StructSize;   // Must be set to sizeof(TPDFBookmark)
    SI32          Action;       // Action handle or -1 if not set.
    UI32          Color;        // Optional text color. Black is the default value (0).
    SI32          DestPage;     // Destination page.
    struct TPDFRect DestPos;    // Destination position.
    TDestType     DestType;     // Destination type.
    const char*   NamedDestA;   // If set, DestPage, DestPos, and DestType were already taken from
                                // the destination if it was found in the file.
    const UI16*   NamedDestW;   // Either the Unicode or Ansi string is set but never both.
    LBOOL         Open;         // If true, child nodes should be visible.
    SI32          Parent;       // The immediate parent node or -1 for a root node.
    const void*   StructElem;   // Bookmarks can contain a reference to a structure element. There
                                // is no API to access structure elements yet.
    TBmkStyle     Style;        // The font style that should be used to render this node.
    const char*   TitleA;       // The bookmark title.
    const UI16*   TitleW;       // Either the Unicode or Ansi string is set but never both.
    UI32          TitleLen;     // String length in characters.
};
```

This function returns the properties of a bookmark. The parameter *Bmk* is required it must not be NULL. The member *Style* is a bitmask, maybe more than one constant is set at time, i.e. *bmsBold* and *bmsItalic*. The destination types are described in detail at SetBookmarkDest().

Return values:

If the function succeeds the return value is 1 and the structure *Bmk* is filled with values. If the function fails the return value is 0.

GetBookmarkCount

Syntax:

```
SI32 pdfGetBookmarkCount(
    const PPDF* IPDF) // Instance pointer
```

The function returns the number of bookmarks defined in the document.

GetBorderStyle

Syntax:

```
SI32 pdfGetBorderStyle(
    const PPDF* IPDF) // Instance pointer
typedef enum
{
    bsSolid       = 0, // Solid border
    bsBevelled    = 1, // Bevelled border
    bsInset       = 2, // Inset border
    bsUnderline   = 3, // Underline only
    bsDashed      = 4, // Dashed border
    bsUserDefined = 5  // Internal, cannot occur
}TBorderStyle;
```

The function returns the global border style which is used for newly created Interactive Form fields. To determine the style of a specific field use the function GetFieldBorderStyle().

GetBuffer

Syntax:

```
char* pdfGetBuffer(
    const PPDF* IPDF,  // Instance pointer
    UI32 ADDR BufSize) // Buffer length in bytes
```

The function returns a pointer to the buffer of a memory based PDF file. A memory based PDF file will be created if the parameter *OutFile* of the function CreateNewPDF() is an empty string or set to NULL.

The parameter *BufSize* is set to the size of the buffer in bytes. The returned pointer is a pointer to the original buffer; it must not be freed or changed. When the buffer is not longer needed call FreePDF() to release the PDF file.

GetCapHeight

Synatax:

```
double pdfGetCapHeight(
    const PPDF* IPDF) // Instance pointer
```

The function returns the cap height of the active font. The cap height is a typographic value that specifies the maximum height of a character in the active font without serifs.

GetCharacterSpacing

Syntax:

```
double pdfGetCharacterSpacing(
    const PPDF* IPDF) // Instance pointer
```

The function returns the current character spacing.

```
Default value = 0
```

Value = 0	Character
Value = 10.0	C h a r a c t e r

GetCheckBoxChar

Syntax:

```
SI32 pdfGetChecBoxChar(
    const PPDF* IPDF) // Instance pointer

typedef enum
{
   ccCheck,
   ccCircle,
   ccCross1,
   ccCross2,
   ccCross3,
   ccCross4,
   ccDiamond,
   ccSquare,
   ccStar
}TCheckBoxChar;
```

The function returns the character used for newly created check boxes.

Check box characters

ccCheck

ccCircle

ccCross1

ccCross2

ccCross3

ccCross4

ccDiamond

ccSquare

ccStar

GetCheckBoxCharEx

Syntax:

```
SI32 pdfGetCheckBoxCharEx(
  const PPDF* IPDF, // Instance pointer
  UI32 AField)      // Handle of a check box
```

The function returns character index of the font ZapfDingbats that is used to display the on state of the check box. If the function fails the return value is a negative error code.

GetCheckBoxDefState

Syntax:

```
SI32 pdfGetCheckBoxDefState(
  const PPDF* IPDF, // Instance pointer
  UI32 AField)      // Handle of a check box
```

This function returns the default state of a check box, that is 1 == checked or 0 == unchecked. The parameter *AField* must be a valid handle of a check box. The default state can differ from the current visible state of a check box. The default state is used when a form is reset with a Reset Form Action.

Return values:

If the function succeeds the return value is either 0 or 1 depending on the default state. If the function fails the return value is a negative error code.

GetCMap

Syntax:

```
LBOOL pdfGetCMap(
  const PPDF* IPDF,       // Instance pointer
  UI32 Index,             // CMap's array index
  struct TPDFCMap* CMap) // Structure to be filled with properties
struct TPDFCMap
{
    UI32  StructSize;      // Must be set to sizeof(TPDFCMap)!
    char* BaseCMap;        // Required base CMap if any
    UI32  CIDCount;        // 0 if not set
    char* CMapName;        // The CMap name
    UI32  CMapType;        // Should be one!
    float CMapVersion;     // The CMap version
    char* DSCBaseCMap;     // DSC comment
    float DSCCMapVersion;  // DSC comment
    char* DSCResName;      // DSC comment
    char* DSCTitle;        // DSC comment
    char* FileNameA;       // Either the Ansi or Unicode file name is set
    UI16* FileNameW;       // Either the Ansi or Unicode file name is set
    char* FilePathA;       // Either the Ansi or Unicode path is set
    UI16* FilePathW;       // Either the Ansi or Unicode path is set
    char* Ordering;        // Supported Character Collection
    char* Registry;        // The registrant (this is usually Adobe)
    UI32  Supplement;      // The Supplement number of this CMap
    UI32  WritingMode;     // 0 == Horizontal, 1 == Vertical
};
```

The function returns the most important properties of an external CMap file. The search path of external CMaps can be set with SetCMapDir(). The function returns the number of available CMap files which can be accessed with GetCMap(). It is also possible to use the function GetCMapCount() to determine the number of available CMap files.

The member StructSize of the structure TPDFCMap must be initialized with sizeof(TPDFCMap) before calling the function (C/C++, Delphi only). The structure size is used to identify different versions of the structure if extensions will be made in later releases.

The parameter *Index* represents an array index into the CMap cache. It can be in the range 0..NumCMaps -1.

The member *CMapName* of the structure TPDFCMap should be used to load a CMap into memory with LoadCMap(). Note that the function LoadCMap() returns a CMap handle that cannot be used with this function.

Values which are taken from the DSC comment section contain the prefix DSC. DSC comments are optional and maybe not available when using CMap files from other vendors than Adobe.

Adobe provides for all character collections identity mappings; the member *DSCResName* is set to Identity in this case so that such CMaps can be easily identified.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetCMapCount

Syntax:

```
SI32 pdfGetCMapCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of available CMap files. This value can be used to access the header information of CMap files with GetCMap().

GetCollectionInfo

Syntax:

```
LBOOL pdfGetCollectionInfo(
   const PPDF* IPDF,                   // Instance pointer
   struct TPDFCollectionInfo* Value) // Structure to be filled

typedef enum TPDFColSplitInfo
{
   spiNone,       // Do not display a splitter
   spiHorizontal, // Display a horizontal splitter
   spiVertical    // Display a vertical splitter
}TPDFColSplitInfo;
```

```
struct TPDFCollectionInfo
{
    UI32            StructSize; // Must be set to sizeof(TPDFCollectionInfo)
    SI32            InitalFile; // Possible values are -2, -1, or a valid embedded files handle.
    TColView        InitalView; // Determines how embedded files should be displayed.
    const char*     SortBy;     // Optional name of the first field that should be used to sort the
                                // list of embedded files.
    LBOOL           SortDesc;   // If true, the list of embedded files should be sorted in
                                // descending order.
    TPDFColSplitInfo SplitInfo; // Determines whether a splitter bar should be displayed.
    float           SplitPos;   // Meaningful only, if SplitInfo is not spiNone. The initial
                                // position of the splitter bar, specified as a percentage of the
                                // available window area (0 to 100).
};
```

The function returns the most important properties of a PDF Collection, also called PDF Portfolio. A PDF collection consists of a container PDF and a set of embedded files which make up the collection.

The embedded files are mostly PDF files but can be of arbitrary type.

Remarks:

Adobe has changed the way how PDF Collections are displayed in Acrobat or Reader several times. PDF Collections were initially fully integrated in Adobes Acrobat or Reader. Beginning with Acrobat 10 the user interface based on Flash but Flash became obsolete and was finally fully disabled. PDF collections do no longer work with Acrobat 10 or 11 at all.

In Acrobat DC PDF collections work again but the implementation is very restricted and it is not known whether Adobe will maintain this feature anymore.

Due to the unclear status it is not planned to add further functions in order to access all available properties of PDF collections.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetColorSpace

Syntax:

```
SI32 pdfGetColorSpace(
    const PPDF* IPDF) // Instance pointer

typedef enum
{
    esDeviceRGB  = 0,  // Device color space
    esDeviceCMYK = 1,  // Device color space
    esDeviceGray = 2,  // Device color space
    esCalGray    = 3,  // CIE-based color space
    esCalRGB     = 4,  // CIE-based color space
    esLab        = 5,  // CIE-based color space
    esICCBased   = 6,  // ICC-based color space -> contains an ICC profile
    esPattern    = 7,  // Special color space
    esIndexed    = 8,  // Special color space
    esSeparation = 9,  // Special color space
    esDeviceN    = 10, // Special color space
    esNChannel   = 11, // Special color space
    esInitSpace  = -1  // Internal -> This value cannot occur.
}TExtColorSpace;
```

The function returns the active color space.

GetColorSpaceCount

Syntax:

```
SI32 pdfGetColorSpaceCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of color space objects which are used in the current document. Color spaces can be accessed with GetColorSpaceObj().

GetColorSpaceObj

Syntax:

```
LBOOL pdfGetColorSpaceObj(
   const PPDF* IPDF)              // Instance pointer
   UI32 Handle,                   // Color space handle
   struct TPDFColorSpaceObj* CS)  // Structure to be filled

struct TPDFColorSpaceObj
{
   TExtColorSpace Type;
   TExtColorSpace Alternate;// Alternate or base space of Indexed or Pattern color spaces.
   void*  IAlternate;      // Optional alternate color space.
   BYTE*  Buffer;          // ICC profile or color table of an Indexed color space.
   UI32   BufSize;         // Buffer length in bytes.
   float* BlackPoint;      // CIE black point. If set, the array contains exactly 3 values.
   float* WhitePoint;      // CIE white point. If set, the array contains exactly 3 values.
   float* Gamma;           // If set, one value per component.
   float* Range;           // min/max per component or .a .b components of a Lab color space.
   float* Matrix;          // XYZ matrix. If set, the array contains exactly 9 values.
   UI32   NumInComponents;  // Number of input components.
   UI32   NumOutComponents; // Number of output components.
   UI32   NumColors;       // HiVal + 1. Indexed color space only.
   BYTE*  Colorants[32];   // Colorant names (Separation, DeviceN, and NChannel only).
   UI32   ColorantsCount;  // The number of colorants in the array.
   BYTE*  Metadata;        // Optional XMP metadata stream -> ICCBased only.
   UI32   MetadataSize;    // Metadata length in bytes.
   void*  IFunction;       // Pointer to function object -> Special color spaces only
   void*  IAttributes;     // Optional attributes of DeviceN or NChannel color spaces
   void*  IColorSpaceObj;  // Pointer of the corresponding color space object
   void*  Reserved01;
   void*  Reserved02;
   void*  Reserved03;
   void*  Reserved04;
   void*  Reserved05;
   void*  Reserved06;
   void*  Reserved07;
   void*  Reserved08;
   void*  Reserved09;
};
```

The function retrieves the most important properties of a color space. The parameter handle must be a valid handle of a color space. A color space handle is a simple array index. The number of color spaces is returned by the function GetColorSpaceCount().Reserved fields of the structure TPDFColorSpaceObj must be initialized with NULL.

The alternate color space of special color spaces can be accessed with GetColorSpaceObjEx(). The attributes of a DeviceN or NChannel color space can be accessed with GetDeviceNAttributes().

Note that the range of a Lab color space consists only of four values because the L component has a fixed range from 0 through 100.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetColorSpaceObjEx

Syntax:

```
LBOOL pdfGetColorSpaceObjEx(
   const PPDF* IColorSpace,       // Pointer of a color space object
   struct TPDFColorSpaceObj* CS) // Structure to be filled
```

The function retrieves the most important properties of a color space like GetColorSpaceObj() but accepts a pointer of a color space object instead. See also GetColorSpaceObj().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetCompressionFilter

Syntax:

```
SI32 pdfGetCompressionFilter(
   const PPDF* IPDF) // Instance pointer

typedef enum
{
   cfFlate = 0,
   cfJPEG  = 1,
   cfJP2K  = 7
}TCompressionFilter;
```

The function returns the standard compression filter for images. Note that 1 bit images are always compressed with Flate.

GetCompressionLevel

Syntax:

```
SI32 pdfGetCompressionLevel(
    const PPDF* IPDF) // Instance pointer
typedef enum
{
    clNone    = 0,
    clDefault = 1,
    clFastest = 2,
    clMax     = 3
}TCompressionLevel;
```

The function returns the active compression level. JPEG images are compressed with optimized Huffmann tables if the compression level is clMax. This results in 5% to 10% better compression ratio without losing quality. The compression level can also be used to switch the compression mode to real or integer when using the JPEG2000 compression filter (see also InsertImage()).

GetContent

Syntax:

```
SI32 pdfGetContent(
    const PPDF* IPDF,   // Instance pointer
    char* ADDR BufSize) // Buffer
```

This function returns a pointer to the content stream of the currently open page or template. The parameter *BufSize* gets the buffer size in bytes; this parameter must not be set to NULL.

The page or template must be open with EditPage() or EditTemplate() beforehand.

When no further objects must be added it is possible to edit the buffer directly to improve processing speed. However, when changing the buffer size it is highly recommended to update the content stream with the function SetContent().

When new contents must be added, copy the buffer into a new one and replace the content stream with SetContent() when finished. Do not change a content stream when you don't know exactly what you are doing. If the content stream contains invalid operators after editing the PDF file will be damaged.

Remarks:

The function returns a pointer to the original content buffer of the currently open page or template. DynaPDF is the owner of this buffer, it must not be freed in you application.

Return values:

If the function succeeds the return value is the buffer size if bytes. If the function fails the return value is a negative error code.

GetDefBitsPerPixel

Syntax:

```
SI32 pdfGetDefBitsPerPixel(
    const PPDF* IPDF) // Instance pointer
```

The function returns default color depth in bits per pixel, which determines whether images should be down sampled. If the return value is 8 images are converted to 256 indexed color images.

At time of publication only two values are supported:

- *24 bit*: No conversion
- *8 bit*: Conversion to 256 indexed color image

```
Default value = 24
```

GetDescent

Syntax:

```
double pdfGetDescent(
    const PPDF* IPDF) // Instance pointer
```

The function returns the descender of the active font. The descender is a typographic value that specifies the maximum extent to which characters in the active font descend below the baseline.

The descender is normally a negative value. However, it is always returned as a positive value to make the usage easier.

Return values:

If the function succeeds the return value is the typographic descender of the active font as positive double. If the function fails the return value is a negative error code.

GetDeviceNAttributes

Syntax:

```
LBOOL pdfGetDeviceNAttributes(
  void* IAttributes,                      // Pointer to attribute object
  struct TDeviceNAttributes* Attributes) // Structure to be filled

struct TDeviceNAttributes
{
    void* IProcessColorSpace;    // Pointer to process color space
    BYTE* ProcessColorants[8];   // Process colorant names
    UI32  ProcessColorantsCount; // Number of process colorants
    void* Separations[32];       // Pointers to separation color spaces
    UI32  SeparationsCount;      // Number of separation color spaces
    void* IMixingHints;          // Pointer to optional mixing hints.
    void* Reserved01;
    void* Reserved02;
    void* Reserved03;
    void* Reserved04;
};
```

The function retrieves attributes of a DeviceN or NChannel color space. The parameter *IAttributes* must be a valid pointer of an attribute object. This pointer is available in the structure TPDFColorSpaceObj. See GetColorSpaceObj() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetDocInfo

Syntax:

```
SI32 pdfGetDocInfo(
    const PPDF* IPDF,    // Instance pointer
    TDocumentInfo DInfo, // Document info entry
    UI16* ADDR Value)    // Value as Unicode string (null-terminated)

typedef enum
{
    diAuthor      = 0,
    diCreator     = 1,
    diKeywords    = 2,
    diProducer    = 3,
    diSubject     = 4,
    diTitle       = 5,
    diCompany     = 6,
    diPDFX_Ver    = 7,  // GetInDocInfo() or GetInDocInfoEx()
    diCustom      = 8,  // Not supported -> Use GetDocInfoEx()
    diPDFX_Conf   = 9,  // GetInDocInfo() or GetInDocInfoEx()
    diCreationDate = 10, // Available after a PDF file was imported
    diModDate     = 11  // GetInDocInfo() or GetInDocInfoEx()
}TDocumentInfo;
```

The function retrieves a document info entry as Unicode string. The parameter *Value* holds a pointer to the original Unicode value, it must not be NULL. Do not change or free the value. GetDocInfo() returns only the values of predefined document info entries. All document info entries, incl. user defined keys can be accessed with GetDocInfoEx().

Return value:

The return value is the string length in characters. If the info entry is not set it returns 0 and *Value* is initialized to NULL.

GetDocInfoCount

Syntax:

```
SI32 pdfGetDocInfoCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of document info entries defined in the document. This function can be used in combination with GetDocInfoEx() to enumerate the document info entries of the PDF file.

GetDocInfoEx

Syntax:

```
SI32 pdfGetDocInfoEx(
   const PPDF* IPDF,        // Instance pointer
   UI32 Index,              // Entry index
   TDocumentInfo ADDR DInfo, // Document info entry type
   char* ADDR Key,          // Only set for user defined keys
   char* ADDR Value,        // The value of the info entry
   LBOOL ADDR Unicode)      // Is value in Unicode format?
```

The function returns a document info entry. The parameter *Index* must be a valid index into the array of document info entries. The number of available entries is returned by GetDocInfoCount(). The parameter *Value* holds a pointer to the original value, it must not be NULL. Do not change or free the value. If the parameter *Unicode* is true, the value is a Unicode string. Make a type cast to UI16* in this case. The parameter *Key* contains always an Ansi string if set (user defined keys only).

Return values:

If the function succeeds the return value is the string length in characters of the parameter *Value*. Depending on the string format make a type cast to UI16*.

GetDocUsesTransparency

Syntax:

```
LBOOL pdfGetDocUsesTransparency(
    const PPDF* IPDF, // Instance pointer
    UI32 Flags)       // No flags defined yet, the parameter is ignored
```

The function checks whether a PDF file uses native PDF Transparency (PDF 1.4). The file uses transparency when it contains soft masks, blend modes other than Normal, or fill or stroke alpha values smaller than 1.0.

The function checks all pages, templates, extended graphics states, images, annotations, and form fields.

Return values:

If the file uses transparency the return value is 1. If it uses no transparency the return value is 0.

GetDrawDirection

Syntax:

```
SI32 pdfGetDrawDirection(
    const PPDF* IPDF) // Instance pointer
```

```
typedef enum
{
    ddCounterClockwise = 0,
    ddClockwise        = 1
}TDrawDirection;
```

The function returns the actual draw direction for closed vector graphics such as rectangles, circles, ellipses and so on.

GetDynaPDFVersion

Syntax:

```
char* pdfGetDynaPDFVersion(void)
```

The function returns the version string of DynaPDF. The return value is a pointer to a null-terminated static string. The caller must not change or free the string.

GetDynaPDFVersionInt

Syntax:

```
SI32 pdfGetDynaPDFVersionInt(void)
```

The function returns the DynaPDF version as an integer. This is a pure interface function that parses the version string returned by GetDynaPDFVersion(). Therefore, the function is compatible with all DynaPDF versions.

A DynaPDF version string consists of 4 four parts: major version, revision (unused, always zero), minor version, and build number.

Example:

4.0.3.7 = Major = 4, revision (always zero), minor = 3, build = 7.

The same version as an integer would be calculated as follows:

```
Version = Major * 10000000 + Minor * 10000 + Build
```

The above version string would be returned as 40030007 = 4 * 10000000 + 3 * 10000 + 7.

Return values:

If the function succeeds the return value is the version number, a value greater zero. If the function fails the return value is zero.

GetEmbeddedFile

Syntax:

```
BOOL pdfGetEmbeddedFile(
    const PPDF* IPDF,                  // Instance pointer
    UI32 Handle,                       // Handle of an embedded file
    struct TPDFFileSpec* FileSpec,     // See below
    LBOOL Decompress)                  // If true, the file is decompressed

struct TPDFFileSpec
{
    char* Buffer;        // Buffer of an embedded file.
    UI32  BufSize;       // Buffer size in bytes.
    LBOOL Compressed;    // Should be false if Decompress was true.
    void* ColItem;       // Pointer to user defined collection item.
    char* Name;          // Name of the file specification in the name tree.
    LBOOL NameUnicode;   // Is Name in Unicode format?
    char* FileName;      // File name as 7 bit ASCII string.
    LBOOL IsURL;         // If true, FileName contains a URL.
    char* UF;            // PDF 1.7. Same as FileName but Unicode is allowed.
    LBOOL UFUnicode;     // Is UF in Unicode format?
    char* Desc;          // Description.
    LBOOL DescUnicode;   // Is Desc in Unicode format?
    UI32  FileSize;      // Decompressed stream size or zero if not known.
    char* MIMEType;      // MIME media type name (RFC 2046).
    char* CreateDate;    // Creation date as string.
    char* ModDate;       // Modification date as string.
    char* CheckSum;      // 16 byte MD5 digest if set.
};
```

The function retrieves the most important properties of an embedded file as well as a pointer to the file buffer. If the parameter *Decompress* is set to true the embedded file stream is decompressed. Set this parameter to true if you want to extract the embedded file. Otherwise the parameter should be set to false. DynaPDF decompresses Flate compressed embedded file streams only. However, other filters can occur but this is unusual. If the file buffer was successfully decompressed the member Decompressed of the structure TPDFFileSpec is set to true.

Remarks:

The structure TPDFFileSpec contains pointers to the original values of the embedded file. These values must not be changed or freed in your application.

Return values:

If the function succeeds the return value 1 and the structure TPDFFileSpec is filled with values. If the function fails the return value is 0.

GetEmbeddedFileCount

Syntax:

```
SI32 pdfGetEmbeddedFileCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of embedded files available in the PDF file. Note that the number of embedded files from an external PDF file is available after the PDF file has been imported. Depending of the used import flags embedded files can be excluded from import. See SetImportFlags(), or ImportPDFFile() for further information.

GetEmbeddedFileNode

Syntax:

```
LBOOL pdfGetEmbeddedFileNode(
   const IEFN* IEF,          // Pointer of an embedded file node
   struct TPDFEmbFileNode* F, // see below
   LBOOL Decompress)          // If true, the file will be decompressed
struct TPDFEmbFileNode
{
    UI32               StructSize; // Must be set to sizeof(TPDFEmbFileNode).
    const char*        Name;       // UTF-8 encoded name. This is usually an ASCII string.
    struct TPDFFileSpec EF;        // Embedded file.
    IEFN*              NextNode;   // Next node if any.
};
struct TPDFFileSpec
{
    char* Buffer;       // Buffer of an embedded file.
    UI32  BufSize;      // Buffer size in bytes.
    LBOOL Compressed;   // Should be false if Decompress was true.
    void* ColItem;      // Pointer to user defined collection item.
    char* Name;         // Name of the file specification in the name tree.
    LBOOL NameUnicode;  // Is Name in Unicode format?
    char* FileName;     // File name as 7 bit ASCII string.
    LBOOL IsURL;        // If true, FileName contains a URL.
    char* UF;           // PDF 1.7. Same as FileName but Unicode is allowed.
    LBOOL UFUnicode;    // Is UF in Unicode format?
    char* Desc;         // Description.
    LBOOL DescUnicode;  // Is Desc in Unicode format?
    UI32  FileSize;     // Decompressed stream size or zero if not known.
    char* MIMEType;     // MIME media type name (RFC 2046).
    char* CreateDate;   // Creation date as string.
    char* ModDate;      // Modification date as string.
    char* CheckSum;     // 16 byte MD5 digest if set.
};
```

The function retrieves the properties of an embedded file node. The member *StructSize* of the structure TPDFEmbFileNode must be set to sizeof(TPDFEmbFileNode) before the function can be called.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetEMFPatternDistance

Syntax:

```
double pdfGetEMFPatternDistance(
   const PPDF* IPDF) // Instance pointer
```

The function returns the distance between lines of standard patterns during EMF conversion.

Default value = 4.0

GetErrLogMessage

Syntax:

```
LBOOL pdfGetErrLogMessage(
   const PPDF* IPDF,      // Instance pointer
   UI32 Index,            // Array index
   struct TPDFError* Err) // see below
struct TPDFError
{
   UI32        StructSize; // Must be initialized to sizeof(TPDFError)
   const char* Message;    // The error message
   SI32        ObjNum;     // PDF object number or -1 if not available
   SI32        Offset;     // File offset or -1 if not available
   const char* SrcFile;    // Source file
   UI32        SrcLine;    // Source line
   SI32        ErrCode;    // ErrCode and ErrType are set if the error mode was set to emUseErrLog.
   SI32        ErrType;    // ErrCode and ErrType are set if the error mode was set to emUseErrLog.
};
```

The function returns an error object of the internal error log. The function GetErrLogMessageCount() returns the number of available error messages.

DynaPDF uses the error log to store non-fatal errors or warnings when it is not required to break processing or if more detailed error information should be provided than is possible with the normal exception handling.

For example, when opening a damaged PDF file with OpenImportFile() the function tries first to open the file in normal mode. All errors which occur in this step are written to the error log. Depending on the kind of found errors, the function maybe falls back into repair mode and reads the file again. If this action is successful then the function returns with success and no error message is provided over the normal exception handling. So, you'll not notice that the file contained damages unless you take a look into the error log.

The error log was introduced with DynaPDF 3.0. It is used when reading a PDF file, e.g. OpenImportFile(), ImportPage(), ImportPDFFile(), and when parsing or rendering PDF pages, e.g. ParseContent(), IsEmptyPage(), IsColorPage(), GetPageText(), RasterPage(), RasterPDFFile(), and so on.

The members *ErrCode* and *ErrType* are set only if the error mode was set to *emUseErrLog*. See SetErrorMode() for further information. *ErrType* is a bit mask that can contain the following flags:

```
#define E_WARNING      0x02000000
#define E_SYNTAX_ERROR 0x04000000
#define E_VALUE_ERROR  0x08000000
#define E_FONT_ERROR   0x10000000
#define E_FATAL_ERROR  0x20000000
#define E_FILE_ERROR   0x40000000
```

At time of publication only one flag is set at any one time. Future versions may be set multiple flags, e.g. E_SYNTAX_ERROR and E_WARNING. Because of this, it is recommended to mask out the error type with a bitwise *and* operator.

The number of possible error messages is restricted by default to 100 messages per PDF instance. However, it is possible to adjust the maximum number of messages with SetMaxErrLogMsgCount().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0. Check whether the member *StructSize* of the structure TPDFError was properly initialized in the latter case.

GetErrLogMessageCount

Syntax:

```
SI32 pdfGetErrLogMessageCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of error messages in the internal error log.

GetErrorMessage

Syntax:

```
char* pdfGetErrorMessage(
   const PPDF* IPDF) // Instance pointer
```

The function returns the last error message as pointer to a null-terminated static string, or NULL if no error occurred. Use this function if an error callback function can not be used (see also SetOnErrorProc()).

GetErrorMode

Syntax:

```
SI32 pdfGetErrorMode(
    const PPDF* IPDF) // Instance pointer

typedef SI32 TErrMode;
#define emIgnoreAll       0x00000000 // default
#define emSyntaxError     0x00000001
#define emValueError      0x00000002
#define emWarning         0x00000004
#define emFileError       0x00000008
#define emFontError       0x00000010
#define emAllErrors       0x0000FFFF
#define emNoFuncNames     0x10000000 // Do not output function names
// Special flags for CheckConformance().
// These flags add info messages to the error log. The error callback function is not invoked.
#define emLogFonts        0x00010000 // If set, CheckConformance() logs which fonts were replaced
                                     // with system fonts or converted to Type3.
#define emLogFontsVerbose 0x00020000 // If set, the path to the font file is added to the message too
```

The function returns the current error mode. The return value is a bit mask, to check whether a specific flag is set use a bitwise *and* operator.

The special flag emNoFuncNames names can be used to avoid the output of the function name in error messages. Error messages start normally always with the function name in which the error occurred. While this information is useful during development, it is often not useful in an end user application.

The meaning of the error mode flags is described in detail at *Customized Exception handling*.

GetField (obsolete)

Syntax:

```
LBOOL pdfGetField(
    const PPDF* IPDF,       // Instance pointer
    UI32 AHandle,           // Field handle
    struct TPDFField* Field) // see below

struct TPDFField
{
    SI32   FieldType;    // Field type (see TFieldType)
    LBOOL  Deleted;      // If true, the field was deleted
    struct TPDFRect BBox; // Bounding box
    SI32   Handle;       // Field handle
    char*  FieldName;    // Field name
    UI32   FieldNameLen; // Field name length in characters
    UI32   BackCS;       // Color space of background / border color
    UI32   TextCS;       // Color space of text color
    UI32   BackColor;    // Background color
    UI32   BorderColor;  // Border color
    UI32   TextColor;    // Text color
    LBOOL  Checked;      // Only set if the field is a check box
    SI32   Parent;       // Parent field handle if any
    UI32   KidCount;     // Greater zero if the field has childs
    char*  Font;         // Used font by the field (PostScript name)
    double FontSize;     // Font size
    void*  Value;        // Field value (char* or UI16*)
    LBOOL  UniVal;       // If True, Value is a Unicode string
    UI32   ValLen;       // String length in characters
```

```
    void*  ToolTip;       // Tool tip if any
    LBOOL  UniToolTip;    // If True, ToolTip is a Unicode string
    UI32   ToolTipLen;    // String length in characters
};
```

The function retrieves the most important properties of a field. This function is marked as obsolete, please use GetFieldEx() instead. The parameter *AHandle* must be a valid field handle. The parameter *Handle* must be a valid field handle. Field handles are simple array indexes into the field array of the global AcroForm object. To enumerate all fields of a document execute the function in a loop from 0 though GetFieldCount() -1.

Field colors are returned in the native color space of the field.

Since Acrobat 6 field names can be defined as Ansi or Unicode strings. To determine whether the string is in Ansi or Unicode format compare the string length with the member FieldNameLen as follows (**C/C++**):

```
TPDFField f;
if (!pdfGetField(pdf, aField, &f)) return 1;
if (f.FieldName && strlen(f.FieldName) != f.FieldNameLen)
{
    // We have an Unicode name!
    UI16* nameW = (UI16*)f.FieldName;
}
```

Delphi:

The Delphi interface contains a special function to convert the returned Ansi string back to Unicode:

```
if not pdf.GetField(aField, f) then Exit;
if f.FielNameLen > 0 and StrLen(f.FieldName) <> f.FieldNameLen then begin
    // Use the function ToUnicode of the class TPDF to convert the string
    // back to Unicode
    nameW := pdf.ToUnicode(f.FieldName, f.FieldNameLen * 2);
end;
```

There is no need to check the string format in Visual Basic, VB .Net or C# because a maybe required conversion is automatically applied.

Remarks:

It is also possible to enumerate the fields on a per page basis with GetPageField(). The export values of combo and list boxes can be accessed with GetFieldExpValueEx().

Return values:

If the function succeeds the return value 1 and the structure *Field* is filled with values. If the function fails the return value is 0.

GetFieldBackColor

Syntax:

```
UI32 pdfGetFieldBackColor(
    const PPDF* IPDF, // Instance pointer
```

The function returns the default background color used for newly created interactive form fields. Note that color values must be defined in the current color space. See SetColorSpace() for further information.

If the return value is equal NO_COLOR the background is transparent.

```
#define NO_COLOR 0xFFFFFFF1
Default value = NO_COLOR // Transparent
```

GetFieldBorderColor

Syntax:

```
UI32 pdfGetFieldBorderColor(
    const PPDF* IPDF) // Instance pointer
```

The function returns the default border color used for newly created interactive form fields. Note that color values must be defined in the current color space. See SetColorSpace() for further information.

If the return value is equal NO_COLOR the border is transparent.

```
#define NO_COLOR 0xFFFFFFF1
Default value = 0 // Black
```

GetFieldBorderStyle

Syntax:

```
SI32 pdfGetBorderStyle(
    const PPDF* IPDF, // Instance pointer
    UI32 AField)      // Field handle
typedef enum
{
    bsSolid       = 0, // Solid border
    bsBevelled    = 1, // Bevelled border
    bsInset       = 2, // Inset border
    bsUnderline   = 3, // Underline only
    bsDashed      = 4, // Dashed border
    bsUserDefined = 5  // Occurs if a field uses an undefined value
}TBorderStyle;
```

The function returns the border style of a specific form field. The parameter *AField* must be a valid field handle. It is also possible to change the style of a specific field, see SetFieldBorderStyle() for further information.

Return values:

If the function succeeds the return value is the border style, a value greater or equal zero. If the function fails the return value is a negative error code.

GetFieldBorderWidth

Syntax:

```
double pdfGetFieldBorderWidth(
    const PPDF* IPDF, // Instance pointer
    UI32 AField)      // Field handle
```

The function returns the line width of the border of a field. The parameter *AField* must be a valid field handle. All field types are supported by this function.

Return values:

If the function succeeds the return value is the line width of the field's border (a value greater or equal zero). If the function fails the return value is a negative error code.

GetFieldCalcOrder

Syntax:

```
SI32 pdfGetFieldCalcOrder(
    const PPDF* IPDF, // Instance pointer
    UI32** Out)       // Required -> address of a UI32* variable
```

The function retrieves the calculation order of form fields which contain an action in the *OnCalc* event (see AddActionToObj() for further information). This is the order in which their values will be recalculated when the value of any field changes. The retrieved array values are field handles.

The parameter *Out* must be the address of a UI32* variable. The function stores the array in an internal Unicode buffer. To release the buffer when no longer needed call FreeUniBuf(). The buffer is automatically released when CloseFile() or FreePDF() is called the next time.

To change the calculation order of a specific field call SetFieldCalcOrder().

Example (C++):

```
...
// Assume we have a form in which the fields "total" and "subtotal" are
// calculated in the wrong order. The field subtotal should be calculated
// before the field total.
SI32    fieldTotal = pdfFindField(PDF, "total");
SI32 fieldSubTotal = pdfFindField(PDF, "subtotal");
if (fieldTotal >= 0 && fieldSubTotal >= 0)
{
    UI32* calcOrder;
    SI32 i, count, found = 0, fieldTotalIndex, fieldSubTotalIndex;
    if ((count = pdfGetFieldCalcOrder(PDF, &calcOrder)) >= 0)
    {
        for (i = 0; i < count; i++)
```

```
        {
            if (calcOrder[i] == fieldTotal)
            {
                found |= 1;
                fieldTotalIndex = i;
                if (found == 3) break; // Both fields found?
            }else if (calcOrder[i] == fieldSubTotal)
            {
                found |= 2;
                fieldSubTotalIndex = i;
                if (found == 3) break; // Both fields found?
            }
        }
        if (found == 3)
        {
            if (fieldTotalIndex < fieldSubTotalIndex)
                pdfSetFieldCalcOrder(PDF, fieldTotalIndex, fieldSubTotalIndex);
        }
        pdfFreeUniBuf(PDF); // Optional but recommended: release the buffer
    }
}
...
```

Return values:

If the function succeeds the return value is greater or equal zero. If the function fails the return value is a negative error code.

GetFieldChoiceValue

Syntax:

```
LBOOL pdfGetFieldChoiceValue(
    const PPDF* IPDF,                 // Instance pointer
    UI32 AField,                      // Field handle
    UI32 ValIndex,                    // Value index
    struct TPDFChoiceValue* Value)    // See below
struct TPDFChoiceValue
{
    UI32  StructSize;  // Must be initialized with sizeof(TPDFChoiceValue)
    char* ExpValueA;   // Export value (optional)
    UI16* ExpValueW;   // Export value (optional)
    UI32  ExpValueLen; // Export value length in characters
    char* ValueA;      // Value (can be an empty string or NULL)
    UI16* ValueW;      // Value (can be an empty string or NULL)
    UI32  ValueLen;    // Value length in characters
    LBOOL Selected;    // If true, the value is selected
};
```

The function retrieves a choice value of a combo or list box. The function can also be used to extract optional export values of radio buttons. The parameter *AField* must be a valid handle of a combo box, list box, or radio button. *ValIndex* is the array index of the choice value to be retrieved.

For list and combo boxes call GetFieldExpValCount() to determine how many choice values are available in the combo or list box.

Note that choice values can be inherited from a parent group field. This is the case if the field is part of a field group. See "Interactive Forms/What is a Field Group?" for further information.

Radio buttons can define export values for their children. Since radio buttons contain these values, they are not returned by GetFieldEx() or GetPageFieldEx(). The export values are stored in the same order as the *Kids* array of the structure TPDFFieldEx.

The member *StructSize* of the structure TPDFChoiceValue must be initialized with sizeof(TPDFChoiceValue) before calling the function because this member is used to identify different versions of the structure. All other members can be left uninitialized.

The returned string values can be in Unicode or Ansi format but only one string format is set at time.

Return values:

If the function succeeds the return value is 1 and the parameter *Value* is filled with values. If the function fails the return value is 0.

GetFieldColor

Syntax:

```
SI32 pdfGetFieldColor(
   const PPDF* IPDF,       // Instance pointer
   UI32 AField,            // Field handle
   TFieldColor ColorType,  // Which color should be returned?
   SI32 ADDR ColorSpace,   // Color space see TPDFColorSpace
   UI32 ADDR Color)        // Color value

typedef enum
{
    fcBackColor   = 0,
    fcBorderColor = 1,
    fcTextColor   = 2
}TFieldColor;

typedef enum
{
    csDeviceRGB  = 0,
    csDeviceCMYK = 1,
    csDeviceGray = 2
}TPDFColorSpace;
```

The function retrieves a specific color of an interactive form field. The parameter *AField* must be a valid field handle. The parameter *ColorSpace* defines the color space in which the color was defined.

Return values:

If the function succeeds the return value is 1 and the parameters *ColorSpace* and *Color* are filled with values. If the function fails the return value is 0.

GetFieldCount

Syntax:

```
SI32 pdfGetFieldCount(
    const PPDF* IPDF) // Instance pointer
```

The function returns the number of fields contained in the document.

GetFieldEx

Syntax:

```
LBOOL pdfGetFieldEx(
    const PPDF* IPDF,           // Instance pointer
    UI32 Handle,                // Field handle
    struct TPDFFieldEx* Field)  // Structure to be filled

struct TPDFFieldEx
{
    UI32            StructSize;    // Must be set to sizeof(TPDFFieldEx)
    LBOOL           Deleted;       // Marked as deleted?
    struct TPDFRect BBox;          // Bounding box in bottom-up coordinates
    TFieldType      FieldType;     // Field type
    TFieldType      GroupType;     // See description
    UI32            Handle;        // Field handle
    UI32            BackColor;     // Background color
    TExtColorSpace  BackColorSP;   // Color space of the background color
    UI32            BorderColor;   // Border color
    TExtColorSpace  BorderColorSP; // Color space of the border color
    TBorderStyle    BorderStyle;   // Border style
    float           BorderWidth;   // Border width
    float           CharSpacing;   // Text fields only
    LBOOL           Checked;       // Check boxes only
    UI32            CheckBoxChar;  // ZapfDingbats character index
    TCheckBoxState  DefState;      // Check boxes only
    const char*     DefValueA;     // Optional default value
    const UI16*     DefValueW;     // Optional default value
    const void*     IEditFont;     // Pointer to default editing font
    const char*     EditFont;      // Postscript name of the editing font
    UI32            ExpValCount;   // Combo and list boxes only.
    const char*     ExpValueA;     // Check boxes only
    const UI16*     ExpValueW;     // Check boxes only
    TFieldFlags     FieldFlags;    // Field flags
    const void*     IFieldFont;    // Pointer to the field font.
    const char*     FieldFont;     // Postscript name of the font
    UI32            Reserved;      // Reserved field
    double          FontSize;      // Font size. 0.0 = auto font size
    const char*     FieldNameA;    // Field name (can be NULL)
    const UI16*     FieldNameW;    // Field name (can be NULL)
    THighlightMode  HighlightMode; // Highlight mode
    LBOOL           IsCalcField;   // If true, the OnCalc event is
                                   // connected with a JavaScript action
    const char*     MapNameA;      // Optional unique mapping name
    const UI16*     MapNameW;      // Optional unique mapping name
    UI32            MaxLen;        // Text fields -> zero = not restricted
    const void**    Kids;          // Array of children -> GetFieldEx2()
    UI32            KidCount;      // Number of fields in the array
    void*           Parent;        // Pointer to parent field or NULL
    SI32            PageNum;       // Page on which the field is used or -1
    SI32            Rotate;        // Rotation angle in degrees
    TTextAlign      TextAlign;     // Text fields only
    UI32            TextColor;     // Text color
    TExtColorSpace  TextColorSP;   // Color space of the field's text
```

```
    float           TextScaling;    // Text fields only
    const char*     ToolTipA;       // Optional tool tip
    const UI16*     ToolTipW;       // Optional tool tip
    const char*     UniqueNameA;    // Optional unique name (NM key)
    const UI16*     UniqueNameW;    // Optional unique name (NM key)
    const char*     ValueA;         // Field value
    const UI16*     ValueW;         // Field value
    float           WordSpacing;    // Text fields only
    const void*     IBarcode;       // If present, this field is a barcode
                                    // field. The field type is set to
                                    // ftText since barcode fields are
                                    // extended text fields.
                                    // GetBarcodeDict() returns the
                                    // properties of the barcode.
    const void*     ISignature;     // Present only for imported signature
                                    // fields which which have a value. That
                                    // means the file was digitally signed.
                                    // GetSigDict() returns the signature
                                    // dictionary. Signed signature fields
                                    // are always marked as deleted!
    const char*     ModDate;        // Last modification date (optional)

    // Push buttons only. The down and roll over states are optional. If
    // not present, then all states use the up state.
    // The handles of the up, down, and roll over states are template
    // handles! The templates can be opened for editing with
    // EditTemplate2() and parsed with ParseContent().
    TBtnCaptionPos  CaptionPos;     // Where to position the caption
    const char*     DownCaptionA;   // Caption of the down state
    const UI16*     DownCaptionW;   // Caption of the down state
    SI32            DownImage;      // Template handle of the down state
    const char*     RollCaptionA;   // Caption of the roll over state
    const UI16*     RollCaptionW;   // Caption of the roll over state
    SI32            RollImage;      // Templ. handle of the roll over state
    const char*     UpCaptionA;     // Caption of the up state
    const UI16*     UpCaptionW;     // Caption of the up state
    SI32            UpImage;        // Template handle of the up state
    SI32            OC;             // Handle of an OCG or OCMD or -1
    SI32            Action;         // Action handle or -1 if not set. This action is executed when
                                    // the field is activated.
    TActionType     ActionType;     // Meaningful only, if Action >= 0.
    IEVT*           Events;         // See GetObjEvent() if set.
    UI32            FmtTextColor;   // Zero if not set. Usually RGB red otherwise.
    const char*     FmtTextA;       // Text fields only. The formatted field value if the field
                                    // contains an OnFormat Javascript action.
    const UI16*     FmtTextW;       // Note that either the Ansi or Unicode value is set.
};

typedef enum
{
    bcpCaptionOnly,  // Default
    bcpImageOnly,    // No caption; image only
    bcpCaptionBelow, // Caption below the image
    bcpCaptionAbove, // Caption above the image
    bcpCaptionRight, // Caption on the right of the image
    bcpCaptionLeft,  // Caption on the left of the image
    bcpCaptionOver   // Caption overlaid directly on the image
}TBtnCaptionPos;
```

The function returns the most important properties of a field. The parameter *Handle* must be a valid field handle. Field handles are simple array indexes into the field array of the global AcroForm object. To enumerate all fields of a document execute the function in a loop from 0 though GetFieldCount() -1.

The member *StructSize* must be initialized to sizeof(TPDFFieldEx) before calling the function.

The member *GroupType* specifies the subtype of a field group. A field group is a set of fields which have the same name and type. Such fields contain always the same value with exception of check boxes. If a field group consists of check boxes the export value is also taken into account.

However, a set of fields with identical names are internally stored in a group field. The group type specifies the field type of the group, e.g. ftText if text fields are stored in the group. A field group cannot contain different field types. An indication whether a field is a child of a field group is an empty field name because the field name is taken from the parent group field. Radio buttons are identically organized, but the field type is set to ftRadioBtn in this case.

The page number is not set for group fields and radio button fields because these field types are not directly used on a page. The children of such fields appear in a page but not the corresponding group fields. This behaviour must be taken into account when enumerating fields with GetPageFieldEx() since these field types do never occur when enumerating fields on a per page basis. However, the parent handle is always set if a field is a child of another field.

Inherited attributes of a field group are automatically set to the children of the group. So, there is no need to merge the values of the parent group field with the ones of its children.

The background and border color can be set to the special value NO_COLOR which means the background or border is transparent. The color space is set to DeviceRGB in this case.

String values can be either in Unicode or Ansi format but only one format is set at time.

The parent field and the children of a field (member *Kids*) can be accessed with GetFieldEx2().

Radio Buttons

Radio buttons can define export values for their check boxes. Because the radio button holds these values, they are not returned when calling GetFieldEx() for a check box of a radio button. These export values can be accessed with GetFieldChoiceValue(). The export values are stored in the same order as the *Kids* array. This makes it easy to access the values, just call GetFieldChoiceValue() right after GetFieldEx2() for every child. The parameter *AField* must be set to the handle of the radio button and *Index* to the array index.

If a radio button contains no export values, GetFieldChoiceValue() returns just the export value of the check box. So, there is no need to further validate the result of the function.

Formatted Text Field Value

The member *FmtTextColor*, *FmtTextA* or *FmtTextW* are set for text fields only. If set, this value represents the formatted text that was created by a Javascript action in the field's *OnFormat* event.

At time of publication the Javascript functions *AFNumber_Format()* and *AFSpecial_Format()* are supported only.

The value of a text field that uses a number format should contain a pure number without any formatting characters, e.g. "256.87". The decimal separator, if necessary, should be a period (.) and no comma.

Because all variants of number and date formats are applied via Javascript actions, there is no easy way to determine what kind of values a text field accepts.

Optional Content

Form fields can be part of an Optional Content Group (OCG) or Optional Content Membership Dictionary (OCMD). If *OC* is greater -1 an OCG or OCMD handle was set. To determine whether an OCG handle was set, check whether the value is smaller 0x40000000:

```
bool IsOCG(SI32 Handle)
{
    return (Handle > -1 && Handle < 0x40000000);
}
```

Return vlues:

If the function succeeds the return value is 1 and the structure *Field* is filled with values. If the function fails the return value is 0.

GetFieldEx2

Syntax:

```
LBOOL pdfGetFieldEx2(
  const PPDF* IField,         // Pointer of a field object
  struct TPDFFieldEx* Field) // Structure to be filled
```

The function returns the most important properties of a field like GetFieldEx() but it accepts a field pointer as input. The parameter *IField* must be a valid pointer of a field object that was returned by GetFieldEx(). See also GetFieldEx(). This function can be used to access the parent field or children of a field (member *Kids*).

Remarks:

The function assumes that the parameters *IField* and *Field* are valid. Passing a null pointer to the function causes an access violation!

Return values:

If the function succeeds the return value is 1 and the structure *Field* is filled with values. If the function fails the return value is 0. The only reason why this function can fail is if the member StructSize was initialized to an invalid value.

GetFieldExpValCount

Syntax:

```
SI32 pdfGetFieldExpValCount(
   const PPDF* IPDF, // Instance pointer
   UI32 AField)      // Field handle
```

The function returns the number of values/export values which are defined for a field. Supported field types are combo boxes, list boxes, radio buttons, and check boxes. If the field is a radio button, the function returns the number of check boxes which are connected with the radio button. This number can be used to access the check boxes of the radio button either with GetFieldExpValueEx() or SetFieldExpValue(). If the field is a check box, the return value is always 1 because check boxes do not support multiple export values. However, this function is normally used to determine the number of available choice values within a combo box or list box because only these field types support multiple choice values. See also GetFieldExpValueEx().

Return values:

If the function succeeds the return value is the number of values/export values which are defined for a field. If the function fails the return value is a negative error code.

GetFieldExpValue

Syntax:

```
SI32 pdfGetFieldExpValue(
   const PPDF* IPDF,  // Instance pointer
   UI32 AField,       // Field handle
   char* ADDR Value)  // Field's export value
```

This function retrieves the export value(s) of a check box, list box, or combo box. The parameter *AField* must be a valid field handle of a check box, list box or combo box. The parameter *Value* holds a pointer to the original export value. The retrieved string value must not be changed or freed in your application.

The export value of a check box can be either "Off" if the check box is not selected or an arbitrary string when the check box is selected.

The export value of a combo box or list box can be NULL when no value is selected.

Since Acrobat 5, list boxes support multiple selected values. In this case the export values are delimited by a null-character (0); the last value is terminated with two null-characters. Note that the returned string length is the length without the last two null-terminators.

The export values of a list box that has three selected values are returned as follows:

"Value1\0Value2\0Value3\0\0"

The returned string length would be 20 characters in this example.

To extract the values proceed as follows:

```
...
SI32 valLen;
char* src, *value = NULL;
// aField must be a handle of list box in this example
valLen = pdfGetFieldExpValue(pdf, aField, value);
SI32 len = strlen(value); // Get the string length
if (valLen > 0 && len < valLen)
{
   src = value;
   do
   {
      fwrite(src, 1, len, f);
      fwrite("\r\n", 1, 2, f);
      src += (len +1);    // Skip over the null-terminator
      // This can not cause a buffer overrun because of the two
      // null-terminators at the end of the string.
      len = strlen(src);
   }while(len);
}

...
```

If the string length that was returned by the standard C function strlen() is shorter than the length that was returned by GetFieldExpValue(), multiple values are selected in the list box.

The code above writes the values to a file and adds an EOL after each value. This is just an example to see how the string can be parsed.

Notice:

The export values of check boxes and combo boxes are **not** terminated with two null-terminators. Try to parse the export value only if the field is a list box. The field type can be determined with the functions GetFieldType(), GetFieldEx() or GetPageFieldEx().

Return values:

If the function succeeds the return value is the string length excluding the last null-terminator(s). If the function fails the return value is a negative error code.

GetFieldExpValueEx

Syntax:

```
SI32 pdfGetFieldExpValueEx(
   const PPDF* IPDF,     // Instance pointer
   UI32 AField,          // Field handle
   UI32 ValIndex,        // Value index
   char* ADDR Value,     // Index value
   char* ADDR ExpValue,  // Index export value
   LBOOL ADDR Selected)  // Is the value selected?
```

This function can be used to enumerate the choice values of a combo box, list box, or radio button. If the field is a check box, the parameter *ValIndex* will be ignored and the export value and selected state are returned. The parameters *Value* and *ExpValue* hold pointers to the original null-terminated string values. These values must not be changed or freed in your application. Note that either the parameter *Value* or *ExpValue* can be set to NULL depending on the field type.

To enumerate the choice values of a combo box or list box call GetFieldExpValCount() to determine the number of available values. The parameter *ValIndex* can then be used to access each value / export value pair within the array.

Beginning with PDF 1.5 choice values of combo and list boxes can be in Unicode format. Because Unicode strings are not supported by this function the function GetFieldChoiceValue() should be used to enumerate choice values of these field types.

Example (C++):

```cpp
...
// aField is a handle of a combo box in this example; we want to
// deselect the currently selected value, that's all.
char* value, *expValue;
LBOOL selected;
SI32 valCount = pdfGetFieldExpValCount(pdf, aField);
for (SI32 i = 0; i < valCount; i++)
{
   if (pdfGetFieldExpValueEx(pdf, aField, i, value, expValue, selected))
   {
      if (selected)
      {
         pdfSetFieldExpValue(pdf, aField, i, value, expValue, false);
         break;
      }
   }
}
...
```

Remarks:

The parameters *Value*, *ExpValue*, and *Selected* require variables so that the values can be set by the function. To change a choice value of a combo box or list box call the function SetFieldExpValue().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetFieldFlags

Syntax:

```
SI32 pdfGetFieldFlags(
    const PPDF* IPDF, // Instance pointer
    UI32 AField)      // Field handle
```

```
typedef UI32 TFieldFlags;
// Basic flags supported by all field types except group fields and
// radio button fields.
#define ffReadOnly        0x00000001
#define ffRequired        0x00000002
#define ffNoExport        0x00000004
#define ffInvisible       0x00000008
#define ffHidden          0x00000010
#define ffPrint           0x00000020
#define ffNoZoom          0x00000040
#define ffNoRotate        0x00000080
#define ffNoView          0x00000100
// Special flags supported by specific fields only
#define ffMultiline       0x00001000
#define ffPassword        0x00002000
#define ffNoToggleToOff   0x00004000
#define ffRadioIsUnion    0x04000000
#define ffCommitOnSelCh   0x08000000
#define ffEdit            0x00040000
#define ffSorted          0x00080000
#define ffFileSelect      0x00100000
#define ffMultiSelect     0x00200000
#define ffDoNotSpellCheck 0x00400000
#define ffDoNotScroll     0x00800000
#define ffComb            0x01000000
```

The function returns the flags of a specific interactive form field. The parameter *AField* must be a valid handle of an interactive form field.

Interactive form fields and annotations support the same basic set of field flags because both object types are in fact annotations. When we talk about form fields then we talk about an extended version of annotations because form fields are annotations. However, to make the handling easier, form fields and annotations are handled separately by DynaPDF.

The flags are described in detail on the next page.

Return values:

If the function succeeds the return value is a number greater or equal zero representing the flags used by the field (this is a bit mask, to check whether a specific flag is set, use the bitwise and operator). If the function fails the return value is a negative error code.

Flag	Description
ffInvisible	If set, do not display the field if it does not belong to one of the standard field types and no field handler is available.
ffHidden	*(PDF 1.2)* If set, do not display or print the field or allow it to interact with the user, regardless of its field type or whether an field handler is available.
ffPrint	*(PDF 1.2)* If set, print the field when the page is printed. If clear, never print the field, regardless of whether it is displayed on the screen. This can be useful, for example, for fields representing interactive pushbuttons, which would serve no meaningful purpose on the printed page.
ffNoZoom	*(PDF 1.3)* If set, do not scale the field's appearance to match the magnification of the page. The location of the field on the page (defined by the upper-left corner of its field rectangle) remains fixed, regardless of the page magnification.
ffNoRotate	*(PDF 1.3)* If set, do not rotate the field's appearance to match the rotation of the page. The upper-left corner of the field rectangle remains in a fixed location on the page, regardless of the page rotation.
ffNoView	*(PDF 1.3)* If set, do not display the field on the screen or allow it to interact with the user. The field may be printed (depending on the setting of the ffPrint flag), but should be considered hidden for purposes of on-screen display and user interaction.
ffReadOnly	*(PDF 1.3)* If set, do not allow the field to interact with the user. The field may be displayed or printed (depending on the settings of the afNoView and ffPrint flags), but should not respond to mouse clicks or change its appearance in response to mouse motions.
ffRequired	If set, the field must have a value at the time it is exported by a submit-form action (see CreateSubmitAction() for further information). Supported by all fields except group fields.

Flag	Description
ffNoExport	If set, the field must not be exported by a submit-form action (see CtreateSubmitAction() for further information). Supported by all fields except group fields.
ffMultiline	If set, the field may contain multiple lines of text; if clear, the field's text is restricted to a single line. Supported by button fields, text fields.
ffPassword	If set, the field is intended for entering a secure password that should not be echoed visibly to the screen. Characters typed from the keyboard should instead be echoed in some unreadable form, such as asterisks or bullet characters. To protect password confidentiality, the value of the text field is not stored in the PDF file if this flag is set. Supported by text fields only.
ffNoToggleToOff	If set, exactly one radio button must be selected at all times; clicking the currently selected button has no effect. If clear, clicking the selected button deselects it, leaving no button selected. Supported by radio button fields only.
ffEdit	If set, the combo box includes an editable text box as well as a drop list; if clear, it includes only a drop list. Supported by combo boxes only.
ffSorted	If set, the field values are sorted in ascending order. Supported by combo boxes and list boxes only.
ffFileSelect	*(PDF 1.4)* If set, the text entered in the field represents the pathname of a file whose contents are to be submitted as the value of the field. Supported by text fields only.
ffMultiSelect	*(PDF 1.4)* If set, more than one of the field 's option items may be selected simultaneously; if clear, no more than one item at a time may be selected. This flag is supported by list boxes only.
ffDoNotSpellCheck	*(PDF 1.4)* If set, the text entered to the field will not be spell-checked. Supported by text fields, combo boxes. If the field type is combo box, this flag is meaningful only if the flag ffEdit is also set.

Flag	Description
ffDoNotScroll	*(PDF 1.4)* If set, the field will not scroll (horizontally for single-line fields, vertically for multiple-line fields) to accommodate more text than will fit within its field rectangle. Once the field is full, no further text will be accepted. Supported by text fields only.
ffComb	*(PDF 1.5)* Meaningful only if *MaxLen* is set (see CreateTextField()) and if the ffMultiline, ffPassword, and ffFileSelect flags are clear. If set, the field is automatically divided up into as many equally spaced positions, or combs, as the value of *MaxLen*, and the text is laid out into those combs. Supported by text fields only.
ffCommitOnSelCh	*(PDF 1.5)* If set, the new value is committed as soon as a selection is made with the pointing device. This allows applications to perform an action once a selection is made, without requiring the user to exit the field. If clear, the new value is not committed until the user exits the field. Supported by combo boxes and list boxes only.
ffRadioIsUnion	*(PDF 1.5)* If set, a group of radio buttons within a radio button field that use the same export value for the on state will turn on and off in unison; that is if one is checked, they are all checked. This flag requires Acrobat 6 or higher. Supported by check boxes only.

GetFieldGroupType

Syntax:

```
SI32 pdfGetFieldGroupType(
    const PPDF* IPDF, // Instance pointer
    UI32 AField)      // Field handle

typedef enum
{
    ftButton    = 0,
    ftCheckBox  = 1,
    ftRadioBtn  = 2,
    ftComboBox  = 3,
    ftListBox   = 4,
    ftText      = 5,
    ftSignature = 6,
    ftGroup     = 7
}TFieldType;
```

The function returns the base type of a field group. A field group is a set of fields which have all the same name. Such fields contain always the same value with exception of checkboxes. If a field group consists of checkboxes the export value is also taken into account.

However, a set of fields with identical names are internally stored in a group field. The group type specifies the field type of the group, e.g. ftText if text fields are stored in the group. A field group cannot contain different field types. An indication whether a field is a child of a field group is an empty field name because the field name is taken from the parent group field in this case. Radio buttons are identically organized, but the field type is ftRadioBtn in this case.

Return values:

If the function succeeds the return value is the field group type that is a value greater or equal zero. Make a type cast to TFieldType in this case. If the function fails the return value is a negative error code.

GetFieldHighlightMode

Syntax:

```
SI32 pdfGetFieldHighlightMode(
    const PPDF* IPDF, // Instance pointer
    UI32 AField)      // Field handle


typedef enum
{
    hmNone     = 0,
    hmInvert   = 1,
    hmOutline  = 2,
    hmPush     = 3,
    hmPushUpd  = 4  // Update appearance stream on changes
}THighlightMode;
```

The function returns the highlight mode of buttons, checkboxes, and signature fields. All other field types have no property highlight mode, the function returns hmNone in this case.

Return values:

If the function succeeds the return value is the highlight mode, a value greater or equal zero. Make a type cast to THighlightMode in this case. If the function fails the return value is a negative error code.

GetFieldIndex

Syntax:

```
SI32 pdfGetFieldIndex(
    const PPDF* IPDF, // Instance pointer
    UI32 AField)      // Field handle
```

The function returns the page index or tab order of the field. The index of newly created fields starts at 1000. This makes it easier to set a field to another position inside the tab order. The field index can be changed with the function SetFieldIndex(). Note that fields must be sorted by index with the function SortFieldsByIndex(). Fields must be sorted for each page separately.

Return values:

If the function succeeds the return value is the field index, a value greater or equal zero. If the function fails the return value is a negative error code.

GetFieldMapName

Syntax:

```
SI32 pdfGetFieldMapName(
    const PPDF* IPDF,    // Instance pointer
    UI32 AField,         // Field handle
    void* ADDR Value,    // Pointer to mapping name (null-terminated)
    LBOOL ADDR Unicode)  // If true, Value is an Unicode string
```

The function retrieves the mapping name of a specific form field if set. The parameter *AField* must be a valid field handle. The parameter *Value* gets a pointer to the original string value, it must not be NULL. Do not change the or free the value. Depending on whether *Unicode* is true, *Value* is pointer to a null-terminated Unicode string.

The mapping name is used when interactive form field data is exported from the document.

Return values:

If the function succeeds the return value is the string length in characters and the parameters *Value* and *Unicode* are filled with values. If the function fails the return value is a negative error code.

GetFieldName

Syntax:

```
SI32 pdfGetFieldName(
    const PPDF* IPDF, // Instance pointer
    UI32 AField,      // Field handle
    char* ADDR Name)  // Pointer to field name (null-terminated)
```

The function retrieves the name of a specific interactive form field. The parameter *AField* must be a valid field handle. Note that the check boxes of a radio button field do not have a name.

The parameter *Name* gets a pointer to the original field name, it must not be NULL. The field name must not be changed or freed by the application.

Since Acrobat 6 field names can be defined as Ansi or Unicode strings. To determine whether the string is in Ansi or Unicode format compare the string length with the return value as follows (C/C++):

```
char* fieldName = NULL;
SI32 fieldNameLen = pdfGetFieldName(pdf, field, fieldName);
if (fieldNameLen > 0 strlen(fieldName) != fieldNameLen)
{
    // We have an Unicode name!
    UI16* nameW = (UI16*)fieldName;
}
```

Delphi:

```
nameLen := pdf.GetFieldName(myField, name);
if nameLen > 0 then begin
    if Length(name) <> nameLen then begin
        // Use the function ToUnicode to convert the string back to Unicode
        nameW := pdf.ToUnicode(name);
        ...
    end else begin
        ...
    end;
end;
```

There is no need to check the string format in Visual Basic, VB .Net or C# because a maybe required conversion is automatically applied.

Return values:

If the function succeeds the return value is the string length of the name in characters excluding the null-terminator. If the function fails the return value is a negative error code.

GetFieldOrientation

Syntax:

```
SI32 pdfGetFieldOrientation(
   const PPDF* IPDF, // Instance pointer
   UI32 AField)       // Field handle
```

The function returns the orientation of a field. The orientation is measured in degrees; it is always a multiple of 90 or 0.

Return values:

If the function succeeds the return value is the field orientation in degrees. If the function fails the return value is a negative error code, a value smaller -270.

GetFieldTextAlign

Syntax:

```
SI32 pdfGetFieldTextAlign(
   const PPDF* IPDF, // Instance pointer
   UI32 AField)       // Text field or button field handle
typedef enum
{
   taLeft,
   taCenter,
   taRight,
   taJustify
}TTextAlign;
```

The function returns the text alignment of a text field or button field.

Return values:

If the function the function succeeds the return value is the text alignment used by the field. If the function fails the return value is a negative error code.

GetFieldTextColor

Syntax:

```
UI32 pdfGetFieldTextColor(
   const PPDF* IPDF) // Instance pointer
```

The function returns the default text color used for newly created fields.

GetFieldToolTip

Syntax:

```
SI32 pdfGetFieldToolTip(
   const PPDF* IPDF,    // Instance pointer
   UI32 AField,         // Field handle
   void* ADDR Value,    // Pointer to value (null-terminated string)
   LBOOL ADDR Unicode)  // If true, Value is an Unicode string
```

The function retrieves a pointer to the tool tip string of a specific interactive form field. The parameter *AField* must be a valid field handle. The parameter *Value* gets a pointer to the original string value, it must not be NULL. Do not change the or free the value. Depending on whether *Unicode* is true, Value is pointer to a null-terminated *Unicode* string.

Return values:

If the function succeeds the return value is 1 and the parameters *Value* and *Unicode* are filled with values. If the function fails the return value is 0.

GetFieldType

Syntax:

```
SI32 pdfGetFieldType(
   const PPDF* IPDF, // Instance pointer
   UI32 AField)      // Field handle
typedef enum
{
    ftButton    = 0,
    ftCheckBox  = 1,
    ftRadioBtn  = 2,
    ftComboBox  = 3,
    ftListBox   = 4,
    ftText      = 5,
    ftSignature = 6,
    ftGroup     = 7
}TFieldType;
```

The function returns the field type. The parameter *AField* must be a valid field handle.

Return values:

If the function succeeds the return value is the field type (make a typecast to TFieldType to determine the field type). If the function fails the return value is a negative error code.

GetFileSpec

Syntax:

```
LBOOL pdfGetFileSpec(
    const IFSR* IFS,           // Pointer of a file specification
    struct TPDFFileSpecEx* F) // see below
struct TPDFFileSpecEx
{
    UI32        StructSize;     // Must be set to sizeof(TPDFFileSpecEx).
    const char* AFRelationship; // PDF 2.0
    const void* ColItem;        // If != NULL the embedded file contains a collection item
                                // with user defined data. This entry can
                                // occur in PDF Collections only (PDF 1.7). See chapter
                                // "PDF Collections" for further information.
    const char* DescriptionA;   // Optional description string.
    const UI16* DescriptionW;   // Optional description string.
    const char* DOS;            // Optional DOS file name.
    IEFN*       EmbFileNode;    // -> GetEmbeddedFileNode().
    const char* FileName;       // File name as 7 bit ASCII string.
    LBOOL       FileNameIsURL;  // If true, FileName contains a URL.
    const BYTE* ID1;            // Optional file ID. Meaningful only if FileName points to
                                // a PDF file. This is a binary string.
    UI32        ID1Len;         // String length in bytes.
    const BYTE* ID2;            // The file was modified if ID1 and ID2 are different.
    UI32        ID2Len;         // String length in bytes.
    LBOOL       IsVolatile;     // If true, the file changes frequently with time.
    const char* Mac;            // Optional Mac file name.
    const char* Unix;           // Optional Unix file name.
    IRFN*       RelFileNode;    // Optional related files array. -> GetRelFileNode().
    PIMG*       Thumb;          // Optional thumb nail image. -> GetImageObjEx().
    const char* UFileNameA;     // PDF 1.7. Same as FileName but Unicode is allowed.
    const UI16* UFileNameW;     // Either the Ansi or Unicode string is set but never both.
};
```

The function retrieves the properties of a file specification. The member *StructSize* must be set to sizeof(TPDFFileSpecEx) before the function can be executed.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetFillColor

Syntax:

```
UI32 pdfGetFillColor(
    const PPDF* IPDF) // Instance pointer
```

The function returns the current color used for fillings. The returned color value is not converted to the active color space. For example, if the color space will be changed with SetColorSpace(), the color value is still the same. Colors must be correctly defined in the current color space.

```
Default value = 0 (black)
```

GetFont (obsolete)

Syntax:

```
LBOOL fntGetFont(
    const void* IFont,      // Pointer of a font object
    struct TPDFFontObj* F) // Structure to be filled
struct TPDFFontObj
{
    float      Ascent;        // Ascent
    char*      BaseFont;      // PostScript Name or Family Name
    float      CapHeight;     // Cap height
    float      Descent;       // Descent
    UI16*      Encoding;      // Unicode mapping 0..255 if set
    UI32       FirstChar;     // First char
    UI32       Flags;         // Font flags -> font descriptor
    char*      FontFamily;    // Optional Font Family (Family Name)
    LBOOL      FontFamilyUni; // Is FontFamily in Unicode format?
    char*      FontName;      // Font name -> font descriptor
    TFontType  FontType;      // If ftType0 the font is a CID font.
    float      ItalicAngle;   // Italic angle
    UI32       LastChar;      // Last char
    float      SpaceWidth;    // Space width in font units or default value.
    float*     Widths;        // Glyph widths 0..WidthsCount -1
    UI32       WidthsCount;   // Number of widths in the array
    float      XHeight;       // x-height
    float      DefWidth;      // Default character widths
    char*      FontFile;      // File buffer (if imported & embedded) or file path.
    UI32       Length1;       // Clear text portion of Type1 font, or full length.
    UI32       Length2;       // Encrypted portion of a Type1 font program.
    UI32       Length3;       // Length of the fixed-content portion or zero.
    TFontFileSubtype FontFileType; // See below;
typedef enum
{
    ftMMType1  = 0, // Multiple Master
    ftTrueType = 1, // TrueType font
    ftType0    = 2, // CID font
    ftType1    = 3, // Type1 font
    ftType3    = 4  // Type3 font
}TFontType;
typedef enum
{
    ffsType1C       = 0, // CFF based Type1 font
    ffsCIDFontType0C = 1, // CFF based Type1 CID font
    ffsOpenType     = 2, // TrueType based OpenType font
    ffsOpenTypeC    = 3, // CFF based OpenType font
    ffsCIDFontType2 = 4, // TrueType based CID Font
    ffsNoSubtype    = 9  // The font file is in the format of FontType
}TFontFileSubtype;
```

This function is marked as obsolete since an extended version is available now. Please use GetFontInfo() if possible.

The function returns the most important properties of a font. The parameter *IFont* must be a valid pointer to a font object. Such a pointer is returned by GetPageText(), EnumDocFonts(), or by the content parser (see ParseContent()). The parameter *F* must be a valid pointer to a TPDFFontObj structure.

The font metrics such as Ascent, Descent, Widths, and so on are returned in font units. The values must be scaled to the given font size as follows:

```
double scale   = FontSize / 1000.0;
double ascent  = font.Ascent  * scale;
double descent = font.Descent * scale;
```

The font's encoding consists of exactly 256 Unicode values if set. The encoding is not set if a CID font is selected.

The font flags describe certain properties of the font. The most important flags are the following:

- `0x1      // Fixed pitch font`
- `0x2      // Serif style`
- `0x4      // Symbol font`
- `0x8      // Script style`
- `0x20     // Non-symbolic font`
- `0x40     // Italic style`
- `0x40000 // Force Bold (Type1 fonts only)`

The full set of available flags can be found in the PDF Reference.

The member FontFile provides a pointer to the font file buffer if the font is embedded and if it was imported from an external PDF file. In this case and if the font type is **not** ftType1 the member Length1 specifies the length of the font buffer.

Type1 fonts consist of three portions. Length1 specifies the length of the clear text portion, Length2 specifies the length of the eexec encrypted binary portion, and Length3 specifies the length of the fixed content portion (512 zeros plus cleartomark). The latter portion is optional and mostly not set in PDF fonts. The buffer length is the sum of the three lengths.

If the font was loaded from the system the parameter FontFile contains the file path to the font file. The member Length1 is set to zero in this case.

Since DynaPDF 3.0.8.16 the default string format for font file paths on Windows was changed to UTF-16. You can make a type cast to UI16* on Windows, as long as the Ansi version of AddFontSearchPath() will not be used.

Remarks:

The function does not use the exception handling of DynaPDF. No error message is set if the function fails. However, the only possible error is out of memory if the parameters are valid.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetFontCount

Syntax:

```
SI32 pdfGetFontCount(
  const PPDF* IPDF) // Instance pointer
```

The function returns the number of fonts which are used in the document. The properties of a font can be accessed with GetFontEx().

GetFontEx (obsolete)

Syntax:

```
LBOOL pdfGetFontEx(
  const PPDF* IPDF,       // Instance pointer
  UI32 Handle,            // Font handle
  struct TPDFFontObj* F) // Structure to be filled
```

This function is marked as obsolete since an extended version is available now. Please use GetFontInfoEx() if possible.

The function retrieves the most important properties of a font like GetFont() but accepts a font handle instead. A font handle is a simple array index. To enumerate all fonts of a document execute the function in a loop from zero to GetFontCount() - 1. See also GetFont().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetFontInfo

Syntax:

```
LBOOL fntGetFontInfo(
   const void* IFont,      // Pointer of a font object
   struct TPDFFontInfo* F) // Structure to be filled
typedef enum
{
   beWinAnsi   = 0, // This represents the Windows code page 1252.
   beMacRoman  = 1, // Mac Roman encoding.
   beMacExpert = 2, // Mac Expert encoding.
   beStandard  = 3  // Standard encoding is a special encoding for Type1 fonts.
}TBaseEncoding;

typedef enum
{
    ftMMType1  = 0, // Multiple Master
    ftTrueType = 1, // TrueType font
    ftType0    = 2, // CID font
    ftType1    = 3, // Type1 font
    ftType3    = 4  // Type3 font
}TFontType;

typedef enum
{
    ffsType1C        = 0, // CFF based Type1 font
    ffsCIDFontType0C = 1, // CFF based Type1 CID font
```

```
    ffsOpenType       = 2, // TrueType based OpenType font
    ffsOpenTypeC      = 3, // CFF based OpenType font
    ffsCIDFontType2   = 4, // TrueType based CID Font
    ffsNoSubtype      = 9  // The font file is in the format of FontType
}TFontFileSubtype;

struct TCIDMetric
{
   float Width; // Vertical displacment for text selection.
   float x;      // Positioning vector
   float y;      // Positioning vector
};

struct TPDFFontInfo
{
    UI32             StructSize;       // Must be set to sizeof(TPDFFontInfo).
    float            Ascent;           // Ascent (optional).
    float            AvgWidth;         // Average character width (optional).
    TBaseEncoding    BaseEncoding;     // Valid only if HaveEncoding is true.
    char*            BaseFont;         // PostScript Name or Family Name.
    float            CapHeight;        // Cap height (optional).
    char*            CharSet;          // The charset describes which glyphs are present in the font.
    UI32             CharSetSize;      // Length of the CharSet in bytes.
    char*            CIDOrdering;      // SystemInfo -> Character collection.
    char*            CIDRegistry;      // SystemInfo -> Issuer of the character collection.
    BYTE*            CIDSet;           // CID fonts only. This is a table of bits indexed by CIDs.
    UI32             CIDSetSize;       // Length of the CIDSet in bytes.
    UI32             CIDSupplement;    // CIDSystemInfo -> Supplement number.
    BYTE*            CIDToGIDMap;      // Allowed for embedded TrueType based CID fonts only.
    UI32             CIDToGIDMapSize;  // Length of the stream in bytes.
    BYTE*            CMapBuf;          // Only available if the CMap was embedded.
    UI32             CMapBufSize;      // Buffer size in bytes.
    char*            CMapName;         // CID fonts only external CMap name or encoding.
    float            Descent;          // Descent (optional).
    UI16*            Encoding;         // Unicode mapping 0..255 -> not available for CID fonts.
    UI32             FirstChar;        // First char (simple fonts only).
    UI32             Flags;            // See description below.
    TBBox            FontBBox;         // This is the size of the largest glyph in this font.
    BYTE*            FontBuffer;       // Available if the font was embedded or loaded from a buffer.
    UI32             FontBufSize;      // Font file size in bytes.
    char*            FontFamilyA;      // Optional Font Family (Family Name).
    UI16*            FontFamilyW;      // Optional Font Family (Family Name).
    char*            FontFilePathA;    // Only available for system fonts.
    UI16*            FontFilePathW;    // Either the Ansi or Unicode path is set, but never both.
    TFontFileSubtype FontFileType;     // See description below.
    char*            FontName;         // Font name (should be the same as BaseFont).
    char*            FontStretch;      // Optional -> Condensed, and so on.
    TFontType        FontType;         // If ftType0 the font is a CID font.
    float            FontWeight;       // Font weight (optional).
    char*            FullNameA;        // System fonts only.
    UI16*            FullNameW;        // System fonts only.
    LBOOL            HaveEncoding;     // If true, BaseEncoding was set from the font's encoding.
    float*           HorzWidths;       // Horizontal glyph widths -> 0..HorzWidthsCount -1.
    UI32             HorzWidthsCount;  // Number of horizontal widths in the array.
    LBOOL            Imported;         // If true, the font was imported from an external PDF file.
    float            ItalicAngle;      // Italic angle.
    char*            Lang;             // Optional language code defined by BCP 47.
    UI32             LastChar;         // Last char (simple fonts only).
    float            Leading;          // Leading (optional).
    UI32             Length1;          // Length of the clear text portion of a Type1 font.
    UI32             Length2;          // Length of the encrypted portion of a Type1 font program.
    UI32             Length3;          // Length of the fixed-content portion of a Type1 font rogram.
    float            MaxWidth;         // Maximum glyph width (optional).
    BYTE*            Metadata;         // Optional XMP metadata stream about the font file.
    UI32             MetadataSize;     // Buffer size in bytes.
    float            MisWidth;         // Missing width (default = 0.0).
    BYTE*            Panose;           // CID fonts only -> Optional 12 bytes long Panose string.
    char*            PostScriptNameA;  // System fonts only.
    UI16*            PostscriptNameW;  // System fonts only.
```

```
    float             SpaceWidth;        // Space width in font units. A default value is set if the
                                         // font contains no space character.
    float             StemH;             // The thickness of horizontal stems.
    float             StemV;             // The thickness of vertical stems.
    BYTE*             ToUnicode;         // ToUnicode CMap. Only available for imported fonts.
    UI32              ToUnicodeSize;     // Buffer size in bytes.
    TFltPoint         VertDefPos;        // Default vertical displacement vector.
    TCIDMetric*       VertWidths;        // Vertical glyph widths -> 0..VertWidthsCount -1.
    UI32              VertWidthsCount;   // Number of vertical widths in the array.
    UI32              WMode;             // Writing Mode -> 0 == Horizontal, 1 == Vertical.
    float             XHeight;           // The height of lowercase letters measured from the baseline.
};
```

The function returns the most important properties of a font. The parameter *IFont* must be a valid pointer to a font object. Such a pointer is returned by GetPageText(), EnumDocFonts(), or by the content parser (see ParseContent()). The parameter *F* must be a valid pointer to a TPDFFontInfo structure.

The font metrics such as Ascent, Descent, HorzWidths, VertWidths, and so on are returned in font units. The values must be scaled to the given font size as follows:

```
double scale   = FontSize / 1000.0;
double ascent  = font.Ascent  * scale;
double descent = font.Descent * scale;
```

The font's encoding consists of exactly 256 Unicode values if set. The encoding is not set if a CID font is selected.

The font flags describe certain properties of the font. The most important flags are the following:

- 0x1   // Fixed pitch font
- 0x2   // Serif style
- 0x4   // Symbol font
- 0x8   // Script style
- 0x20   // Non-symbolic font
- 0x40   // Italic style
- 0x40000 // Force Bold (Type1 fonts only)

The full set of available flags can be found in the PDF Reference.

The members Length1, Length2, and Length3 are already validated. The sum of the lengths cannot exceed the buffer size.

The member CIDToGIDMap is a buffer of the original CIDToGIDMap stream if available. The buffer must be converted to an array of unsigned short. The values are stored in big endian format, high order byte first.

Remarks:

The function does not use the exception handling of DynaPDF. No error message is set if the function fails. However, the only possible error is out of memory if the function parameters are valid.

Note that the function was mainly designed to return the properties of fonts which were imported from external PDF files. Many properties of system fonts, like CharSet, CIDSet, ToUnicode CMap and so on cannot be returned since these objects will be created when the PDF file in memory is closed.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetFontInfoEx

Syntax:

```
LBOOL pdfGetFontInfoEx(
    const PPDF* IPDF,        // Instance pointer
    UI32 Handle,             // Font handle
    struct TPDFFontInfo* F) // Structure to be filled
```

The function retrieves the most important properties of a font like GetFontInfo() but accepts a font handle instead. A font handle is a simple array index. To enumerate all fonts of a document execute the function in a loop from zero to GetFontCount() - 1. See also GetFontInfo().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetFontOrigin

Syntax:

```
SI32 pdfGetFontOrigin(
    const PPDF* IPDF) // Instance pointer
```
```
typedef enum
{
    orDownLeft = 0, // Baseline
    orTopLeft  = 1  // Upper left corner of the font's bounding box
}TOrigin;
```

The function returns the current font origin. The font origin is automatically set to orTopLeft or orDownLeft if the coordinate system will be changed. The origin can be changed at runtime with the function SetFontOrigin().

```
Default value = orDownLeft
```

Font origin:

0 0

The origin is top left

GetFontMetrics

Syntax:

```
LBOOL pdfGetFontMetrics(
    const PPDF* IPDF,                    // Instance pointer
    UI32 Handle,                         // Font handle
    struct TPDFFontMetrics* Metrics)     // see below

struct TPDFFontMetrics
{
    UI32        StructSize;     // Must be set to sizeof(TPDFFontMetrics)
    float       Ascent;         // Ascent (usWinAscent)
    float       AvgWidth;       // Average character width
    float       CapHeight;      // Cap height
    float       DefWidth;       // Default width
    float       Descent;        // Descent
    UI32        FirstChar;      // First char
    float       FixedWidth;     // Width of all glyphs in a fixed pitch font
    TFStyle     FontStyle;      // Font style with which the font was loaded
    struct TBBox FontBBox;      // Font's bounding box
    LBOOL       IsFixedPitch;   // Is this a fixed pitch font?
    LBOOL       IsSymbolFont;   // Is this is a symbolic font?
    LBOOL       IsStdFont;      // Is this is a standard font?
    float       ItalicAngle;    // Italic angle of italic fonts
    UI32        LastChar;       // Last char
    float       LineGap;        // sTypoLineGap of the OS/2 table of TrueType and OpenType fonts
    UI32        MacStyle;       // macOS style flags
    TFStyle     RealStyle;      // This is the real style of the font
    float       StemV;          // Width of vertical stems
    float       StrikeoutPos;   // Strikeout position
    float       StrikeoutWidth; // Width of the strikeout line
    float       UnderlinePos;   // Underline position
    float       UnderlineWidth; // Width of the underline
    UI32        UnicodeRange1;  // Bitset -> TUnicodeRange1
    UI32        UnicodeRange2;  // Bitset -> TUnicodeRange2
    UI32        UnicodeRange3;  // Bitset -> TUnicodeRange3
    UI32        UnicodeRange4;  // Bitset -> TUnicodeRange4
    UI32        Weight;         // Font weight
    UI32        WidthClass;     // Width class of the font
    float       WinDescent;     // usWinDescent of the OS/2 table of TrueType and OpenType fonts
    UI32        WinStyle;       // fsSelection of the OS/2 table of TrueType and OpenType fonts
    float       XHeight;        // X-height
    float       TypoAscender;   // sTypoAscender of the OS/2 table of TrueType and OpenType fonts
    float       TypoLeading;    // Typographic leading or line height of the font. This is
                                // TypoAscender - Descent + LineGap. If typographic metrics are
                                // not available then TypoAscender is set to Ascent.
};
```

The function retrieves the most important metrics of a font. The parameter *Handle* must be a valid font handle that was returned by SetFont(), SetFontEx(), or SetCIDFont().

The member *StructSize* must be initialized to sizeof(TPDFFontMetrics) before the function can be called. The values are scaled to a normalized font size of 1000 units. In order to scale the values to an arbitrary font size multiply the values with (fontSize / 1000.0).

Example:

```
m.Ascent *= 12.0f / 1000.0f;
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetFontSearchOrder

Syntax:

```
void pdfGetFontSearchOrder(
   const PPDF* IPDF,        // Instance pointer
   TFontBaseType Order[4]) // Array to which the order can be copied
```

The function copies the current font search order to the parameter *Order*. Note that the function does not perform any validity check.

GetFontSelMode

Syntax:

```
SI32 pdfGetFontSelMode(
   const PPDF* IPDF) // Instance pointer

typedef enum
{
   smFamilyName     = 0,
   smPostScriptName = 1,
   smFullName       = 2
}TFontSelMode;
```

The function returns the current font selection mode. The font selection mode describes what kind of font name must be passed to SetFont(). Note that a font cannot be found if the wrong mode is set.

```
Default value = smFamilyName
```

GetFontSize

Syntax:

```
double pdfGetFontSize(
   const PPDF* IPDF) // Instance pointer
```

The function returns the font size of the active font.

Return values:

The function returns the font size, a value greater zero on success, or a negative error code on failure.

GetFontWeight

Syntax:

```
SI32 pdfGetFontWeight(
   const PPDF* IPDF) // Instance pointer
```

The function returns the weight that will be used to emulate a bold font style. See SetFontWeight() for further information.

Page 435 of 854

Font weights:

- `100 – 300  // Ultra light, Light`
- `400 – 500  // Standard`
- `600 – 1000 // Bold, Extra bold`

GetFTextHeight

Syntax:

```
double pdfGetFTextHeight(
   const PPDF* IPDF,  // Instance pointer
   TTextAlign Align,  // Base text alignment
   const char* AText) // Formatted text

typedef enum
{
   taLeft,
   taCenter,
   taRight,
   taJustify
}TTextAlign;
```

The function measures the height of a formatted text block. Before calling this function a font (see SetFont()) and the output rectangle must be set with the function SetTextRect(). The parameter *Height* of SetTextRect() should be set to **-1** to avoid page breaks. Otherwise the height of the first text block is returned that fits into the given height of the rectangle.

The parameter *AText* must contain the same text (incl. format tags if any) as which should be printed later with WriteFText().

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is the height of the formatted text block. If the function fails the return value is a negative error code.

GetFTextHeightEx

Syntax:

```
double pdfGetFTextHeightEx(
   const PPDF* IPDF,  // Instance pointer
   double Width,      // Width of output rectangle
   TTextAlign Align,  // Base text alignment
   const char* AText) // Formatted text
```

The function measures the height of a formatted text block. The function works in the same way as GetFTextHeight() but the output rectangle must not be set manually beforehand.

The height of the output rectangle is set to **-1** to avoid a page break during measuring.

The parameter *AText* must contain the same text (incl. format tags if any) as which should be printed later with WriteFText().

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is the height of the formatted text block. If the function fails the return value is a negative error code.

GetFullyQualifiedFieldName

```
SI32 pdfGetFullyQualifiedFieldName(
   const PPDF* IPDF, // Instance pointer
   UI32 Handle,      // Field handle
   UI16* ADDR Value) // Address of a UI16* variable
```

The function returns the fully qualified name of a field. Note that children of a radio button, for example, have no name by its own. The export value must be compared to distinguish the children fields in this case.

Return values:

If the function succeeds the return value is the name length in characters. If the function fails, the return value is a negative error code.

GetGlyphIndex

Syntax:

```
SI32 pdfGetGlyphIndex(
   const PPDF* IPDF, // Instance pointer
   UI32 Index)       // Code page, Unicode, or CJK index
```

The function returns the glyph index of the given code page or Unicode index. How *Index* must be defined depends on the code page with which the font was loaded. For example, if the font was loaded with a 8 bit code page, then indexes below 256 are treated as index into the code page table. Higher indexes are treated as Unicode value in this case. Note that all Unicode encoded glyphs of the font are accessible in this case, also if the font was loaded with a 8 bit code page!

At time of publication the function does not support CJK code pages which require a conversion to Unicode, e.g. cpCJK_Big5_Uni, cpCJK_EUC_JP_Uni, and so on.

Return values:

If the function succeeds the return value is greater or equal zero. If the function fails the return value is a negative error code.

GetGlyphOutline

Syntax:

```
SI32 pdfGetGlyphOutline(
    const PPDF* IPDF,                    // Instance pointer
    UI32 Index,                          // Glyph index
    struct TPDFGlyphOutline* Outline) // see below

struct TFRect
{
    SI16 MinX;
    SI16 MinY;
    SI16 MaxX;
    SI16 MaxY;
};
struct TI32Point
{
    SI32 x;
    SI32 y;
};
struct TPDFGlyphOutline
{
    float             AdvanceX; // Glyph width in horizontal writing mode.
    float             AdvanceY; // Glyph height in vertical writing mode.
    float             OriginX;  // Placement vector (vertical writing mode only).
    float             OriginY;  // Placement vector (vertical writing mode only).
    SI16              Lsb;      // Left side bearing (already applied, info only).
    SI16              Tsb;      // Top side bearing (already applied, info only).
    LBOOL             HaveBBox; // If true, BBox was computed.
    struct TFRect     BBox;     // Bounding box of the glyph outline.
    struct TI32Point* Outline;  // Encoded outline. See description.
    UI32              Size;     // Number of points.
};
```

The function returns the outline of a glyph of the active font. The usage is slightly different depending on the used programming language:

C, C++, Delphi

The function must be called twice, the first time with the parameter *Outline* set to NULL or nil to determine the size of the glyph outline (the function returns the number of points in this case). The member *Outline* of the TPDFGlyphOutline structure can now be allocated and the function can be called again. See example below the function description for further information.

C#, VB 6, VB .Net, PHP

The function returns the outline already in the first function call.

Note that not all glyphs have an outline. A space character, for example, has no outline and therefore the size can be zero. The function returns normalized outlines scaled to a font size of 1000 units.

Outline format

The member *Outline* of the TPDFGlyphOutline structure contains coodinates encoded as 24.6 bit fixed integer values. The remaining 2 bits contain the type of point or corresponding command that was returned:

```
TI32Point &p = glyph.Outline[i];
UI32 cmd = ((p.y & 3) << 2) | (p.x & 3); // Extract the command
SI32 x   = p.x >> 2;                     // Remove the command. Result = 24.6 bit fixed point value!
SI32 y   = p.y >> 2;                     // Remove the command. Result = 24.6 bit fixed point value!

double fx = x / 64.0;                    // Conversion to PDF units
double fy = y / 64.0;                    // Conversion to PDF units

cmd == 1 // MoveTo
cmd == 2 // LineTo
cmd == 3 // Bezier_2_3    // One more point follows
cmd == 4 // Bezier_1_2_3 // Two more points follow
```

Other values as listed above cannot occur and must be treated as an error.

Return values:

If the function succeeds the return value is the size of the outline in points. If the function fails the return value is a negative error code.

Example (C++):

```
void DrawGlyph(const PPDF* PDF, double PosX, double PosY, double FontSize, TPDFGlyphOutline &Outline)
{
    SI32 i, idx = 0, cmd, size = Outline.Size;
    double x, y, x1=0.0, y1=0.0, x2=0.0, y2=0.0;

    if (!size) return;

    double fs = FontSize / 1000.0;
    double  s = fs / 64.0;

    for (i = 0; i < size; i++)
    {
        TI32Point &p = Outline.Outline[i];
        cmd = ((p.y & 3) << 2) | (p.x & 3);
        x   = PosX + (p.x >> 2) * s;
        y   = PosY - (p.y >> 2) * s; // Variant for top down coordinates (+ otherwise).
        switch(cmd)
        {
            case 1: pdfMoveTo(PDF, x, y); break;
            case 2: pdfLineTo(PDF, x, y); break;
            case 3:
                switch(++idx)
                {
                    case 1: x1 = x; y1 = y; break;
                    case 2:
                        idx = 0;
                        pdfBezier_2_3(PDF, x1, y1, x, y);
                        break;
                }
                break;
            case 4:
                switch(++idx)
                {
                    case 1: x1 = x; y1 = y; break;
                    case 2: x2 = x; y2 = y; break;
                    case 3:
                        idx = 0;
```

```
                              pdfBezier_1_2_3(PDF, x1, y1, x2, y2, x, y);
                              break;
                }
                break;
            default: throw "Unknown error!";
        }
    }
    pdfClosePath(PDF, fmFill);
}

void DrawTextOutline(const PPDF* PDF, const BYTE* Text, SI32 Len)
{
    SI32 i, idx, size;
    TPDFGlyphOutline glyph;
    double x, y, scale, fontSize;

    pdfCreateNewPDF(PDF, "test.pdf");

    pdfSetPageCoords(PDF, pcTopDown);

    pdfAppend(PDF);

        memset(&glyph, 0, sizeof(glyph));

        // The font size is NOT considered by GetGlyphOutline()!
        pdfSetFont(PDF, "Arial", fsRegular, 10.0, false, cp1252);

        x = y = 50.0;
        fontSize = 40.0f;
        scale    = fontSize / 1000.0;

        for (i = 0; i < Len; i++)
        {
            idx = pdfGetGlyphIndex(PDF, Text[i]);
            if ((size = pdfGetGlyphOutline(PDF, idx, NULL)) >= 0)
            {
                glyph.Size = size;
                // Note that size can be zero!
                if (size > 0)
                    glyph.Outline = (TI32Point*)malloc(size * sizeof(TI32Point));
                pdfGetGlyphOutline(PDF, idx, &glyph);
                // See definition above
                DrawGlyph(PDF, x, y + fontSize, fontSize, glyph);
                x += glyph.AdvanceX * scale;

                if (size > 0) free(glyph.Outline);
            }
        }
    pdfEndPage(PDF);

    pdfCloseFile(PDF);
}
```

GetGoToAction

```
LBOOL pdfGetGoToAction(
    const PPDF* IPDF,                   // Instance pointer
    UI32 Handle,                        // Action handle
    struct TPDFGoToAction* Action)  // see below

struct TPDFGoToAction
{
    UI32        StructSize;      // Must be set to sizeof(TPDFGoToAction).
    SI32        DestPage;        // Destination page (the first page is denoted by 1).
    float*      DestPos;         // Array of 4 floating point values if set.
    TDestType   DestType;        // Destination type.
```

```
    // GoToR (Go To Remote) actions only:
    IFSR*       DestFile;       // see GetFileSpec().
    const char* DestNameA;      // Optional named destination that shall be loaded when
                                // opening the file.
    const UI16* DestNameW;      // Either the Ansi or Unicode string is set but never both.
    SI32        NewWindow;      // Meaningful only if DestFile points to a PDF file.
                                // -1 = viewer default, 0 = false, 1 = true.
    SI32        NextAction;     // -1 or next action handle to be executed if any.
    TActionType NextActionType; // Only set if NextAction is >= 0.
};
```

The function retrieves the properties of a GoTo or GoTo Remote action. The member *StructSize* must be set to sizeof(TPDFGoToAction) before the function can be called. The members *DestPos* and *DestType* are not meaningful for GoTo Remote actions.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetGoToRAction

Syntax:

```
LBOOL pdfGetGoToRAction(
    const PPDF* IPDF,               // Instance pointer
    UI32 Handle,                    // Action handle
    struct TPDFGoToAction* Action)  // see below
```

The functions GetGoToAction() and GetGoToRAction() are identically. Both functions can be called for GoTo and GoTo Remote actions. See GetGoToAction() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetGStateFlags

Syntax:

```
UI32 pdfGetGStateFlags(
    const PPDF* IPDF) // Instance pointer
```

The returns the current graphics state flags. The available flags are described at SetGStateFlags().

GetHideAction

```
LBOOL pdfGetHideAction(
    const PPDF* IPDF,               // Instance pointer
    UI32 Handle,                    // Action handle
    struct TPDFHideAction* Action)  // see below
struct TPDFHideAction
{
    UI32         StructSize;    // Must be set to sizeof(TPDFHideAction).
    const IFLD** Fields;        // Array of field pointers -> GetFieldEx2().
    UI32         FieldsCount;   // Number of fields in the array.
    LBOOL        Hide;          // Hide or show the fields in the array?
    SI32         NextAction;    // -1 or next action handle to be executed if any.
    TActionType  NextActionType; // Only set if NextAction is >= 0.
};
```

The function retrieves the properties of a Hide action. The member *StructSize* must be set to sizeof(TPDFHideAction) before the function can be called.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetIconColor

Syntax:

```
UI32 pdfGetIconColor(
    const PPDF* IPDF) // Instance pointer
```

The function returns the icon color used for newly created text annotations. The color value must always be defined in DeviceRGB because normal annotations do not support the color spaces DeviceGray or DeviceCMYK. See also SetIconColor().

GetImageBuffer

Syntax:

```
char* pdfGetImageBuffer(
    const PPDF* IPDF,  // Instance pointer
    UI32 ADDR BufSize) // Variable to get the buffer size in bytes
```

The function returns the buffer of an image that was created in memory by CreateImage(). The buffer size is set to the parameter *BufSize*. The returned pointer is a pointer to the original buffer; it must not be freed or changed. When the image buffer is no longer needed it must be released with FreeImageBuffer().

GetImageCount

Syntax:

```
SI32 pdfGetImageCount(
    const PPDF* IPDF,    // Instance pointer
    const char* FileName) // Image file
```

The function returns the number of images contained in a multi-page image. At time of publication TIFF images are supported only.

Return values:

If the function succeeds the return value is the number of images inside the image file. If the function fails the return value is a negative error code.

GetImageCountEx

Syntax:

```
SI32 pdfGetImageCountEx(
    const PPDF* IPDF,   // Instance pointer
    const void* Buffer, // Pointer to file buffer
    UI32 BufSize)       // Buffer size
```

The function determines the number of images in a multi-page image in the same way as GetImageCount(), but accepts a file buffer as input. At time of publication TIFF images are supported only.

GetImageHeight

Syntax:

```
SI32 pdfGetImageHeight(
    const PPDF* IPDF, // Instance pointer
    UI32 AHandle)     // Image handle
```

The function returns the height of an image in pixel. This value is may be smaller than the height of the original image, if it was downscaled by DynaPDF.

Return values:

If the function succeeds the return value is the height of the image in pixel. If the function fails the return value is a negative error code.

GetImageObj

Syntax:

```
LBOOL pdfGetImageObj(
    const PPDF* IPDF,   // Instance pointer
    UI32 Handle,        // Image handle
    TParseFlags Flags,  // See below
    TPDFImage* Image)   // Structure to be filled with data
typedef UI32 TParseFlags;
#define pfNone            0x00000000 // Default
#define pfDecomprAllImages 0x00000002 // Decompress all image types
#define pfNoJPXDecode     0x00000004 // Do not decompress JPEG2000 images
#define pfDitherImagesToBW 0x00000008 // Floyd-Steinberg dithering.
#define pfConvImagesToGray 0x00000010 // Convert the image to DeviceGray
#define pfConvImagesToRGB  0x00000020 // Convert the image to DeviceRGB
#define pfConvImagesToCMYK 0x00000040 // Convert the image to DeviceCMYK
#define pfImageInfoOnly    0x00000080 // Return only the image properties
```

The function retrieves the properties of an image as well as the decompressed image buffer if needed. By default all images are returned decompressed, with exception of image types which are already stored in a valid file format like JPEG and JPEG 2000 images.

If all image types should be decompressed set the flag pfDecompressAllImages.

This function allocates memory that should be released with FreeImageObj() when finish.

Image handles are simple array indexes. The number of image objects can be determined with GetImageObjCount(). Note that this array does not include inline images which are stored in content streams. Such images can only be accessed with ParseContent().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetImageObjCount

Syntax:

```
SI32 pdfGetImageObjCount(
   const PPDF* IPDF) // PDF Instance
```

The function returns the number of image objects which were loaded with DynaPDF functions or imported from external PDF files. The images can be accessed with GetImageObj().

Return values:

The return value is the number of image objects. This function cannot fail.

GetImageObjEx

Syntax:

```
LBOOL pdfGetImageObjEx(
   const PPDF* IPDF,        // Instance pointer
   const void* ImagePtr,    // Image pointer
   TParseFlags Flags,       // See description at GetImageObj()
   struct TPDFImage* Image) // Structure to be filled with data
```

The function can be used to decompress an image mask or soft mask that is associated with a base image that was returned by ParseContent() or GetImageObj().

The function allocates memory that should be released with FreeImageObjEx() when finish.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetImageWidth

Syntax:

```
SI32 pdfGetImageWidth(
   const PPDF* IPDF, // Instance pointer
   UI32 AHandle)     // Image handle
```

The function returns the width of an image in pixel. This value is may be smaller than the width of the original image, if it was downscaled by DynaPDF.

Return values:

If the function succeeds the return value is the width of the image in pixel. If the function fails the return value is a negative error code.

GetImportDataAction

Syntax:

```
LBOOL pdfGetImportDataAction(
    const PPDF* IPDF,                    // Instance pointer
    UI32 Handle,                         // Action handle
    struct TPDFImportDataAction* Action) // see below

struct TPDFImportDataAction
{
    UI32               StructSize;    // Must be set to sizeof(TPDFImportDataAction).
    struct TPDFFileSpecEx Data;        // The data or file to be loaded.
    SI32               NextAction;    // -1 or next action handle to be executed if any
    TActionType        NextActionType; // Only set if NextAction is >= 0.
};
```

The function retrieves the properties of an Import Data Action. The member *StructSize* must be set to sizeof(TPDFImportDataAction) before the function can be called. The struture TPDFFileSpecEx is described in detail at GetFileSpec(). It is not required to initialize the member *StructSize* of this struture too.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetImportFlags

Syntax:

```
UI32 pdfGetImportFlags(
    const PPDF* IPDF) // Instance pointer
```

The function returns the current import flags used to import PDF files. The flags are described in detail at SetImportFlags().

GetImportFlags2

Syntax:

```
UI32 pdfGetImportFlags2(
    const PPDF* IPDF) // Instance pointer
```

The function returns the current import flags used to import PDF files. The flags are described in detail at SetImportFlags2().

GetInBBox

Syntax:

```
SI32 pdfGetInBBox(
    const PPDF* IPDF,          // Instance pointer
    UI32 PageNum,              // Page number of external PDF file
    TPageBoundary Boundary,    // Type of bounding box to be returned
    struct TPDFRect ADDR BBox) // Bounding box
```

```
typedef enum
{
    pbArtBox,    // Art box
    pbBleedBox,  // Bleed box
    pbCropBox,   // Crop box
    pbTrimBox,   // Trim box
    pbMediaBox   // Media box
}TPageBoundary;

struct TPDFRect
{
    double Left;
    double Bottom;
    double Right;
    double Top;
};
```

The function retrieves a bounding box of an external PDF page. The external PDF file must be opened with the function OpenImportFile() or OpenImporBuffer() beforehand.

The bounding boxes are described in detail under SetBBox().

Return values:

If the function succeeds the return value is 1 and the parameter *BBox* is filled with values. If the function fails the return value is 0.

GetInDocInfo

Syntax:

```
SI32 pdfGetInDocInfo(
    const PPDF* IPDF,    // Instance pointer
    TDocumentInfo DInfo, // Document info entry
    UI16* ADDR Value)    // Value as Unicode string (null-terminated)

typedef enum
{
    diAuthor      = 0,
    diCreator     = 1,
    diKeywords    = 2,
    diProducer    = 3,
    diSubject     = 4,
    diTitle       = 5,
    diCompany     = 6,
    diPDFX_Ver    = 7,  // GetInDocInfo() or GetInDocInfoEx()
    diCustom      = 8,  // Not supported -> Use GetInDocInfoEx()
    diPDFX_Conf   = 9,  // GetInDocInfo() or GetInDocInfoEx()
    diCreationDate = 10, // Available after a PDF file was imported
    diModDate     = 11  // GetInDocInfo() or GetInDocInfoEx()
}TDocumentInfo;
```

The function retrieves a document info entry from an external PDF file as Unicode string. The external PDF file must be opened with the function OpenImportFile() or OpenImporBuffer() beforehand. The parameter *Value* holds a pointer to the original Unicode value, it must not be NULL. Do not change or free the value. This function support predefined document info entries only. User defined entries, as well as predefined entries can be enumerated with GetDocInfoEx().

Return value:

The return value is the string length in characters. If the info entry is not set it returns 0 and *Value* is initialized to NULL. If the function fails the return value is a negative error code.

GetInDocInfoCount

Syntax:

```
SI32 pdfGetDocInfoCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of available document info entries of the currently opened import file or a negative error code on failure. The document info entries can be enumerated with GetInDocInfoEx().

GetInDocInfoEx

Syntax:

```
SI32 pdfGetDocInfoEx(
   const PPDF* IPDF,        // Instance pointer
   UI32 Index,              // Entry index
   TDocumentInfo ADDR DInfo, // Document info entry type
   char* ADDR Key,          // Only set for user defined keys
   char* ADDR Value,        // The value of the info entry
   LBOOL ADDR Unicode)      // Is value in Unicode format?
```

The function returns a document info entry from the currently opened import file. The parameter *Index* must be a valid index into the array of document info entries. The number of available entries is returned by GetInDocInfoCount(). The parameter *Value* holds a pointer to the original value, it must not be NULL. Do not change or free the value. If the parameter *Unicode* is true, the value is a Unicode string. Make a type cast to UI16* in this case. The parameter *Key* contains always an Ansi string if set (user defined keys only).

Return values:

If the function succeeds the return value is the string length in characters of the parameter *Value*. Depending on the string format make a type cast to UI16*.

GetInEncryptionFlags

Syntax:

```
SI32 pdfGetInEncryptionFlags(
   const PPDF* IPDF) // Instance pointer
```

The function returns the encryption flags of the currently opened import file. If no flag is set the return value is zero. To determine whether a PDF file is encrypted call the function GetInIsEncrypted().

GetInFieldCount

Syntax:

```
SI32 pdfGetInFieldCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of top level fields included in the currently opened import file. The number of top level fields may not match the number of fields in the PDF file. For example, interactive forms created by Adobe's Designer contain usually one top level group field that contains all other fields of the document. The return value would be 1 in this case because the children of group fields are not taken into account.

Return values:

If the function succeeds the return value is the number of top level fields, a value greater or equal zero. If the function fails the return value is a negative error code.

GetInIsCollection

Syntax:

```
SI32 pdfGetInIsCollection(
   const PPDF* IPDF) // Instance pointer
```

The function checks whether the currently opened import file is a portable collection. See CreateCollection() for further information.

Return values:

If the function succeeds the return value is either 0 or 1 depending on whether the opened PDF file is a portable collection. If the function fails the return value is a negative error code.

GetInIsEncrypted

Syntax:

```
SI32 pdfGetInIsEncrypted(
   const PPDF* IPDF) // Instance pointer
```

The function checks whether the currently opened import file is encrypted.

Return values:

If the function succeeds the return value is either 0 or 1 depending on whether the opened PDF file is encrypted. If the function fails the return value is a negative error code.

GetInIsSigned

Syntax:

```
SI32 pdfGetInIsSigned(
    const PPDF* IPDF) // Instance pointer
```

This function can be used to determine whether a PDF file contains a digital signature. The PDF file must be opened with OpenImportFile() or OpenImportBuffer() beforehand.

The function checks only whether the signature flag is set in the global AcorForm object. Since not all PDF libraries set the signature flag, it is not guaranteed that the file contains no signature if the return value is false.

The only safe way to check for signatures is to import the file and search for deleted signature fields with GetFieldEx(). If the member *ISignature* is set and if the member *Contents* of the TPDFSigDict structure is non-NULL then the file contains a signature. See GetFieldEx() and GetSigDict() for further information.

Return values:

If the PDF file contains a digital signature the return value is 1. If the file was not digitally signed the function returns 0. A negative error code is returned on failure.

GetInIsTaggedPDF

Syntax:

```
SI32 pdfGetInIsTaggedPDF(
    const PPDF* IPDF) // Instance pointer
```

```
typedef enum TPDFMarkInfo
{
    miNotTagged      = 0, // This is no Tagged PDF file.
    miTagged         = 1, // This is a Tagged PDF file.
    miSuspects       = 2, // The file contains suspect contents that whose ordering does not ordered
                          // meet the requirements of the Tagged PDF specifications.
    miUserProperties = 4  // A flag indicating that at least one structure element contains user
                          // properties. User properties are stored in User Property dictionaries.
}TPDFMarkInfo;
```

The function can be used to check whether the currently open import file is a Tagged PDF file. The PDF file must be opened beforehand with OpenImportFile() or OpenImportBuffer().

The return value is a set flags indicating the tagging status of the file, see *TPDFMarkInfo* above.

Return values:

If the function succeeds, the return value is greater or equal zero. If the function fails, the return value is a negative error code.

GetInIsTrapped

Syntax:

```
SI32 pdfGetInIsTrapped(
   const PPDF* IPDF) // Instance pointer
```

This function returns the value of the Trapped key in the current open import file. The PDF file must be opened with OpenImportFile() or OpenImportBuffer() beforehand. If the key is not set the return value is false.

Return values:

If the function succeeds it returns 0 or 1 depending on the value of the Trapped key. A negative error code is returned on failure.

GetInIsXFAForm

Syntax:

```
SI32 pdfGetInIsXFAForm(
   const PPDF* IPDF) // Instance pointer
```

The function returns true if the current open import file contains an XFA form. If the form is a hybrid form the XFA part can be deleted and the remaining PDF form fields can be edited as usual.

XFA streams can be accessed with GetXFAStream() / GetXFAStreamCount().

The PDF file must be opened with OpenImportFile() or OpenImportBuffer() before this function can be called.

Return values:

If the function succeeds it returns 0 or 1 depending on whether the file contains an XFA form. If the function fails the return value is a negative error code.

GetInkList

Syntax:

```
LBOOL pdfGetInkList(
   const ILST* List, // List pointer returned by GetAnnotEx()
   float** Points,   // Address of a float* variable
   UI32* Count)      // Address of a UI32 variable
```

The function returns a path or ink list of an Ink annotation. The array contains x/y coordinate pairs (measured in bottom up coordinate system). The array length should be even but odd values can occur if the annotation was imported from a malformed document.

The points of an ink list are usually drawn with bezier curves to achieve a smooth transition between points.

Example (C++):

```
...
float* points = NULL;
TPDFAnnotationEx annot;
UI32 i, j, numAnnots, count = 0;

pdfEditPage(pdf, 1);
numAnnots = pdfGetPageAnnotCount(pdf);
for (i = 0; i < numAnnots; i++)
{
    if (pdfGetPageAnnotEx(pdf, i, annot))
    {
        if (annot.Type == atInk)
        {
            for (j = 0; j < annot.InkListCount; j++)
            {
                if (pdfGetInkList(annot.InkList[j], &points, &count))
                {
                    // Do something with the path
                }
            }
        }
    }
}
pdfEndPage(pdf);
...
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetInMetadata

Syntax:

```
LBOOL pdfGetInMetadata(
    const PPDF* IPDF, // Instance pointer
    SI32 PageNum,     // Page number or -1 to access the global XMP stream
    BYTE** Buffer,    // Address of a BYTE* pointer
    UI32* BufSize)    // Address of an unsigned integer variable
```

The function can be used to access the optional metadata streams of pages or the global metadata stream of the current open import file. Metadata streams are in XMP format that is a superset of XML. The PDF file must be opened with OpenImportImportFile() or OpenImportBuffer() beforehand.

Although the global XMP stream does usually exist in todays PDF files, metadata streams are optional and maybe not present. The function returns true if no error occurs, also if no metadata stream is present. The caller must also check whether the parameter *BufSize* was set to a value greater zero to determine whether stream data was returned.

The parameter *Buffer* is assigned to the original stream buffer, if any. DynaPDF is the owner of this data. Do not modify or free the returned memory block.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Usage (C++):

```
...
pdfOpenImportFile(pdf, "c:/test.pdf", ptOpen, NULL);
BYTE* buffer = NULL;
UI32 bufSize = 0;
if (pdfGetInMetadata(pdf, -1, &buffer, &bufSize))
{
   if (bufSize > 0)
   {
      // Do something with the data
   }
}
```

GetInNamedDest (obsolete)

Syntax:

```
LBOOL pdfGetInNamedDest(
   const PPDF* IPDF,           // Instance pointer
   UI32 Index,                 // Array index
   struct TPDFNamedDest* Dest) // Structure to be filled
```

This function is no longer supported. Please use the new function GetNamedDest() instead.

GetInNamedDestCount (obsolete)

Syntax:

```
SI32 pdfGetInNamedDestCount(
   const PPDF* IPDF) // Instance pointer
```

This function is no longer supported. The function returns always zero also if the file contains named destinations. Please use the new function GetNamedDestCount() instead.

GetInOrientation

Syntax:

```
SI32 pdfGetInOrientation(
   const PPDF* IPDF, // Instance pointer
   SI32 PageNum)     // Page number
```

The function returns the orientation of a specific page within the currently open import file (see also OpenImportFile()).

Return values:

If the function succeeds the return value is the orientation of the page in degrees. Possible values are 0, 90, 180, 270, or the same values as negative numbers. If the function fails the return value is a negative error code, a value smaller than -360.

GetInPageCount

Syntax:

```
SI32 pdfGetInPageCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of pages contained in an external PDF file. The external PDF file must be opened with the function OpenImportFile() or OpenImporBuffer() beforehand.

Return values:

If the function succeeds the return value is the number of pages. If the function fails the return value is a negative error code.

GetInPDFVersion

Syntax:

```
SI32 pdfGetInPDFVersion(
   const PPDF* IPDF) // Instance pointer
```

The function returns the PDF version number of an external PDF file. Values below 10 are the minor version: 3 for PDF 1.3, 4 for PDF 1.4 and so on.

If the major version is higher than 1, e.g. PDF 2.0, the function returns the version as follows:

```
MajorVersion * 10 + MinorVersion
```

For PDF 2.1 the result would be 21.

The PDF file must be opened with OpenImportFile() or OpenImporBuffer() beforehand.

Return values:

If the function succeeds the return value is the minor version number. If the function fails the return value is a negative error code.

GetInPDFVersionEx

Syntax:

```
SI32 pdfGetInPDFVersionEx(
   const PPDF* IPDF,                // Instance pointer
   struct TPDFVersionInfo* Value) // Version structure or NULL
```

The function retrieves the version information of the currently open import file. The function can be called with or without the structure TPDFVersionInfo. If value is set to NULL, the function returns the version constant as defined in the enum TPDFVersion or -1 if the version is not supported by DynaPDF.

A return value smaller than -1 indicates that an error occurred.

The version constants are described in detail at GetPDFVersionEx().

GetInPrintSettings

Syntax:

```
LBOOL pdfGetInPrintSettings(
  const PPDF* IPDF,
  struct TPDFPrintSettings* Settings)

struct TPDFPrintSettings
{
    TDuplexMode DuplexMode;        // See below
    SI32  NumCopies;               // -1 means not set. The maximum value is 5
    SI32  PickTrayByPDFSize;       // -1 means not set. 0 == false, 1 == true
    // If set, the array contains PrintRangesCount * 2 values. Each pair
    // consists of the first and last page of the sub-range. The first page in
    // the PDF file is denoted by 0.
    UI32*       PrintRanges;
    UI32        PrintRangesCount;  // Number of ranges
    TPrintScaling PrintScaling;    // psNone means not set
    /* 9 reserved fields follow*/
};

typedef enum
{
  dpmNone,          // Use the default value of the viewer
  dpmSimplex,
  dpmFlipShortEdge,
  dpmFlipLongEdge
}TDuplexMode;
```

The function retrieves the print settings of the current open import file. The PDF file must be opened with OpenImportFile() or OpenImporBuffer() before the function can be used. The print settings are used to initialize the print dialog in a viewer application. See also SetPrintSettings().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetInRepairMode

Syntax:

```
LBOOL pdfGetInRepairMode(
   const PPDF* IPDF) // Instance pointer
```

This function can be used to determine whether the current open import file was opened in repair or normal mode. When a PDF file was loaded in normal mode and if it contains damages which were not recognized when opening the file, e.g. if it is not possible to load one or more pages of it, then it is possible to load the file explicitly in repair mode to repair the damages. Not all damages can be repaired but DynaPDF contains a very robust and powerful PDF parser that is able to repair many damages in a PDF file.

See also OpenImportFile().

Return values:

If the current open import file was already opened in repair mode then the return value is 1. If it was loaded in normal mode then the return value is 0. If no import file is in memory the return value is a negative error code.

GetIsFixedPich

Syntax:

```
LBOOL pdfGetIsFixedPitch(
    const PPDF* IPDF) // Instance pointer
```

The function returns 1 if the active font is a fixed pitch font or 0 if the font is a variable pitch font. You must set a font before this function can be used (see SetFont() for further information).

Return values:

If the function succeeds the return value 0 or 1 depending on whether the font is a fixed pitch font. If the function fails the return value is a negative error code.

GetIsTaggingEnabled

Syntax:

```
LBOOL pdfGetIsTaggingEnabled(
    const PPDF* IPDF) // Instance pointer
```

This function can be used to determine whether tagging is enabled. If the function is called within an open page then it checks whether tagging is enabled for this page. EditPage() does not fully disable tagging when the structure information of a specific page is damaged.

When the function is called outside of an open page then it checks whether tagging for the entire document is enabled. Tagging can be disabled when a page of a PDF file with tagging information was imported while a structure tree was already in memory.

Return values:

If tagging is enabled the return value is 1. If tagging is disabled the return value is 0.

GetItalicAngle

Syntax:

```
double pdfGetItalicAngle(
    const PPDF* IPDF) // Instance pointer
```

The function returns the italic angle in degrees that will be used to emulate italic font styles. This value is only used if an italic version of the wished font could not be found. See SetItalicAngle() for further information.

```
Default value = 14.0
```

GetJavaScript

Syntax:

```
char* pdfGetJavaScript(
    const PPDF* IPDF,    // Instance pointer
    UI32 AHandle,        // Handle of global JavaScript
    UI32 ADDR Len,       // String length in characters
    LBOOL ADDR Unicode)  // If true, the return value is a Unicode string
```

The function returns a global JavaScript as string. The parameter *AHandle* must be a valid handle of a global JavaScript. The parameter *Len* holds the string length in characters without null-terminator. If *Unicode* is true, the return value is a Unicode string. Make a typecast to UI16* in the latter case. The parameters *Len* and *Unicode* must both not be NULL.

The returned string is a pointer to the original value. Do not change or free the string.

To enumerate all JavaScripts of a document use the function GetJavaScriptCount() and use this value in a simple for statement:

```
...
LBOOL unicode;
UI32 len;
char* scriptA;
UI16* scriptW;
SI32 count = pdfGetJavaScriptCount();
for (i = 0; i < count; i++)
{
    scriptA = pdfGetJavaScript(i, len, unicode);
    if (scriptA)
    {
        if (unicode)
        {
            scriptW = (UI16*)scriptA; // String is in Unicode format
            ...
        }else
        {
            ...
        }
    }
}
```

Remarks:

To get a global JavaScript by using the script's name use the function GetJavaScriptEx().

Return value:

If the function succeeds the return value is the JavaScript as Ansi or Unicode string. If the function fails the return value is NULL.

GetJavaScriptAction (obsolete)

Syntax:

```
char* pdfGetJavaScriptAction(
    const PPDF* IPDF,     // Instance pointer
    UI32 AHandle,         // Handle of a JavaScript action
    UI32 ADDR Len,        // String length in characters
    LBOOL ADDR Unicode)   // If true, the return value is a Unicode string
```

The function returns the script of a JavaScript Action as string. The parameter *AHandle* must be a valid handle of a JavaScript Action. The parameter *Len* holds the string length in characters without null-terminator. If *Unicode* is true, the return value is a Unicode string. Make a typecast to UI16* in the latter case. The returned string is a pointer to the original value. Do not change or free the string.

This function is obsolete. Please use GetJavaScriptActionEx() instead.

Return value:

If the function succeeds the return value is the JavaScript as Ansi or Unicode string. If the function fails the return value is NULL.

GetJavaScriptAction2 (obsolete)

Syntax:

```
char* pdfGetJavaScriptAction2(
    const PPDF* IPDF,       // Instance pointer
    TObjType ObjType,       // Object to which the object handle corresponds
    UI32 ObjHandle,         // Handle of the object that contain the action
    UI32 ActIndex,          // Action index that should be accessed
    UI32 ADDR Len,          // Variable to which the length can be set
    LBOOL ADDR Unicode,     // If true, the script is defined in Unicode
    TObjEvent ADDR Event)   // The event in which the action is executed

typedef enum
{
    oeNoEvent,          // Internal -> cannot occur
    oeOnOpen,           // Catalog, Pages
    oeOnClose,          // Pages only
    oeOnMouseUp,        // All fields, page link annotations, bookmarks
    oeOnMouseEnter,     // Form fields only
    oeOnMouseExit,      // Form fields only
    oeOnMouseDown,      // Form fields only
    oeOnFocus,          // Form fields only
    oeOnBlur,           // Form fields only
    oeOnKeyStroke,      // Text fields only
    oeOnFormat,         // Text fields only
    oeOnCalc,           // Text fields, combo boxes, list boxes
    oeOnValidate,       // All form fields, except buttons
    oeOnPageVisible,    // PDF 1.5 -> Form fields only
    oeOnPageInVisible,  // PDF 1.5 -> Form fields only
    oeOnPageOpen,       // PDF 1.5 -> Form fields only
    oeOnPageClose,      // PDF 1.5 -> Form fields only
    oeOnBeforeClosing,  // PDF 1.4 -> Catalog only
    oeOnBeforeSaving,   // PDF 1.4 -> Catalog only
    oeOnAfterSaving,    // PDF 1.4 -> Catalog only
    oeOnBeforePrinting, // PDF 1.4 -> Catalog only
```

```
    oeOnAfterPrinting   // PDF 1.4 -> Catalog only
}TObjEvent;

typedef enum
{
   otAction,
   otAnnotation,
   otBookmark,
   otCatalog,     // PDF 1.4
   otField,
   otPage,
   otPageLink
}TObjType;
```

The function returns the script of a JavaScript Action. The return value is a pointer to the original string buffer. This value must not be changed or freed.

The value of the parameter *ObjHandle* depends on the object type. If the object type is a page, the page number must be used as object handle. The number of actions which are assigned to a specific object is returned by the function GetObjActionCount(). The parameter *ActIndex* represents is array index of the action that should be accessed. The first action in the array has the index 0.

The parameter *Event* is set to the event that causes the execution of the script. If the script is defined in Unicode the parameter *Len* is set to the length in characters. Make a typecast to UI16* in this case because Unicode scripts use 16 bit per character.

This function is obsolete. Please use GetObjActions() / GetJavaScriptActionEx() instead.

Return values:

If the function succeeds the return value is a pointer to the original script buffer. If the function fails or if the action contains an empty script the return value is NULL. However, the function produces an error on failure. If no error callback function is set, call GetErrorMessage() to determine whether an error occurred.

GetJavaScriptActionEx

```
LBOOL pdfGetJavaScriptActionEx(
   const PPDF* IPDF,                     // Instance pointer
   UI32 Handle,                          // Action handle
   struct TPDFJavaScriptAction* Action) // see below

struct TPDFJavaScriptAction
{
   UI32        StructSize;    // Must be set to sizeof(TPDFJavaScriptAction).
   const char* ScriptA;       // The script.
   const UI16* ScriptW;       // Either the Ansi or Unicode string is set but never both.
   UI32        ScriptLen;     // Script length in characters, not bytes!
   SI32        NextAction;    // -1 or next action handle to be executed if any.
   TActionType NextActionType; // Only set if NextAction is >= 0.
};
```

The function retrieves the script of a JavaScript action. The member *StructSize* must be initialized to sizeof(TPDFJavaScriptAction) before calling the function. *NextAction* is the handle of the next action that must be executed, if any.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetJavaScriptCount

Syntax:

```
SI32 pdfGetJavaScriptCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of global JavaScripts contained in a document.

GetJavaScriptEx

Syntax:

```
char* pdfGetJavaScriptEx(
   const PPDF* IPDF,    // Instance pointer
   const char* Name,    // Name of the global JavaScript
   UI32 ADDR Len,       // String length in characters
   LBOOL ADDR Unicode) // If true, return value is a Unicode string
```

The function returns a global JavaScript as string by using the script's name instead of a handle to identify the script. Each global JavaScript has a unique name that identifies the script; this name must be passed to the parameter *Name*. See also AddJavaScript().

The parameter *Len* holds the string length in characters without null-terminator. If *Unicode* is true, the return value is a Unicode string. Make a typecast to UI16* in the latter case. The parameters *Len* and *Unicode* must both not be NULL.

The returned string is a pointer to the original value. Do not change or free the string.

To enumerate all JavaScripts of a document use the function GetJavaScriptCount() and use this value in a simple for statement (see GetJavaScript() for further information).

Return values:

If the function succeeds the return value is the JavaScript as Ansi or Unicode string. If the function fails the return value is NULL.

GetJavaScriptName

Syntax:

```
char* pdfGetJavaScriptName(
   const PPDF* IPDF,    // Instance pointer
   UI32 Handle,         // Handle of global JavaScript
   UI32 ADDR Len,       // Name length in characters
   LBOOL ADDR Unicode)  // If true, the name is in Unicode format
```

The function returns the name of a global JavaScript. The parameter *Handle* must be a valid handle of a global JavaScript. If the parameter *Unicode* is true the returned string is in Unicode format. Make a typecast to UI16* in this case. The parameter *Len* holds the string length in characters.

Use the function GetJavaScriptCount() to determine the number of available scripts. The handles of global JavaScripts are simple array indexes.

Return value:

If the function succeeds the return value is the JavaScript as Ansi or Unicode string. If the function fails the return value is NULL.

GetJPEGQuality

Syntax:

```
SI32 pdfGetJPEGQuality(
   const PPDF* IPDF) // Instance pointer
```

The function returns the current JPEG compression quality which is used when compressing images with the JPEG or JPEG2000 compression filter. The return value can be negative indicating that the pass-through mode is disabled. The absolute value represents the wished image quality in percent if the JPEG compression filter is used.

If the JPEG2000 compression filter is used the value represents a divisor of the uncompressed image size to the wished compressed image size. The absolute value ranges from 0 to 1000. If the value is 0 or 1000 then the loss-less variant of JPEG2000 compression is used (see also SetJPEGQuality()).

GetLanguage

Syntax:

```
char* pdfGetLanguage(
   const PPDF* IPDF) // Instance pointer
```

The function returns the language identifier of the document as an ISO 3166 language tag or IANA tag, or NULL if not set. Language tags are defined as normal Ansi string values. The function returns a pointer to the original string, do not change, or free the value.

The Language identifiers are described in detail under SetLanguage().

GetLastTextPosX, GetLastTextPosY

Syntax:

```
double pdfGetLastTextPosX/Y(
    const PPDF* IPDF) // Instance pointer
```

The functions return the end position of the last drawn text. The returned y-coordinate is the position of the text's baseline measured in bottom-up coordinates.

Return values:

If the function succeeds the return value is the x- or y-coordinate. This value can be negative depending on the coordinate where the string was placed. If the function fails the return value is a negative error code. An error code is smaller than -33554531.0.

Example 1 (Top Down Coordinates):

```
...
pdfSetPageCoords(PDF, pcTopDown);

pdfAppend(PDF);
double fontSize = 20.0;
pdfSetFont(PDF, "Times", fsItalic, fontSize, true, cp1252);
/*
    Because we use top down coordinates in this example we must transform
    the returned y-coordinate to top down coordinates. To achieve this we
    must subtract the value from the page height.
    The coordinate origin of the text in top down coordinates is normally
    the upper left corner. So, we must also subtract the font size because
    the provided y-coordinate lies on the text's baseline.
*/
pdfWriteFTextEx(PDF, 50, 50.0, 200.0, -1.0, taLeft, "Some text that "
    "ends on an unknown position...");
pdfSetFillColor(PDF, PDF_RED);
pdfWriteText(PDF,
                pdfGetLastTextPosX(PDF),
                pdfGetPageHeight(PDF)-pdfGetLastTextPosY(PDF)-fontSize,
                "We are behind the last text line");

pdfEndPage(PDF);
...
```

Example 2 (Rotated text):

```
...
pdfSetPageCoords(PDF, pcTopDown);

pdfAppend(PDF);
pdfSetFont(PDF, "Times", fsItalic, 20.0, true, cp1252);
/*
    The coordinate origin that is passed to RotateCoords() is measured in
    top down coordinates in this example. However, after the function was
    called bottom-up coordinates are active. Because the calculated end
    point is measured in bottom-up coordinates too we can directly use the
    coordinates to place some text behind the previous string. This works
    of course only if the rotated coordinate system is still active.
*/
pdfSaveGraphicState(PDF);
pdfRotateCoords(PDF, 30.0, 150.0, 350.0);
// Note that buttom-up coordinates are now active
pdfWriteFTextEx(PDF, 50, 50.0, 200.0, -1.0, taLeft, "Some rotated text "
    "that ends on an unknown position...");
```

```
pdfSetFillColor(PDF, PDF_RED);
pdfWriteText(PDF,
                 pdfGetLastTextPosX(PDF),
                 pdfGetLastTextPosY(PDF),
                 "We are behind the last text line");

pdfRestoreGraphicState(PDF);
pdfEndPage(PDF);
...
```

Example 3 (WriteAngleText()):

```
...
pdfSetPageCoords(PDF, pcTopDown);

pdfAppend(PDF);
pdfSetFont(PDF, "Times", fsItalic, 20.0, true, cp1252);
/*
    WriteAngleText() calculates the absolute end point measured in bottom-
    up coordinates. Note that the font origin is taken into account. As
    long as no further coordinate transformations are applied the
    coordinates are directly usable with exception that we must transform
    the y-coordinate to top down coordinates in this example.
*/
pdfWriteAngleText(PDF, "Some text...,", 10.0, 150.0, 150.0, 0.0, 0.0);

pdfSetFillColor(PDF, PDF_RED);
pdfWriteText(PDF,
                 pdfGetLastTextPosX(PDF),
                 pdfGetPageHeight(PDF)-pdfGetLastTextPosY(PDF),
                 "We are behind the last text line");

pdfEndPage(PDF);
...
```

GetLaunchAction

Syntax:

```
LBOOL pdfGetLaunchAction(
    const PPDF* IPDF,                    // Instance pointer
    UI32 Handle,                         // Action handle
    struct TPDFLaunchAction* Action)     // see below

struct TPDFLaunchAction
{
    UI32        StructSize;      // Must be set to sizeof(TPDFLaunchAction).
    const char* AppName;         // Optional. The name of the app that should be launched.
    const char* DefDir;          // Optional default directory.
    IFSR*       File;            // see GetFileSpec().
    SI32        NewWindow;       // -1 = viewer default, 0 = false, 1 = true.
    SI32        NextAction;      // -1 or next action handle to be executed if any.
    TActionType NextActionType;  // Only set if NextAction is >= 0.
    const char* Operation;       // Optional. The operation to perform (open or print).
    const char* Parameter;       // Optional parameter that shall be passed to the app.
};
```

The function retrieves the properties of a launch action. The member StructSize must be set to sizeof(TPDFLaunchAction) before the function can be called.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetLayerConfig

Syntax:

```
LBOOL pdfGetLayerConfig(
    const PPDF* IPDF,                 // Instance pointer
    SI32 Index,                       // Index or -1 for the default config
    struct TPDFOCLayerConfig* Out)    // Required. See below

struct TPDFOCLayerConfig
{
    UI32        StructSize; // Must be set to sizeof(TOCLayerConfig)
    TOCGIntent  Intent;     // Possible values oiDesign, oiView, or oiAll.
    LBOOL       IsDefault;  // If true, this is the default configuration.
    const char* NameA;      // Optional configuration name. The default config has
                            // usually no name but all others should have one.
    const UI16* NameW;      // Either the Ansi or Unicode string is set at time.
    UI32        NameLen;    // Length in characters.
};
```

The function retrieves the most important properties of a layer configuration. A PDF file can contain one or more layer configurations so that different layers can be initially shown or hidden.

A PDF file that contains layers (Optional Content Groups in PDF syntax) contains usually at least a default configuration dictionary. This configuration is loaded by default when the visibility state of a layer or optional content group must be determined, e.g. when rendering a page.

To determine the number of available configurations call GetLayerConfigCount(). The first index is denoted by 0.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetLayerConfigCount

Syntax:

```
SI32 pdfGetLayerConfigCount(
    const PPDF* IPDF) // Instance pointer
```

The function returns the number of available layer configurations. Note that it is possible that a document contains layers but no layer configuration. To determine whether a document contains layers or optional content groups use GetOCGCount().

To load a specific layer configuration call LoadLayerConfig().

GetLeading

Syntax:

```
double pdfGetLeading(
    const PPDF* IPDF) // Instance pointer
```

The function returns the current leading. The leading is the distance between two text lines. The default leading in PDF is the font size. The property is used by the function AddContinueText() and WriteFText(). A value of 0 determines that the font size should be used as leading, this is the default behaviour.

```
Default value = 0
```

GetLineCapStyle

Syntax:

```
SI32 pdfGetLineCapStyle(
    const PPDF* IPDF) // Instance pointer

typedef enum
{
    csButtCap    = 0,
    csRoundCap   = 1,
    csSquareCap  = 2
}TLineCapStyle;
```

The function returns the current line cap style used for vector graphics. The meaning of the values are described in detail under SetLineCapStyle().

```
Default value = csButtCap
```

GetLineJoinStyle

Syntax:

```
SI32 pdfGetLineJoinStyle(
    const PPDF* IPDF) // Instance pointer

typedef enum
{
    jsMiterJoin = 0,
    jsRoundJoin = 1,
    jsBevelJoin = 2
}TLineJoinStyle;
```

The function returns the current line join style used for vector graphics. The meaning of the values are described in detail under SetLineJoinStyle().

```
Default value = jsMiterJoin
```

GetLineWidth

Syntax:

```
double pdfGetLineWidth(
    const PPDF* IPDF) // Instance pointer
```

The function returns the current line width used for stroked vector graphics and the border of interactive objects.

GetLinkHighlightMode

Syntax:

```
SI32 pdfGetLinkHighlightMode(
    const PPDF* IPDF) // Instance pointer

typedef enum
{
    hmNone,    // Default
    hmInvert,  // Invert the contents of the annotation's bounding box
    hmOutline, // Invert the annotations border
    hmPush,    // Simulate a push button effect
    hmPushUpd  // Update appearance stream on changes
}THighlightMode;
```

The function returns the current highlight mode used for newly created annotations.

GetLogMetafileSize

Syntax:

```
SI32 pdfGetLogMetafileSize(
    const PPDF* IPDF,    // Instance pointer
    const char* FileName, // File path to EMF file
    struct TRectL* R)    // out -> Bounding rectangle

struct TRectL
{
    SI32 Left;
    SI32 Top;
    SI32 Right;
    SI32 Bottom;
};
```

The function retrieves the logical bounding box of an enhanced or Windows metafile. The parameter *FileName* must be the file path to the EMF or WMF file. The parameter *R* gets the unscaled logical bounding box of the metafile. This bounding box is required to calculate a user defined cutting area or viewport (see the example below the description of the function).

Remarks:

Two WMF formats are available, the old non-portable WMF format and the newer portable WMF format. Both formats must be converted to EMF with the GDI function SetWinMetaFileBits() before the logical bounding box can be calculated (the conversion is done automatically). However, non-portable WMF files are device-dependent, they contain no size information. To get correct results, the default size must be set with the function SetWMFDefExtent() beforehand. The parameters

Width and *Height* are passed to the member *xExt, yExt* of the structure METAFILEPICT which is required to convert WMF files to EMF. The default size is 0, 0, that means the GDI calculates the size automatically but mostly incorrect.

Because the GDI function SetWinMeatFileBits() is used to convert WMF files to EMF, WMF files are not supported under Linux or UNIX.

Return Values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (Delphi):

```
// In this example, we have a paintbox (TPaintBox) on the form
// into which we paint an EMF file. The user can draw a rectangle when
// pressing the left mouse button, this rectangle is our cutting area.
// FView is a private variable of the type TRectL. Note that this
// rectangle must also be drawn in the OnPaint event of the paintbox.
procedure TForm1.PaintBoxMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y:
Integer);
var dc: HDC;
begin
   dc := PaintBox.Canvas.Handle;
   SetROP2(dc, R2_NOT);
   SelectObject(dc, GetStockObject(NULL_BRUSH));
   Rectangle(dc, FView.Left, FView.Top, FView.Right, FView.Bottom);
   FView.Left   := X;
   FView.Top    := Y;
   FView.Right  := X;
   FView.Bottom := Y;
end;


procedure TForm1.PaintBoxMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
var dc: HDC;
begin
    // The rectangle must be drawn twice, one time to delete the
    // previous one, and the next time to draw the rectangle with the
    // new size. There is no need to redraw the paintbox.
   if (ssLeft in Shift) then begin
     dc := PaintBox.Canvas.Handle;
     SetROP2(dc, R2_NOT);
     SelectObject(dc, GetStockObject(NULL_BRUSH));
     Rectangle(dc, FView.Left, FView.Top, FView.Right, FView.Bottom);
     FView.Right  := X;
     FView.Bottom := Y;
     Rectangle(dc, FView.Left, FView.Top, FView.Right, FView.Bottom);
   end;
end;
// ViewRect holds the current size of the graphic on screen.
procedure TForm1.CreatePDF(const EMF: String; const ViewRect: TRectL)
var pdf: TPDF; r: TRectL; sx, sy: Double; tmp: Integer;
begin
    pdf := nil;
    try
        pdf := pdf.Create;
        pdf.CreateNewPDFA('c:\dout.pdf');
        pdf.SetDocInfo(diSubject, 'EMF-Files');
        pdf.SetDocInfo(diTitle, 'Metafiles');
        pdf.SetDocInfo(diCreator, 'Delphi Test Application');
        pdf.Append;
        pdf.SetPageCoords(pcTopDown);
        pdf.GetLogMetafileSize(EMF, r);
        // Make sure that the rectangle contains correct values
        if (FView.Left > FView.Right) then begin
            tmp := FView.Left;
```

```
            FView.Left := FView.Right;
            FView.Right := tmp;
        end;
        if (FView.Top > FView.Bottom) then begin
            tmp := FView.Top;
            FView.Top := FView.Bottom;
            FView.Bottom := tmp;
        end;
        // ViewRect is the current size of the EMF graphic in pixel.
        sx := (r.Right - r.Left) / (ViewRect.Right - ViewRect.Left);
        sy := (r.Bottom - r.Top) / (ViewRect.Bottom - ViewRect.Top);
        // See how the rectangle must be calculated:
        tmp      := r.Left;
        r.Left   := Round(r.Left + FView.Left   * sx);
        r.Right  := Round(tmp    + FView.Right  * sx);
        tmp      := r.Top;
        r.Top    := Round(r.Top  + FView.Top    * sy);
        r.Bottom := Round(tmp    + FView.Bottom * sy);
        // Now we have the unscaled cutting area. We want to output the
        // EMF file onto the entire page with a border of 20 units so
        // that we must only check whether the width or the height can be
        // calculated by DynaPDF to preserve the aspect ratio.
        sx := (r.Right - r.Left) / pdf.GetPageWidth;
        sy := (r.Bottom - r.Top) / pdf.GetPageHeight;
        pdf.SetMetaConvFlags(mfClipView); // we clip the viewport
        if (sx > sy) then
            pdf.InsertMetafileExt(EMF,
                r, 20, 20, pdf.GetPageWidth - 40, 0)
        else
            pdf.InsertMetafileExt(EMF,
                r, 20, 20, 0, pdf.GetPageHeight - 40);
        pdf.EndPage;
        pdf.CloseFile;
    except
        on E: Exception do MessageDlg(E.Message, mtError, [mbOK], 0);
    end;
    if pdf <> nil then pdf.Free;
end;
```

GetLogMetafileSizeEx

Syntax:

```
SI32 pdfGetLogMetafileSizeEx(
    const PPDF* IPDF,    // Instance pointer
    const void* Buffer,  // EMF buffer
    UI32 BufSize,        // Buffer size in bytes
    struct TRectL* R)    // out -> Bounding rectangle
```

The function retrieves the logical bounding box of an enhanced or Windows metafile. The parameter *Buffer* must be a file buffer of an EMF or WMF file.

Because the GDI function SetWinMeatFileBits() is used to convert WMF files to EMF, WMF files are not supported under Linux or UNIX.

The original size of a metafile must be known to compute a user defined cutting area or viewport. See GetLogMetafileSize() for an example application.

Return Values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetMatrix

Syntax:

```
LBOOL pdfGetMatrix(
   const PPDF* IPDF, // Instance pointer
   struct TCTM* M)   // Structure which holds the transformation matrix

struct TCTM
{
   double a;
   double b;
   double c;
   double d;
   double x;
   double y;
};
```

The function retrieves the current transformation matrix. Affine transformations are used to scale, translate, shear, or reflect the coordinate system. Functions which change the coordinate system change always the current transformation matrix, e.g. RotateCoords(), TranslateCoords(), and so on. The transformation matrix can also be changed directly with SetMatrix().

GetMaxFieldLen

Syntax:

```
SI32 pdfGetMaxFieldLen(
   const PPDF* IPDF, // Instance pointer
   UI32 TxtField)    // Text field handle
```

The function returns the allowed maximum string length of a text field, or zero if the length is not restricted. The parameter *TxtField* must be a valid handle of a text field.

Return values:

If the function succeeds the return value is the allowed maximum string length of the text field. If the function fails the return value is a negative error code.

GetMeasureObj

Syntax:

```
LBOOL pdfGetMeasureObj(
   const IMSR* Measure,       // Pointer of a measure object
   struct TPDFMeasure* Value) // See below

struct TPDFMeasure
{
   UI32       StructSize;   // Must be set to sizeof(TPDFMeasure)
   LBOOL      IsRectilinear; // If true, the members of the rectilinear measure dictionary are set.
   /* --- Rectilinear measure dictionary --- */
   INFM**     Angles;       // Number format array to measure angles -> GetNumberFormatObj()
   UI32       AnglesCount;  // Number of objects in the array.
   INFM**     Area;         // Number format array to measure areas -> GetNumberFormatObj()
   UI32       AreaCount;    // Number of objects in the array.
   float      CXY;          // Optional, meaningful only when Y is present.
   INFM**     Distance;     // Number format array to measure distances -> GetNumberFormatObj()
```

```
    UI32        DistanceCount; // Number of objects in the array.
    float       OriginX;       // X-Origin of the measurement coordinate system.
    float       OriginY;       // Y-Origin of the measurement coordinate system.
    const char* RA;            // A text string expressing the scale ratio of the drawing.
    const UI16* RW;            // A text string expressing the scale ratio of the drawing.
    INFM**      Slope;         // Number format array to measure the slope -> GetNumberFormatObj()
    UI32        SlopeCount;    // Number of objects in the array.
    INFM**      X;             // Number format array for measurement of change along the x-axis and,
                               // if Y is not present, along the y-axis as well.
    UI32        XCount;        // Number of objects in the array.
    INFM**      Y;             // Number format array for measurement of change along the y-axis.
    UI32        YCount;        // Number of objects in the array.

    /* --- Geospatial measure dictionary --- */
    double*     Bounds;        // Array of numbers taken pairwise to describe the bounds for which
                               // geospatial transforms are valid.
    UI32        BoundCount;    // Number of values in the array. Should be a multiple of two.

    // The DCS coordinate system is optional.
    LBOOL       DCS_IsSet;     // If true, the DCS members are set.
    LBOOL       DCS_Projected; // If true, the DCS values contains a pojected coordinate system.
    SI32        DCS_EPSG;      // Optional, either EPSG or WKT is set.
    const char* DCS_WKT;       // Optional ASCII string.

    // The GCS coordinate system is required and should be present.
    LBOOL       GCS_Projected; // If true, the GCS values contains a pojected coordinate system.
    SI32        GCS_EPSG;      // Optional, either EPSG or WKT is set.
    const char* GCS_WKT;       // Optional ASCII string.

    double*     GPTS;          // Required, an array of numbers that shall be taken pairwise,
                               // defining points in geographic space as degrees of latitude and
                               // longitude, respectively.
    UI32        GPTSCount;     // Number of values in the array.
    double*     LPTS;          // Optional, an array of numbers that shall be taken pairwise to
                               // define points in a 2D unit square.
    UI32        LPTSCount;     // Number of values in the array.

    const char* PDU1;          // Optional preferred linear display units.
    const char* PDU2;          // Optional preferred area display units.
    const char* PDU3;          // Optional preferred angular display units.
};
```

The function retrieves the properties of a measure dictionary. The parameter *Measure* must be a valid pointer of a measure dictionary. Such a pointer can be returned by GetImageObj(), GetViewport(), ParseContent(), and other functions. The member *StructSize* must be set to sizeof(TPDFMeasure).

Two different types of measure dictionaries are available: one for rectilinear coordinate systems and one for geospatial coordinate systems.

The member *IsRectilinear* specifies which type was returned. Depending on the type different members of the structure are set. Take a look at the comments in the structure definition to determine which values will be present.

The function returns pointers of the original values. Do not modify or free any value!

A rectilinear measure dictionary contains one or more arrays of Number Format dictionaries. These objects can be accessed with GetNumberFormatObj().

To determine how measure information can be used please have a look into the PDF Reference 2.0, ISO/DIS 32000-2, Section Measurement properties.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetMetaConvFlags

Syntax:

```
UI32 pdfGetMetaConvFlags(
    const PPDF* IPDF) // Instance pointer

typedef UI32 TMetaFlags;
#define mfDefault         0x00000 // No flags
#define mfDebug           0x00001 // Insert debug comments
#define mfShowBounds      0x00002 // Show the bounding boxes of text
#define mfNoTextScaling   0x00004 // Do not scale text records
#define mfClipView        0x00008 // Clip the output rectangle
#define mfUseRclBounds    0x00010 // Use the raw bounding box rclBounds
#define mfNoClippingRgn   0x00040 // Ignore clipping regions
#define mfNoFontEmbedding 0x00080 // Don't embed fonts used by the EMF
#define mfNoImages        0x00100 // Ignore image records
#define mfNoStdPatterns   0x00200 // Ignore standard hatch patterns
#define mfNoBmpPatterns   0x00400 // Ignore bitmap patterns
#define mfNoText          0x00800 // Ignore text records
#define mfUseUnicode      0x01000 // Use always Unicode to print text
#define mfUseTextScaling  0x04000 // Scale text (see description)
#define mfNoUnicode       0x08000 // Avoid the usage of Unicode fonts
#define mfFullScale       0x10000 // Scale coordinates to Windows size
#define mfUseRclFrame     0x20000 // See description
#define mfDefBkModeTransp 0x40000 // Initial backg. mode is transparent
#define mfApplyBidiAlgo   0x80000 // Apply the bidirectional algorithm
// Obsolete flags -> These flags are ignored, do not longer use them!
#define mfUseSpacingArray   0x0020 // Enabled by default
#define mfIntersectClipRect 0x2000 // Enabled by default
```

The function returns the conversion flags used to convert enhanced metafiles to PDF. The return value is a bit-mask. Use a bitwise and operator to determine whether a specific flag is set. The conversion flags are described in detail under SetMetaConvFlags().

GetMetadata

Syntax:

```
LBOOL pdfGetMetadata(
    const PPDF* IPDF,     // Instance pointer
    TMetadataObj ObjType, // see below
    SI32 Handle,          // Object handle or -1 for the catalog object
    BYTE** Buffer,        // Address of a BYTE* pointer
    UI32* BufSize)        // Address of an unsigned integer variable

typedef enum
{
    mdoCatalog, // The global XMP stream of the document (no handle needed)
    mdoFont,    // Parameter Handle must be a font handle
    mdoImage,   // Parameter Handle must be an image handle
    mdoPage,    // Parameter Handle must be a page number
    mdoTemplate // Parameter Handle must be a template handle
}TMetadataObj;
```

The function can be used to access the optional metadata streams of pages, fonts, images, pages, templates, as well as the global metadata stream that is associated with the Catalog object. Metadata streams are in XMP format that is a superset of XML.

The global metadata stream will be created when this function is called. The returned stream is a preview of the XMP stream that will be stored in the file when CloseFile() or CloseFileEx() is called. Note that the creation and modification date will be updated when the file is closed (except if the stream will be modified or replaced with SetMetadata()).

The original global XMP stream of external PDF files can be accessed with GetInMetadata().

Metadata streams are optional and maybe not present. The function returns true if no error occurs, also if no metadata stream is present. The caller must also check whether the parameter *BufSize* was set to a value greater zero to determine whether stream data was returned.

The parameter *Buffer* is assigned to the original stream buffer, if any. DynaPDF is the owner of this data. Do not modify or free the returned memory block.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Usage (C++):

```
...
BYTE* buffer = NULL;
UI32 bufSize = 0;
// Get a preview of the global XMP stream
if (pdfGetMetadata(pdf, mdoCatalog, -1, &buffer, &bufSize))
{
   if (bufSize > 0)
   {
      // Do something with the data
   }
}
```

GetMissingGlyphs

Syntax:

```
UI32* pdfGetMissingGlyphs(
  const PPDF* IPDF, // Instance pointer
  UI32 ADDR Count)  // Variable that holds the number of missing glyphs
```

The function returns the character codes that could not be found in the active font. If all characters were found the return value is NULL.

Each character code is encoded as a 32 bit unsigned integer. The character codes which are stored in this array correspond to the source string that was used in the function that outputs the text. For example, if the active font was loaded with a 8 bit code page and if the Ansi version of WriteText()

was used, the array contains the 8 bit codes which could not be found. If the wide string version was used the missing 16 bit codes will be returned.

If the active font is a CID font that uses an external CMap, a missing character code was maybe constructed from a sequence of bytes or from two 16 bit codes (if a wide string function was used).

However, the 32 bit code can be easily converted back to the original code sequence. If the Ansi version of a string function was used the algorithm looks as follows (C++):

```
...
UI32 i, p, value, count;
UI32* missingGlyphs = pdfGetMissingGlyphs(pdf, count);
// More than 4 bytes cannot be encoded in a 32 bit integer
BYTE sequence[4];
for (i = 0; i < count; i++)
{
    value = missingGlyphs[i];
    p     = 0;
    while (value)
    {
        sequence[p] = (BYTE)value;
        value >>= 8;
        ++p; // This is the string length when the loop returns
    }
    // You can now process the sequence...
}
```

Because the wide string versions use 16 bits per character, the conversion looks as follows:

```
UI32 i, p, value;
// More than two 16 bit values cannot be encoded in a 32 bit integer
UI16 sequence[2];
for (i = 0; i < count; i++)
{
    value = missingGlyphs[i];
    p     = 0;
    shift = 0;
    while (value)
    {
        sequence[p] = (UI16)value;
        value >>= 16;
        ++p; // This is the string length when the loop returns
    }
}
```

All string functions report the warning for a missing glyph only once. For example, if you call GetTextWidth() with a text like "This is an €uro" and if the Euro character is not available in the font then a warning is produced. However, if you output the same or another text again that contains the missing Euro character too then no further warning is produced.

GetMiterLimit

Syntax:

```
double pdfGetMiterLimit(
    const PPDF* IPDF) // Instance pointer
```

The function returns the current miter limit used to draw stroked vector graphics. The graphics state parameter miter limit is described in detail under SetMiterLimit(). The initial miter limit to convert EMF files is 2.0.

```
Default value: 10.0
```

GetMovieAction

Syntax:

```
LBOOL pdfGetMovieAction(
    const PPDF* IPDF,                    // Instance pointer
    UI32 Handle,                         // Action handle
    struct TPDFMovieAction* Action) // see below

struct TPDFMovieAction
{
    UI32         StructSize;     // Must be set to sizeof(TPDFMovieAction).
    SI32         Annot;          // Optional. The movie annotation handle identifying the
                                 // movie that shall be played.
    float        FWPosition[2];  // FWPosition.
    UI32         FWScale[2];     // FWScale.
    const char*  Mode;           // Mode.
    SI32         NextAction;     // -1 or next action handle to be executed if any.
    TActionType  NextActionType; // Only set if NextAction is >= 0.
    const char*  Operation;      // Operation.
    float        Rate;           // Rate.
    LBOOL        ShowControls;   // ShowControls.
    LBOOL        Synchronous;    // Synchronous.
    const char*  TitleA;         // The title of a movie annotation that shall be played.
                                 // Either Annot or Title should be set, but not both.
    const UI16*  TitleW;         // Either the Ansi or Unicode string is set at time.
    float        Volume;         // Volume.
};
```

The function retrieves the properties of a Movie action. The member *StructSize* must be set to sizeof(TPDFMovieAction) before the function can be called.

The member FWScale is in the format numerator / denominator.

Return Values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetNamedAction

```
LBOOL pdfGetNamedAction(
    const PPDF* IPDF,                  // Instance pointer
    UI32 Handle,                       // Action handle
    struct TPDFNamedAction* Action)    // see below

struct TPDFNamedAction
{
    UI32         StructSize;      // Must be set to sizeof(TPDFNamedAction).
    const char*  Name;            // Only set if Type == naUserDefined.
    SI32         NewWindow;       // -1 = viewer default, 0 = false, 1 = true.
    SI32         NextAction;      // -1 or next action handle to be executed if any.
    TActionType  NextActionType;  // Only set if NextAction is >= 0.
    TNamedAction Type;            // Known pre-defined actions.
};

typedef enum
{
    naFirstPage,     // PDF 1.2 Go to the first page of the document
    naLastPage,      // PDF 1.2 Go to the last page of the document
    naNextPage,      // PDF 1.2 Go to the next page
    naPrevPage,      // PDF 1.2 Go to the previous page
    naGoBack,        // Go back to last page and position
    naOpenDlg,       // Display the file open dialog
    naPrintDlg,      // Display the print dialog
    naGeneralInfo,   // Display the general info tab
    naFontsInfo,     // Display the fonts info tab
    naSaveAs,        // Display the save as dialog (requires Acrobat)
    naSecurityInfo,  // Display the security settings
    naFitPage,       // Fit the page into the window
    naDeletePages,   // Delete one or more pages
    naQuit,          // Quit the application
    naUserDefined    // Internal value to store unknown imported values
}TNamedAction;
```

The function retrieves the properties of a Named action. The member *StructSize* must be set to sizeof(TPDFNamedAction) before the function can be called.

If the application does not recognize the name, it shall take no action. Only the first four names are defined in the PDF specs. All other pre-defined names were seen in PDF files created by Adobe's Acrobat or Distiller.

Return Values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetNamedDest

Syntax:

```
LBOOL pdfGetNamedDest(
    const PPDF* IPDF,              // Instance pointer
    UI32 Index,                    // Array index
    struct TPDFNamedDest* Dest)    // see below

struct TPDFNamedDest
{
    UI32            StructSize;   // Must be set to sizeof(TPDFNamedDest)
    const char*     NameA;        // The destination name
    const UI16*     NameW;        // The destination name
    UI32            NameLen;      // Length in characters
    const char*     DestFileA;    // If set, the destination is located in
    const UI16*     DestFileW;    // this file
    UI32            DestFileLen;  // Length in characters
    SI32            DestPage;     // Destination page number
    struct TPDFRect DestPos;      // Destination position
    TDestType       DestType;     // Destination type
};
```

The function returns the properties of a named destination. The member *StructSize* must be initialized to sizeof(TPDFNamedDest) before the function can be called. The structure size is used to identify different versions of the structure.

The interpretation of the member *DestPos* depends on the destination type. The different destination types are described in detail at SetBookmarkDest(). If *DestFileA* or *DestFileW* is set then the destination is located in this PDF file. Only one string format is set at time.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetNamedDestCount

Syntax:

```
SI32 pdfGetNamedDestCount(
    const PPDF* IPDF) // Instance pointer
```

The function returns the number of named destinations in the PDF file. The named destinations of an external PDF file are available after the PDF file was imported.

GetNeedAppearance

Syntax:

```
LBOOL pdfGetNeedAppearance(
    const PPDF* IPDF) // Instance pointer
```

The function returns true if the global NeedAppearance flag of the AcroForm is set. See SetNeedAppearance() for further information.

GetNumberFormatObj

Syntax:

```
LBOOL pdfGetNumberFormatObj(
    const INFM* NumberFmt, // Pointer of a Number Format dictionary
    struct TPDFNumberFormat* Value) // See below

struct TPDFNumberFormat
{
    UI32              StructSize; // Must be set to sizeof(TPDFNumberFormat)
    float             C;   // The conversion factor used to multiply a value in partial units.
    SI32              D;   // A positive integer that shall specify the precision or denominator of a
                          // fractional amount.
    TMeasureNumFormat F;   // See below
    LBOOL             FD;  // If true, a fractional value formatted according to the D entry may not
                          // have its denominator reduced or low-order zeros truncated.
    TMeasureLblPos    O;   // See below
    const char*       PSA; // Text to be concatenated to the left of the label specified by U.
    const UI16*       PSW; // Text to be concatenated to the left of the label specified by U.
    const char*       RDA; // Text for the decimal position in displaying numerical values.
    const UI16*       RDW; // Text for the decimal position in displaying numerical values.
    const char*       RTA; // Text to be used between orders of thousands of numerical values.
    const UI16*       RTW; // Text to be used between orders of thousands of numerical values.
    const char*       SSA; // Text that shall be concatenated after the label specified by U.
    const UI16*       SSW; // Text that shall be concatenated after the label specified by U.
    const char*       UA;  // Label -> should be a universally recognized abbreviation.
    const UI16*       UW;  // Label -> should be a universally recognized abbreviation.
};

typedef enum
{
    mnfDecimal,
    mnfFractional,
    mnfRound,
    mnfTruncate
}TMeasureNumFormat;

typedef enum
{
    mlpSuffix,
    mlpPrefix
}TMeasureLblPos;
```

The function retrieves the properties of a Number Format dictionary. The member *StructSize* must be set to sizeof(TPDFNumberFormat) before the function can be called.

The structure members are named as specified in the PDF Reference 2.0. To determine how the values must be used please have a look into the PDF Reference 2.0, ISO/DIS 32000-2, Section Measurement properties.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetObjActionCount (obsolete)

Syntax:

```
SI32 pdfGetObjActionCount(
    const PPDF* IPDF, // Instance pointer
    TObjType ObjType, // Object type
    UI32 ObjHandle)   // Object handle

typedef enum
{
    otAction,
    otAnnotation,
    otBookmark,
    otCatalog,
    otField,
    otPage,
    otPageLink
}TObjType;
```

The function returns the number of actions used by an object. If the object type is a page, then use the page number as handle.

The number of actions used by an object can be used to remove a specific action from an object. See also DeleteActionFromObj(), GetActionTypeEx().

This function is marked as obsolete. Please use GetObjActions() instead.

Return values:

If the function succeeds the return value is the number of actions used by an object. If the function fails the return value is a negative error code.

GetObjActions

Syntax:

```
SI32 pdfGetObjActions(
    const PPDF* IPDF,                    // Instance pointer
    TObjType ObjType,                    // see below
    UI32 ObjHandle,                      // Object handle
    struct TPDFObjActions* Actions) // see below

typedef enum
{
    otAction,
    otAnnotation,
    otBookmark,
    otCatalog,
    otField,
    otPage,
    otPageLink
}TObjType;

struct TPDFObjActions
{
    UI32       StructSize; // Must be set to sizeof(TPDFObjActions).
    SI32       Action;     // Action handle or -1 if not set.
    TActionType ActionType; // The type of the action if Action >= 0.
    const IEVT* Events;     // Additional events if any. -> GetObjEvent().
};
```

The function retrieves the first action and additional trigger events that should be executed when a specific event occurs. The member *StructSize* must be set to sizeof(TPDFObjActions) before the function can be called. If the object type is a page, then use the page number as handle.

Trigger events are supported by the global Catalog object (this is the document root), Pages, and Form Fields. All other objects do not support trigger events.

Actions are defined as a single linked list. That means one action can execute another action (if *NextAction* of that action is >= 0).

Note that an action can reference itself! The application must check whether an action is already in the execution list before it will be executed.

Return values:

- < 0: An error occurred, e.g. due to an invalid handle.
- 0: The object contains no action or trigger event.
- 1: The object contains an action.
- 2: The object contains a trigger event.
- 3: The object contains an action and a trigger event.

GetObjEvent

Syntax:

```
LBOOL pdfGetObjEvent(
    const IEVT* IEvent,        // Pointer of an event object
    struct TPDFObjEvent* Event) // see below
struct TPDFObjEvent
{
    SI32       Action;     // Action to be executed.
    TActionType ActionType; // The type of the action.
    TObjEvent  Event;      // The event when the action should be executed.
    IEVT*      Next;       // Pointer to the next event if any.
};
```

The function retrieves the properties of a trigger event that is defined for a specific object. The action of this event must only be executed if the specified event occurs, e.g. OnMouseUp and so on.

Trigger events are defined as a single linked list. Note that an object event can reference itself! The application must check whether an event is already in the list before the next event will be loaded.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetOCG

Syntax:

```
LBOOL pdfGetOCG(
    const PPDF* IPDF,     // Instance pointer
    UI32 Handle,          // OCG handle or array index
    struct TPDFOCG* Out)  // See below

struct TPDFOCG
{
    UI32        StructSize;    // Must be set to sizeof(TPDFOCG)
    UI32        Handle;        // Handle or array index
    UI32        Intent;        // Bitmask -> TOCGIntent -> Intent and visibility state
    const char* NameA;         // Layer name
    const UI16* NameW;         // Layer name
    LBOOL       HaveContUsage; // If true, the layer contains a Content Usage dictionary.
                               // -> GetOCGContUsage().
    // The following two members can only be set if HaveContUsage is true.
    UI32        AppEvents;     // Bitmask -> see TOCAppEvent. If non-zero, the layer is included in
                               // one or more app events which control the layer state.
    UI32        Categories;    // Bitmask -> see TOCGUsageCategory. The Usage Categories which
                               // control the layer state.
};

typedef enum
{
    aeExport = 1,
    aePrint  = 2,
    aeView   = 4
}TOCAppEvent;

typedef enum
{
    oucNone     = 0,
    oucExport   = 1,
    oucLanguage = 2,
    oucPrint    = 4,
    oucUser     = 8,
    oucView     = 16,
    oucZoom     = 32
}TOCGUsageCategory;

typedef enum
{
    oiDesign  = 2,
    oiView    = 4,
    oiAll     = 8,
    oiEmpty   = 16,
    oiVisible = 32  // If set, the layer is visible
}TOCGIntent;
```

The function returns the properties of an OCG, also called layer. An OCG handle is a simple array index. You can loop over all available OCGs from 0 to GetOCGCount() - 1.

The visibility state is encoded in the member *Intent*. The layer is visible if the the flag *oiVisible* is set.

If the member *HaveContUsage* is true, the OCG contains a Content Usage dictionary. The contents of this dictionary can be accessed with the function GetOCGContUsage(). A Content Usage dictionary contains additional information about a layer or OCG.

If the OCG is also included in an application event, this is the case if *AppEvent* is non-zero, then the Content Usage dictionary is used to control the visibility state of the layer. The member *Categories* specifies which categories control the visibility state.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetOCGContUsage

Syntax:

```
LBOOL pdfGetOCGContUsage(
    const PPDF* IPDF,              // Instance pointer
    UI32 Handle,                   // OCG handle or array index
    struct TPDFOCGContUsage* Out)  // See below

struct TPDFOCGContUsage
{
    UI32           StructSize;     // Must be set to sizeof(TPDFOCGContUsage)
    UI32           ExportState;    // 0 = Off, 1 = On, PDF_MAX_INT = not set
    const char*    InfoCreatorA;   // CreatorInfo -> The application that created the group
    const UI16*    InfoCreatorW;   // CreatorInfo -> The application that created the group
    const char*    InfoSubtype;    // CreatorInfo -> A name defining the type of content, e.g.
                                   // Artwork, Technical etc.
    const char*    LanguageA;      // A language code as described at SetLanguage()
    UI16*          LanguageW;      // A language code as described at SetLanguage()
    UI32           LangPreferred;  // 0 = Off, 1 = On, PDF_MAX_INT = not set. The preffered state if
                                   // there is a partial but no exact match of the lang id
    TOCPageElement PageElement;    // If the group contains a pagination artefact
    UI32           PrintState;     // 0 = Off, 1 = On, PDF_MAX_INT = not set
    const char*    PrintSubtype;   // The type of content that is controlled by the OCG, e.g.
                                   // Trapping, PrintersMarks or Watermark.
    UI32           UserNamesCount; // The user names can be accessed with GetOCGUsageUserName()
    TOCUserType    UserType;       // The user for whom the OCG is primarily intendet
    UI32           ViewState;      // 0 = Off, 1 = On, PDF_MAX_INT = not set
    float          ZoomMin;        // Minimum zoom factor at which the OCG should be On. -1 = not set
    float          ZoomMax;        // Maximum zoom factor at which the OCG should be On. -1 = not set
};

typedef enum
{
  utIndividual,
  utOrganization,
  utTitle,
  utNotSet
}TOCUserType;
```

The function returns the properties of the Content Usage dictionary that is associated with an OCG or layer. If the OCG is not included in an application event, then the contents in this dictionary serves as pure information.

If the OCG is included in one or more application events, then these settings control also the visibility state of the OCG. The function GetOCG() returns the events and categories which control the layer visibility.

If *UserNamesCount* is greater zero, the dictionary contains also user names. The user names can be accessed with GetOCGUsageUserName().

Rerturn values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetOCGCount

Syntax:

```
SI32 pdfGetOCGCount(
    const PPDF* IPDF) // Instance pointer
```

The function returns the number of OCGs or layers which are available in document.

Return values:

This function cannot fail, the return value is always greater or equal zero.

GetOCGUsageUserName

Syntax:

```
LBOOL pdfGetOCGUsageUserName(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle,      // OCG handle
    UI32 Index,       // Array index of the name
    char* ADDR NameA, // Address of a char* variable
    UI16* ADDR NameW) // Address of a UI16* variable
```

The function returns a user name of a Content Usage dictionary that is associated with an OCG or layer. The functions a pointer of the original string that is stored in the PDF file. Either the Ansi or Unicode string is returned but never both. DynaPDF is the owner of the string. You must not modify or free the string.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetOCHandle

Syntax:

```
SI32 pdfGetOCHandle(
    const IOCG* OC) // Pointer of an OCG or OCMD
```

The function returns the handle of an OCG or OCMD. A pointer of an OCG (Optional Content Group) or OCMD (Optional Content Membership Dictionary) is provided in functions like GetImageObj(), or ParseContent().

Return values:

If *OC* points to an OCMD the return value is an OCMD handle, a value greater or equal 0x40000000. If *OC* points to an OCG the return value is an OCG handle, a value greater or equal zero. If the function fails the return value is a negative error code.

GetOCUINode

Syntax:

```
IOCN* pdfGetOCUINode(
    const PPDF* IPDF,               // Instance pointer
    const IOCN* Node,               // Node pointer or NULL to get the root
    struct TPDFOCUINode* OutNode)   // See below

struct TPDFOCUINode
{
    UI32 StructSize;     // Must be set to sizeof(TOCUINode)
    const char* LabelA;  // Optional label.
    const UI16* LabelW;  // Either the Ansi or Unicode string is set at time but never both.
    UI32 LabelLength;    // Length in characters without the null-terminator.
    IOCN* NextChild;     // If set, the next child node that must be loaded.
    LBOOL NewNode;       // If true, a new child node must be created.
    SI32 OCG;            // Optional OCG handle. -1 if not set -> GetOCG().
};
```

The function retrieves the most important properties of an Optional Content UI node (UI stands for User Interface).

Notice:

> If the PDF file was not already imported with ImportPDFFile() the function ImportOCProperties() must be called to import the global Optional Content Properties.

> UI nodes are part of a layer configuration. Therefore, a layer configuration must be loaded with LoadLayerConfig() before the function can be called.

Optional Content Groups (OCGs) which are returned by this function should be visible in the user interface of a viewer application. All other OCGs should not be shown by default.

UI nodes are stored in a single linked list. To get the pointer of the root node set the parameter *Node* to NULL. The parameter *OutNode* will be ignored in this case but it is required otherwise.

The next node of the root is the return value of this function. If no further node is available the return value will be NULL.

Every node can contain one or more child nodes, an optional label, or a reference to an OCG.

To achieve the same nesting as in Adobe's Acrobat or Reader the nodes must be added to a custom tree control as shown in the following simplified algorithm (Delphi):

```
function GetString(const ValueA: PAnsiChar; const ValueW: PWideChar): String;
begin
// Make never a direct typecast from PAnsiChar to String because this causes a buffer overrun!
if ValueA <> nil then
   Result := String(AnsiString(ValueA))
else
   Result := String(WideString(ValueW));
end;
```

```delphi
// This algorithm is a bit simplified. In a real world application we must store the OCG handles in
// the tree node so that the layer state can be changed. The Delphi tree control supports the user
// defined pointer Data which can be used for this purpose. However, Data is a pointer and therefore
// we must store the OCG handles in a list before we can set the pointer. In addition, a standard
// tree control does not support the features we need to show or hide layers.
// We assume that the PDF file was already opened.
procedure TForm1.LoadLayerTreeNodes(Parent: TTreeNode; Next: IOCN; Level: Integer);
var node: TOCUINode; ocg: TPDFOCG; lbl: String; prnt, saved: TTreeNode;
begin
    if Level > 32 then Exit; // A maximum recursion depth of 32 should be more than enough.
    node.StructSize := sizeof(node);
    ocg.StructSize  := sizeof(ocg);
    saved := Parent;
    while Next <> nil do begin
        node.NextChild := nil;
        node.OCG        := -1;
        Next            := FPDF.GetOCUINode(Next, @node);
        if node.OCG < 0 then begin
            lbl := GetString(node.LabelA, node.LabelW);
            if node.NewNode then
                prnt := LayerTree.Items.AddNode(nil, Parent, lbl, nil, naAddChild)
            else
                prnt := Parent;
            LoadLayerTreeNodes(prnt, node.NextChild, Level +1);
            Parent := saved; // Important! Restore the parent tree node.
        end else begin
            if FPDF.GetOCG(node.OCG, ocg) then begin
                lbl := GetString(ocg.NameA, ocg.NameW);
                if node.NewNode then begin
                    saved  := Parent; // Important! Save the parent tree node.
                    Parent := LayerTree.Items.AddNode(nil, Parent, lbl, nil, naAddChild)
                end else
                    LayerTree.Items.AddNode(nil, Parent, lbl, nil, naAddChild);
            end;
        end;
    end;
end;

procedure TForm1.LoadLayerTree;
var next: IOCN;
begin
    FPDF.ImportOCProperties;            // Import the global Optional Content Properties
    FPDF.LoadLayerConfig(-1);           // Load the default layer configuration
    next := FPDF.GetOCUINode(nil, nil); // Get the root node
    LoadLayerTreeNodes(nil, next, 0);   // Load the children of the root
end;
```

Although the syntax is slightly different among different programming languages, the general algorithm stays the same.

GetOpacity

Syntax:

```
double pdfGetOpacity(
    const PPDF* IPDF) // Instance pointer
```

The function returns the opacity value used to draw the visible appearance of an annotation (requires Acrobat 5 or higher). A value of **1** determines that the annotation is completely opaque. If the value is zero, newly created annotations will be invisible. The opacity flag is not supported by all annotation types.

At time of publication the opacity property is used for annotations only. Later versions of DynaPDF use this property may be for other PDF objects too.

GetOrientation

Syntax:

```
SI32 pdfGetOrientation(
    const PPDF* IPDF) // Instance pointer
```

The function returns the orientation of the current open page if an open page was detected or the default orientation for newly created pages if no open page was detected.

The orientation is measured in degrees; it is also always a multiple of 90 or 0.

```
Default value = 0 (Portrait)
```

GetOutputIntent

Syntax:

```
LBOOL pdfGetOutputIntent(
    const PPDF* IPDF,                 // Instance pointer
    UI32 Index,                       // Array index
    struct TPDFOutputIntent* Intent)  // See below

struct TPDFOutputIntent
{
    UI32  StructSize;          // Must be set to sizeof(TPDFOutputIntent)
    BYTE* Buffer;              // ICC profile buffer (optional)
    UI32  BufSize;             // Buffer size in bytes
    char* InfoA;               // Info string or name of the ICC profile
    UI16* InfoW;               // Info string or name of the ICC profile
    UI32  NumComponents;       // Number of components
    char* OutputConditionA;    // Describtion of the output device
    UI16* OutputConditionW;    // Describtion of the output device
    char* OutputConditionIDA;  // Describtion of the output device
    UI16* OutputConditionIDW;  // Describtion of the output device
    char* RegistryNameA;       // The registry in which OutputConditionID is
    UI16* RegistryNameW;       // defined.
    char* SubType;             // GTS_PDFX, GTS_PDFA1, or ISO_PDFE1
};
```

The function returns an output intent as well as the properties of it. An output intent is an ICC profile that describes the characteristics of the output device. The profile is mostly embedded but certain standards like PDF/X 4p, for example, allow also the usage of non-embedded profiles.

All strings in the structure TPDFOutputIntent can be defined as Unicode or Ansi string (Windows code page 1252) but only one format is present at time.

The parameter *Index* is the array index of the output intent. To determine the number of available intents call GetOutputIntentCount().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetOutputIntentCount

Syntax:

```
SI32 pdfGetOutputIntentCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of available output intents. This function cannot fail.

GetPageAnnot (obsolete)

Syntax:

```
LBOOL pdfGetPageAnnot(
   const PPDF* IPDF, // Instance pointer
   UI32 Index,       // Array index in the page
   struct TPDFAnnotation ADDR Annot) // Structure to be filled
```

The function retrieves the most important properties of an annotation like GetAnnot(). This function is marked as obsolete and should no longer be used. If possible use GetPageAnnotEx() instead. The parameter *Index* must be the array index of the page in which the annotation is used. To enumerate the annotations of a page execute the function in a loop from 0 to GetPageAnnotCount() - 1.

Return Values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetPageAnnotEx

Syntax:

```
LBOOL pdfGetPageAnnotEx(
   const PPDF* IPDF, // Instance pointer
   UI32 Index,       // Array index in the page
   struct TPDFAnnotationEx ADDR Annot) // Structure to be filled
```

The function retrieves the most important properties of an annotation like GetAnnotEx(). The parameter *Index* must be the array index of the page in which the annotation is used. To enumerate the annotations of a page execute the function in a loop from 0 to GetPageAnnotCount() - 1. See GetAnnotEx() for further information.

Return Values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetPageAnnotCount

Syntax:

```
SI32 pdfGetPageAnnotCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of annotations which are used by a page. The page must be opened with EditPage() before the function can be used. The annotations of the page can be accessed with GetPageAnnotEx().

Return values:

If the function succeeds the return value is the number of annotations which are used by a page. If the function fails the return value is a negative error code.

GetPageBBox (Rendering Engine)

Syntax:

```
LBOOL pdfGetPageBBox(
   IPGE* PagePtr,          // Pointer of a page object
   TPageBoundary Type,     // The bounding box that should be returned
   struct TFltRect* BBox) // Out -> The bounding box
```

The function returns a bounding box of a page. The parameter *PagePtr* must be a valid page pointer that was returned by GetPageObject(). The function accesses the page object without any overhead, and hence, is very fast.

Note that all bounding boxes with exception of the media box are optional and maybe not available. When the bounding box is not available the function initializes the parameter *BBox* with zero and returns false.

Remarks:

The functions of the rendering engine should be fast as possible and use less error checking than normal DynaPDF functions. For example, the function does not check whether valid pointers or NULL were passed to the function.

Return values:

When the bounding box is available the return value is 1. When the bounding box is not available the functions 0.

GetPageCoords

Syntax:

```
SI32 pdfGetPageCoords(
   const PPDF* IPDF) // Instance pointer

typedef enum
{
  pcBottomUp = 0,
  pcTopDown  = 1
}TPageCoord;
```

The native coordinate system of the Portable Document Format is bottom-up. However, DynaPDF supports also top-down coordinates to make the usage of the library easier. See also SetPageCoords().

```
Default value = pcBottomUp
```

GetPageCount

Syntax:

```
SI32 pdfGetPageCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of pages of the current PDF file if any.

GetPageField (obsolete)

Syntax:

```
SI32 pdfGetPageField(
   const PPDF* IPDF,        // Instance pointer
   UI32 Index,              // Field index
   struct TPDFField* Field) // Structure to be filled
```

The function returns the most important properties of a field. This function is marked as obsolete, please use GetPageFieldEx() instead. The parameter *Index* must be a valid index to the page's field array. To enumerate the fields of a page execute the function in a loop from 0 to GetPageFieldCount() - 1.

See also GetField().

Return values:

If the function succeeds the return value 1 and the structure *Field* is filled with values. If the function fails the return value is 0.

GetPageFieldCount

Syntax:

```
SI32 pdfGetPageFieldCount(
    const PPDF* IPDF) // Instance pointer
```

The function returns the number of fields used by a page. The page must be opened beforehand with the function EditPage().

Return values:

If the function succeeds the return value is number of fields used by the page. If the function fails the return value is a negative error code.

GetPageFieldEx

Syntax:

```
LBOOL pdfGetPageFieldEx(
    const PPDF* IPDF,            // Instance pointer
    UI32 Index,                  // Field index
    struct TPDFFieldEx* Field) // Structure to be filled
#define PDF_ANNOT_INDEX 0x40000000 // Special flag to indicate that an
                                   // annotation index was passed to the
                                   // function.
```

The function returns the most important properties of a field.

Notice:

The internal organization of fields and annotations were changed in DynaPDF 3.0. Prior versions stored fields and annotations in separate arrays so that the parameter *Index* refered directly into the field array. Due to new features in PDF and to preserve the z-order of overlapping fields and annotations, fields and annotations are now stored in the same array because forms fields are in fact Widget annotations.

To achieve optimal processing speed the fields of a page should now be enumerated as follows:

```
...
    SI32 i, count;
```

```
   TPDFFieldEx f;
   pdfEditPage(PDF, 1);
   // Initialize the structure size
   f.StructSize = sizeof(f);
   // GetPageAnnotCount() returns the number of all annotations, incl.
   // form fields!
   count = pdfGetPageAnnotCount(PDF);
   for (i = 0; i < count; i++)
   {
      // GetPageFieldEx() simply returns false if a non-widget annotation
      // is stored at an array index. The function does also not raise an
      // exception in this case if the flag PDF_ANNOT_INDEX is set.
      // So, there is no need to check the annotation type beforehand.
      if (pdfGetPageFieldEx(i | PDF_ANNOT_INDEX, &f))
      {
         // do something with the data
      }
   }
   pdfEndPage(PDF);
...
```

The old way to enumerate the fields from 0 to GetPageFieldCount() -1 does still work but it is slower because the function must find every field in the array.

See also GetFieldEx().

Return values:

If the function succeeds the return value 1 and the structure *Field* is filled with values. If the function fails the return value is 0.

GetPageHeight

Syntax:

```
double pdfGetPageHeight(
   const PPDF* IPDF) // Instance pointer
```

The function returns the height of the currently open page. If no open page can be detected the return value is the default height which will be used for newly created pages. The page height refers to the media box of a page. The real size is maybe smaller if a crop box is present. The crop takes precedence because it crops the media box.

If SetUseVisibleCoords() was set to true, the function checks whether a cop box is present and returns the size of this box if set. A PDF unit represents 1/72 inch. See also GetBBox().

GetPageLabel

Syntax:

```
LBOOL pdfGetPageLabel(
    const PPDF* IPDF,               // Instance pointer
    UI32 Index,                     // Page label handle or array index
    struct TPDFPageLabel* Label)    // Page label structure

struct TPDFPageLabel
{
    UI32             StartRange;    // Number of the first page in the range. If no further label
                                    // follows, the last page in the range is pdfGetPageCount().
                                    // The first page is denoted by 1.
    TPageLabelFormat Format;        // Number format to be used.
    SI32             FirstPageNum;  // First page number to be displayed in the page label.
                                    // Subsequent pages are numbered sequentially from this value.
    char*            Prefix;        // Optional prefix
    UI32             PrefixLen;     // Length of the prefix in characters
    LBOOL            PrefixUni;     // Prefix is a Unicode string?
};

typedef enum
{
    plfDecimalArabic,    // 1,2,3,4...
    plfUppercaseRoman,   // I,II,III,IV...
    plfLowercaseRoman,   // i,ii,iii,iv...
    plfUppercaseLetters, // A,B,C,D...
    plfLowercaseLetters, // a,b,c,d...
    plfNone
}TPageLabelFormat;
```

The function retrieves the properties of a page label. The parameter *Index* must be the array index of the page label. The number of available page labels can be determned with the function GetPageLabelCount().

A page label is used to construct a format string. The start range identifies the first page to which this label applies. If no further page labels follow, the label is applied to all susequent pages through GetPageCount(). The member *FirstPageNum* represents the numeric portion of the first page label in the range. Subsequent pages are numbered sequentially from this value.

Return values:

In the function succeeds the return value is 1. If the function fails the return value is 0.

GetPageLabelCount

Syntax:

```
SI32 pdfGetPageLabelCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of page labels defined in the document.

GetPageLayout

Syntax:

```
SI32 pdfGetPageLayout(
   const PPDF* IPDF) // Instance pointer

typedef enum
{
   plSinglePage     = 0, // Show one page at time
   plOneColumn      = 1, // Show the pages continuous
   plTwoColumnLeft  = 2, // Two columns, start with left column
   plTwoColumnRight = 3  // Two columns, start with right column
}TPageLayout;
```

The function returns the page layout that is used when opening the document with Adobe's Acrobat.

```
Default value = plOneColumn
```

GetPageMode

Syntax:

```
SI32 pdfGetPageMode(
   const PPDF* IPDF) // Instance pointer

typedef enum
{
   pmUseNone     = 0, // No page mode is applied
   pmUseOutlines = 1, // Show outline tree
   pmUseThumbs   = 2, // Show thumbnails
   pmFullScreen  = 3  // Open the document in full screen mode
}TPageMode;
```

The function returns the page mode that is used when opening the document with Adobe's Acrobat.

GetPageNum

Syntax:

```
SI32 pdfGetPageNum(
   const PPDF* IPDF) // Instance pointer
```

The function returns the page number of the currently open page. If no open page can be detected the return value is a negative error code.

GetPageObject (Rendering Engine)

Syntax:

```
IPGE* pdfGetPageObject(
    const PPDF* IPDF, // Instance pointer
    UI32 PageNum)     // Page number
```

The function returns the pointer of a page object to enable fast access to certain properties of it, e.g. the bounding boxes or the page orientation. This function is mostly used during rendering because it provides fast access to page properties without any unnecessary overhead.

The live time of a page pointer ends when the PDF file in memory will be released, e.g. when CloseFile() or FreePDF() was called.

Return values:

If the function succeeds the return value is a pointer of the page object. If the function fails the return value is NULL. The function can only fail when an invalid page number was passed to the function.

GetPageOrientation (Rendering Engine)

Syntax:

```
SI32 pdfGetPageOrientation(
    IPGE* PagePtr) // Pointer of a page object
```

The function return the orientation of the page in degrees. The function accesses the page object without any overhead, and hence, is very fast.

Remarks:

The functions of the rendering engine should be fast as possible and use less error checking than normal DynaPDF functions. For example, the function does not check whether a valid page pointer or NULL was passed to the function.

GetPageText

Syntax:

```
LBOOL pdfGetPageText(
    const PPDF* IPDF,        // Instance pointer
    struct TPDFStack* Stack) // Operation stack, see below

struct TTextRecordA
{
   float Advance; // Negative values move the cursor to the right
   BYTE* Text;    // Raw text
   UI32  Length;  // Raw text length in bytes
};

struct TTextRecordW
{
    float Advance; // Negative values move the cursor to the right
    UI16* Text;    // Translated Unicode string (not null-terminated!)
```

```
    UI32  Length;   // Length in characters
    float Width;    // String width measured in text space
};

struct TPDFStack
{
    struct TCTM          ctm;           // Pre-multiplied global transf. matrix
    struct TCTM          tm;            // Pre-multiplied text transf. matrix
    double               x;             // Unused -> always 0.0
    double               y;             // Unused -> always 0.0
    double               FontSize;      // Font size measured in text space!
    double               CharSP;        // Current character spacing
    double               WordSP;        // Current word spacing
    double               HScale;        // Current horizontal scaling
    double               TextRise;      // Always 0.0 -> already included in tm
    double               Leading;       // Current leading
    double               LineWidth;     // Current line width
    TDrawMode            DrawMode;      // Text draw mode (see SetDrawMode())
    TPDFColorSpace       FillCS;        // Color space of fill color
    TPDFColorSpace       StrokeCS;      // Color space of stroke color
    UI32                 FillColor;     // Current fill color (text color)
    UI32                 StrokeColor;   // Current stroke color
    void*                BaseObject;    // Internal
    LBOOL                CIDFont;       // See description
    char*                Text;          // Pointer to raw binary text buffer
    UI32                 TextLen;       // Text length in bytes
    struct TTextRecordA* RawKern;       // Raw kerning array
    struct TTextRecordW* Kerning;       // Translated Unicode kerning array
    UI32                 KerningCount;  // Number of kerning records
    float                TextWidth;     // Text width incl. kerning space
    void*                IFont;         // Font object used to print the string
    LBOOL                Embedded;      // If true, the font is embedded
    float                SpaceWidth;    // Measured in text space
    LBOOL                ConvColors;    // See description
    TPDFColorSpace       DestSpace;     // See description
    UI32                 DeleteKerningAt; // See description
    UI32                 FontFlags;     // PDF font flags
    // ------------------- 12 Reserved fields -------------------
    SI32                 Reserved1..12;
    ...
};
```

This function can be used to extract the text of a page or template, or to find a specific text that should be replaced or deleted with the function ReplacePageText() or ReplacePageTextEx().

The structure TPDFStack must be initialized with InitStack() before calling the function the first time. All members of this structure are read only with the exception of the variables in red color. These values can be changed after the structure has been initialized with InitStack(). Do not change the value of another member; otherwise the function causes maybe an access violation!

Structure TPDFStack:

Member	Description
ctm	The matrix represents the current coordinate system (user space). The matrix is already pre-multiplied because the function returns only when a text showing operator was found.
tm	This matrix represents the text coordinate system in which all text properties are calculated. We call this coordinate system text space. To

	enable the calculation of the text position the matrix must be multiplied with the matrix ctm.
x, y	These members are no longer used. All positioning operators are already included in the text transformation matrix tm.
FontSize	The font size measured in text space used to display the text. The value must be transformed to user space to determine the visible font size (see descption below).
FillCS	This is the current color space in which the color value of FillColor is defined. If ConvColor is set to true (default) the value is always equal to DestSpace. If ConvColor is set to false the fill color is still returned in a device color space but the color space can be changed arbitrary often.
	Colors of complex color spaces, such as Separation or DeviceN, are always converted into the alternate color space. If the original or alternate color space is an ICC based color space the color value is returned as is and the color space is set depending on the number of components. The ICC profile is not invoked to calculate the color value in the device color space. Future versions of DynaPDF maybe use the ICC profile to convert the color into device color space.
StrokeCS	This is the current color space in which the color value of StrokeColor is defined. The color space is set in an identical manner like FillCS.
FillColor	The fill color represents the text color if the text draw mode is dmNormal. Depending on the draw mode the fill and stroke color can be invoked to render the text.
StrokeColor	The stroke color can be ignored if the text draw mode is not dmStroke, dmFillStroke, dmStrokeClip or dmFillStrokeClip since it is then not used to render the text.
CIDFont	If true, a CID font is used to render the text. It is not possible to replace a text with ReplacePageText() in this case. However, it is still possible to delete the text. If the text should be replaced use either ReplacePageTextEx() or WriteTextMatrixEx().
DrawMode	The draw mode used to render the text. See SetTextDrawMode() for further information.
Leading	The leading is already included in the text transformation matrix tm. The

value of leading must **not** be considered to calculate the text position.

TextRise	The text rise is already included in the text transformation matrix tm. The value is always 0.0.
WordSP	Current word spacing, info only.
CharSP	Current character spacing, info only.
HScale	Current horizontal scaling, info only.
LineWidth	The current line width; it can be ignored if the text is not stroked.
Text	The raw text line that was found in the content stream. This is a non-null-terminated binary string; it can contain single or multi-byte characters. This member should no longer be used to extract or replace text. Use the member Kerning instead.
TextLen	The string length in bytes, not characters! If TextLen is zero no text was found or an error occurred.
RawKern	The raw kerning array. The text in this array points directly into the source text buffer. The strings are not null-terminated.
Kerning	Already translated UTF-16 Unicode kerning array. A kerning record consists of the members Advance, Text (UTF-16), Length, and Width. The member Advance represents the amount in which the text is moved on the x-axis. Negative values move the cursor to the right, positive to the left. Normal kerning between characters is usually always less than the half space width which can be taken from the structure TPDFStack. This makes it easy to determine whether a space character is emulated with kerning space. Note that negative values move the cursor to right! A positive value of Advance cannot emulate a space character because the cursor is moved to the wrong direction. The parameter Text contains the Unicode text that is **not** null-terminated. Length is the text length in characters. The text width and advance are calculated in text space.

Note that it is not always possible to translate a text to Unicode. The translation will fail if a CID font depends on an external CMap file that is not available. External CMaps should always be loaded with SetCMapDir() before GetPageText() is executed the first time.

It is also possible that a PDF file contains an invalid encoding or ToUnicode CMap. Because the encoding or ToUnicode CMap is only used

	for Copy & Paste operations you cannot assume that the conversion to Unicode is always possible.
KerningCount	The number of kerning records. RawKern and Kerning contain always the same number of records.
TextWidth	The text width measured in text space. This is the width of the entire kerning array incl. kerning space.
IFont	A pointer to the font object used to print the text. The most important properties of the font can be returned with GetFont().
Embedded	If true, the font is embedded. This information is very important if the text should be replaced. The function ReplacePageText() can be used if the font is not embedded and if the member CIDFont is false. Otherwise, the function ReplacePageTextEx() or WriteTextMatrixEx() must be used to replace the text, see the examples on the following pages for further information.
SpaceWidth	The space width can be used to determine whether a space character is emulated at a given position. A default space width is set if the font does not contain a space character. Note that many documents emulate spaces with kerning space. Such documents contain usually no space character. It is usually best to use the half space width to determine whether a space character is emulated. This is especially important if a document uses condensed fonts which contain no space character.
ConvColor	If true (default), all colors are converted to the destination color space DestSpace. The default color space is DeviceRGB. This variable can be changed after the structure has been initialized with InitStack(). If set to false, colors of complex color spaces are still converted into the alternate color space but the color space can then be changed arbitrary often. See also the description of FillCS.
DestSpace	The destination color space in which all colors should be converted. If ConvColor is set to false, the value of this member is ignored. If the destination color space should be DeviceCMYK initialize the members FillColor and StrokeColor with PDF_CMYK(0,0,0,255); which represents black.
DeleteKerningAt	This member can be used in combination with ReplacePageText(). ReplacePageText() deletes or replaces normally always a complete text record but this is often not required especially if a text must be deleted or

replaced in the middle or end of a kerning array. To make text replacement easier it is possible to preserve an arbitrary number of kerning records from deletion. The value of DeleteKerningAt represents the first array index which should be deleted. All kerning records above this index will be deleted too. Take a look into the demo examples/edit_text which is delivered with DynaPDF to determine how this member can be used.

FontFlags

The font flags describe important characteristics of the current font:

- `0x00001 // Fixed pitch font`
- `0x00002 // Serif style`
- `0x00004 // Symbol font`
- `0x00008 // Script style`
- `0x00020 // Non-symbolic font`
- `0x00040 // Italic style`
- `0x40000 // Force Bold (Type1 fonts only)`

External CMaps

A widely used technique to reduce the amount of data that must be stored in a PDF file is the usage of non-embedded CID fonts. CID fonts, whether embedded or not, can depend on external CMaps which must be available at runtime.

To process strings of such fonts correctly DynaPDF must be able to load required CMap files if necessary. Therefore, DynaPDF is delivered with the most important CMap files which are provided by Adobe Systems. These CMaps can be found in the DynaPDF installation directory at /Resource/CMap/. Applications which extract text from PDF files should include these CMaps so that they can be loaded at runtime.

The search path to external CMaps must be set with SetCMapDir() before executing GetPageText() the first time. The function creates a CMap cache that is hold in memory until the PDF instance will be deleted. The search path(s) to external CMap files should be set only one time per PDF instance and one PDF instance should be used to process so many PDF files as possible. This can significantly improve processing speed.

Order of Text records

GetPageText() returns always when a text showing operator was found. That means the returned text represents not a text line. It can be a single character up to a complete text line depending on how the text is stored in the PDF file.

The order in which text is returned is essentially arbitrary. It depends on the file creator whether text is stored in the logical reading order. For example, most PDF drivers convert headers and footers first. Such strings appear then at the beginning of the content stream. All other strings are in turn not necessarily ordered and one text line can be stored in several different text objects.

A text search or text replacement algorithm must correctly handle cases in which a word or sentence is separated into different text objects. In the worst case GetPageText() returns always only a single

character. As long as the text is not rotated it is relatively easy to determine whether a text record lies on the same y-axis, but finding an arbitrary rotated text that is also stored in several different text objects requires further math.

The position of a text object is calculated from the two transformation matrices *ctm* and *tm*. The global transformation matrix *ctm* represents the current coordinate system when a text showing operator was found. The matrix *ctm* is already pre-multiplied because GetPageText() does not return when a new transformation matrix is applied.

The text transformation matrix *tm* represents the text coordinate system in which text properties such as text width, font size, character spacing, word spacing, or the space width are calculated. All text positioning operators are already included in this matrix.

The combination of both matrices represents the final user space in which the text is rendered. Both matrices must be combined to enable the calculation of the text position and orientation (see the examples on the following pages to determine how the matrices must be combined).

Organization of content streams and pages

A PDF page consists of a content stream and a resource array which contains the resources such as fonts, images, and so on which can be used by the page. The content stream contains the PDF operators which paint the contents of a PDF page.

The PDF format supports two object types which support vector graphics and images: ordinary pages and the so called "Form XObjects" which act as a template (we call this object type template here). A template consists in turn of a content stream and a resource array like a page object and it is possible to convert a page to a template. A page object can display an arbitrary number of templates and a template can in turn display arbitrary other templates. It is important to understand that the content of a template is physically stored in another content stream because the function InitStack() prepares only the currently open content stream of the page or template for editing.

Only this content stream can be parsed and edited. Templates which occur in a page or other template must be parsed separately. Because templates can contain other templates it is usually best to parse templates recursively.

However, if texts must be deleted or replaced you must make sure that a template is not edited twice if it occurs in another page or template. Such a duplicate check is strongly required and it must be applied every time a template should be processed.

Whether a page or template contains templates can be determined with GetTemplCount(). Such subsequent templates can then be opened for editing with EditTemplate(). When finish, the template must be closed with EndTemplate().

Organization of text objects

A text object consists of a transformation matrix and the text. Several other properties are taken from the current graphics state such as the font, font size, character spacing, word spacing and so on.

Text objects use a separate coordinate system which is represented by the text transformation matrix *tm*. We call this coordinate system text space. All text properties such as font size, text width and so on are calculated in text space. The PDF format supports also several text positioning operators to decrease the size of a text object. To make the usage of the function easier DynaPDF includes all text positioning operators already in the text transformation *tm*.

The text coordinate system must be transformed to user space by multiplying the text matrix with the current transformation matrix *cm* to enable the calculation of the text position. The combined matrix must be recalculated each time GetPageText() returns a new text object.

As mentioned earlier a content stream is **not** organized into text lines and the order in which text objects occur is essentially arbitrary. A text record can occur in two different formats: as an array or as one coherent text string. The array form enables the definition of kerning between characters in a compact format since PDF viewers ignore any available kerning information in a font resource. The strings in a kerning array lie always on the same text line.

The kerning array is also often used to emulate space characters because word spacing does not work with CID fonts. Most PDF drivers use the same algorithm to format text of single and multi-byte fonts; that is the reason why space characters are very often emulated with kerning space. However, it is quite easy to determine whether a space character is emulated at given position: if the displacement is larger than the half space width we can assume that a space character was emulated at this position. The half space width should be used because the fonts of documents which emulate space characters with kerning space contain often no space character. DynaPDF sets a default space width in this case which can be too large if a condensed font is used.

However, the array form is just one possible format to enable kerning between characters. Due to several reasons the array form is sometimes not used. Many PDF drivers update the text position with text positioning operators instead. This technique produces not only much greater content streams it splits text records also into separate ones. This complicates the identification of word boundaries a lot because each record is returned in a separate GetPageText() call. We need now the coordinates to determine whether the text must be assigned to the same line. If the text is not rotated this is not a big deal but if the coordinate system is rotated or if it contains other transformations some further math is required to determine whether a text record must be assigned to the current line.

We want now take a look into a PDF content stream to determine how an arbitrary text can be stored in a PDF file. The following text can be stored in many different ways and it is important to understand that many variants are possible and exist in real PDF files.

The rendered result of the string "The fox eats the lazy mouse." looks quite normal:

The fox eats the lazy mouse.

However, a PDF driver does not necessarily store this text in one record, there are many possible variants:

```
%This is the easiest variant, one record contains the entire text line.
```

```
%It would be returned in one GetPageText() call as one coherent kerning
%record.
(The fox eats the lazy mouse.)Tj

%This version emulates the spaces with kerning space.
%It would be returned in one GetPageText() call with 6 kerning records.
[(The)-280(fox)-280(eats)-280(the)-280(lazy)-280(mouse.)]TJ

%This version uses PDF positioning operators to emulate spaces.
%It produces 6 separate GetPageText() calls.
(The)Tj
2.8 0 Td
(fox)Tj
2.8 0 Td
(eats)Tj
2.8 0 Td
(the)Tj
2.8 0 Td
(lazy)Tj
2.8 0 Td
(mouse.)Tj
```

In the worst case each text record consists of only one character and it is also possible that the entire text occurs unsorted or combined with other texts which lie on completely different positions than this one. There is not necessarily a logical connection between what you see on screen and what is stored in the PDF file. Especially if a PDF file contains tables the order of text records is sometimes very difficult to understand.

Possible encoding issues

If text must be extracted, deleted, or replaced then it is very important that the text in the PDF file can be converted to Unicode. This conversion is possible if the font uses a standard encoding like WinAnsi or MacRoman, if it contains a ToUnicode CMap, or if it contains PostScript Character names which are listed in the Adobe Glyph List, or if it uses a predefined external CMap and if this CMap is available in one of the CMap search paths (SetSetCMapDir() for further information).

More complicated is the processing of certain European scripts such as Russian, Greek, Czech, and so on. A common technique to process such scripts is to convert the original font to a symbol font to avoid the usage of a CID font (multi-byte font) because the PDF format supports only four pre-defined 8 bit encodings (WinAnsi, MacRoman, MacExpert, and Symbol). The advantage is that 8 bit strings can be stored in the PDF file which results in a smaller file size and the PDF file is still compatible to older Acrobat versions prior 4.0 because CID fonts are supported since PDF 1.3.

The problem is that if the font resource contains no ToUnicode CMap or PostScript character names it is no longer possible to convert the text to Unicode. Depending on how a PDF file was created the encoding is also often not known by the PDF driver, e.g. when converting PCL or AFP files to PDF.

Such PDF files can be viewed and printed correctly but it is not possible to extract human readable strings from them.

How to calculate the absolute string position?

The absolute string position can be calculated from the matrices *ctm* and *tm*. Before the string position can be computed the matrix *tm* must be transformed into user space. This can be done by multiplying the matrices *ctm* and *tm* into another one:

```
TCTM MulMatrix(TCTM &M1, TCTM &M2)
{
   TCTM retval;
   retval.a = M2.a * M1.a + M2.b * M1.c;
   retval.b = M2.a * M1.b + M2.b * M1.d;
   retval.c = M2.c * M1.a + M2.d * M1.c;
   retval.d = M2.c * M1.b + M2.d * M1.d;
   retval.x = M2.x * M1.a + M2.y * M1.c + M1.x;
   retval.y = M2.x * M1.b + M2.y * M1.d + M1.y;
   return retval;
}
```

The usage is as follows:

```
TCTM m = MulMatrix(stack.ctm, stack.tm);
```

Note that the order in which the matrices are multiplied is important; the reversed order would produce an incorrect result.

Now we need a function that transforms a point with the matrix:

```
void Transform(TCTM &M, double &x, double &y)
{
   double ox = x;
   x = ox * M.a + y * M.c + M.x;
   y = ox * M.b + y * M.d + M.y;
}
```

The text position can now easily calculated as follows:

```
// Get the text matrix in user space
TCTM m = MulMatrix(stack.ctm, stack.tm);
double x = 0.0;
double y = 0.0;
Transform(m, x, y);
```

How to caluculate the font size?

The font size that is provided in the structure TPDFStack is measured in text space. If you want to know the visible font size then the value must be transformed to user space:

First, we must multiply the matrices *ctm* and *tm* with MulMatrix() as shown above and pass the resulting matrix to GetScaleY() as described below; the return value is the scaling factor on the y-axis. Finally, the font size must be multiplied with the scaling factor:

```cpp
// Distance between two points
double CalcDistance(double x1, double y1, double x2, double y2)
{
   double dx = x2-x1;
   double dy = y2-y1;
   return sqrt(dx*dx + dy*dy);
}
// Scaling factor of the y-axis
double GetScaleY(TCTM &M)
{
   double x1 = 0.0;
   double y1 = 0.0;
   double x2 = 0.0;
   double y2 = 1.0;
   Transform(M, x1, y1);
   Transform(M, x2, y2);
   if (y1 > y2)
      return -CalcDistance(x1, y1, x2, y2);
   else
      return CalcDistance(x1, y1, x2, y2);
}

TCTM   m = MulMatrix(stack.ctm, stack.tm); // User space matrix
double fs = stack.FontSize * GetScaleY(m);  // Real font size
```

How to calculate the rotation angle?

If you want to know whether the string is rotated then use the function TransRotation() to calculate the rotation angle in radians. Note that this function requires again the pre-multiplied matrix in user space.

```cpp
double TransRotation(TCTM &M)
{
   double x1 = 0.0; double x2 = 1.0;
   double y1 = 0.0; double y2 = 0.0;
   Transform(M, x1, y1);
   Transform(M, x2, y2);
   return atan2(y2-y1, x2-x1);
}
```

How to find and replace text in a page?

As mentioned in the previous sections text replacement or text search algorithms are not easy to develop because many things must be considered to get suitable results. To make the development easier DynaPDF is delivered with several example projects which are available in C++, Delphi, VB

.Net, and C#. You should take a look into the examples text_extraction, text_extraction2, edit_text, and text_search to determine how the function GetPageText() can be used.

If you need only a text search algorithm it is better to use the content parser of DynaPDF directly because it is faster than GetPageText() (see the example text_search for further information).

The class CPDFEditText() (used in the example edit_text) contains already a rather complex and complete text replacement algorithm that demonstrates how the functions ReplacePageText() and ReplacePageTextEx() can be used. You should try to understand how this algorithm works so that you can extend it. This class demonstrates especially how space characters can be identified and how they must be handled when replacing texts. However, note that PDF files are generally not designed to edit existing contents. Existing text should only be replaced if there is no other way to achieve the same result or if only minor changes must be applied, e.g. replacing a misspelled word.

Remarks:

GetPageText() parses the content stream of the currently open page or template as it is at time of executing the function. The content stream contains all operators and values which were added beforehand with DynaPDF functions incl. the contents of imported PDF files. If texts should be replaced or deleted it is usually best to process imported page(s) before adding new contents.

Return values:

If the function succeeds and if further records are available the return value is 1. If the function fails or if no further records are available the return value is 0.

If a content stream contains no text the return value is 0 and the members *TextLen* and *KerningCount* are set to 0. If a content stream contains only one text record the return value is 0 that means that no further records are available but the members *TextLen* and *KerningCount* are set to values greater 0.

GetPageWidth

Syntax:

```
double pdfGetPageWidth(
   const PPDF* IPDF) // Instance pointer
```

The function returns the width of the currently open page. If no open page can be detected the return value is the default width which will be used for newly created pages. The page width refers to the media box of a page. The real size is maybe smaller if a crop box is present. The crop takes precedence because it crops the media box.

If SetUseVisibleCoords() was set to true, the function checks whether a cop box is present and returns the size of this box if set. A PDF unit represents 1/72 inch. See also GetBBox().

GetPDFVersion

Syntax:

```
SI32 pdfGetPDFVersion(
    const PPDF* IPDF) // Instance pointer
```

The function returns the PDF version as specified in the member TPDFVersion. The return value is defined as SI32 (signed 32 bit integer) due to historical reasons. Make a typecast to TPDFVersion to get the correct value.

GetPDFVersionEx() can be called to determine the underlying PDF standard as well as the major and minor version.

GetPDFVersionEx

Syntax:

```
LBOOL pdfGetPDFVersionEx(
    const PPDF* IPDF,                // Instance pointer
    struct TPDFVersionInfo* Value) // Required
struct TPDFVersionInfo
{
    UI32        StructSize;     // Must be set to sizeof(TPDFVersionInfo)
    UI32        Major;          // PDF major version, e.g. 1
    UI32        Minor;          // PDF minor version, e.g. 7
    const char* MainVer;        // Set if the file conforms to a specific PDF standard, e.g. "PDF/X
                                // 1a:2003", "PDF/A 3b", and so on.
    const char* SubVer;         // e.g. "ZUGFeRD 2.0", "Factur-X", and so on.
    // PDF/A
    const char* PDFAConformance; // e.g. A, B, E, F, U, or UA.
    UI32        PDFAVersion;    // If greater zero, this is a PDF/A file.
    // PDF/E
    const char* PDFEVersion;    // e.g. PDF/E-1
    // PDF/VT
    const char* PDFVTModDate;   // Modification date, e.g. 2016-05-07T15:42:23+01:00
    const char* PDFVTVersion;   // e.g. PDF/VT-2
    // PDF/X
    const char* PDFXConformance; // e.g. PDF/X-1a:2001 or PDF/X-1a:2003. PDF/X 3 and above do not set
                                // this key anymore.
    const char* PDFXVersion;    // e.g. PDF/X-1:2001, PDF/X-1:2003, and so on.
    // ZUGFeRD, Factur-X, XRechnung
    const char* FXConfLevel;    // ConformanceLevel
    const char* FXDocName;      // DocumentFileName
    const char* FXDocType;      // DocumentType
    const char* FXVersion;      // Version
    SI32        VersionConst;   // If >= 0 this is the TPDFVersion constant as integer. If the
                                // version is not supported by DynaPDF, this member is set to -1.
};
```

The function retrieves the major and minor version of the PDF document as well as the underlying PDF standard(s), if any.The member *StructSize* must be set to sizeof(TPDFVersionInfo) before the function be called.

A PDF file can be compatible to multiple standards like PDF/A and PDF/X. Therefore, more than one version can be set at time.

ZUGFeRD, Factur-X, XRechnung

ZUGFeRD, Factur-X, and XRechnung invoices depend on an embedded XML invoice. The name of this file is set to *FXDocName*. This name can be passed to FindEmbeddedFile() to get a handle of this file. If the file cannot be found make sure that embedded files were imported either via ImportPDFFile() or ImportCatalogObjects().

If these functions were called then check the import flags. The flag *ifEmbeddedFiles* must be set to get embedded files imported. See SetImportFlags() for further information.

Note also that ZUGFeRD 2.1 and Factur-X use the same XMP version identifiers. Therefore, it is not possible to distiguish between these versions without parsing the embedded XML invoice.

VersionConst

The member *VersionConst* is set if DynaPDF supports the file version. Note that a PDF version consists of a version constant plus optional flags. See enum TPDFVersion for further information.

The value is set to -1 if the version is not supported by DynaPDF.

Not implemented

The function does not support all PDF versions currently on the marked. Not recognized are PDF/VCR and PDF/R (PDF Raster) files.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetPrintSettings

Syntax:

```
LBOOL pdfGetPrintSettings(
  const PPDF* IPDF,
  struct TPDFPrintSettings* Settings)

struct TPDFPrintSettings
{
    TDuplexMode DuplexMode;          // See below
    SI32  NumCopies;                 // -1 means not set. The maximum value is 5
    SI32  PickTrayByPDFSize;         // -1 means not set. 0 == false, 1 == true
    // If set, the array contains PrintRangesCount * 2 values. Each pair
    // consists of the first and last page of the sub-range. The first page
    // in the PDF file is denoted by 0.
    UI32*         PrintRanges;
    UI32          PrintRangesCount;  // Number of ranges
    TPrintScaling PrintScaling;      // dpmNone means not set
    /* 9 reserved fields follow*/
};

typedef enum
{
    dpmNone,         // Use the default value of the viewer
    dpmSimplex,
    dpmFlipShortEdge,
    dpmFlipLongEdge
}TDuplexMode;
```

The function retrieves the currently defined print settings of the document. The print settings are used to initialize the print dialog in a viewer application. See also SetPrintSettings().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetPtDataArray

Syntax:

```
LBOOL pdfGetPtDataArray(
  const IPTD* PtData, // Pointer of Point Data dictionary
  UI32 Index,         // Array index
  char** DataType,    // Out -> Data type as string
  float** Values,     // Out -> Pointer of float array
  UI32* ValCount)     // Out -> Number of values in the array
```

The function returns a Point Data dictionary. A point data dictionary specifies points in 2D space for geospatial measures. The parameter DataType specifies the type of points, such as LAT for latitude in degrees, LON for longitute in degrees, or ALT for altitude in metres.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetPtDataObj

Syntax:

```
LBOOL pdfGetPtDataObj(
    const IPTD* PtData, // Pointer of Point Data dictionary
    char** Subtype,      // Out -> Should be Cloud
    UI32* NumArrays)     // Out -> Number of point data arrays
```

The function returns the main properties of a Point Data dictionary. A point data dictionary contains one or more point data arrays. The point data arrays can be accessed with GetPtDataArray().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetRelFileNode

Syntax:

```
LBOOL pdfGetRelFileNode(
    const IRFN* IRF,           // Instance pointer
    struct TPDFRelFileNode* F, // see below
    LBOOL Decompress)          // Decompress the embedded file?
```

```
struct TPDFRelFileNode
{
    UI32              StructSize; // Must be set to sizeof(TPDFRelFileNode).
    const char*       NameA;      // Name of this file spcification.
    const UI16*       NameW;      // Either the Ansi or Unicode name is set.
    struct TPDFFileSpec EF;       // Embedded file.
    IRFN*             NextNode;   // Next node if any.
};
```

The function retrieves the properties of a related files node. An embedded file can contain a related files array that contains additional files related to the base file. For example, the base file could be a composite image, and the related files array could contain the separations for every color channel.

Related files are stored in as a single linked list. The member *NextNode* contains a pointer to the next file, if any. The member *StructSize* must be set to sizeof(TPDFRelFileNode) before the function can be called.

The structure TPDFFileSpec is described in detail at GetEmbeddedFile().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetResetAction

Syntax:

```
LBOOL pdfGetResetAction(
    const PPDF* IPDF,                     // Instance pointer
    UI32 Handle,                          // Handle of a ResetFormAction
    struct TPDFResetFormAction* Value) // See below

struct TPDFResetFormAction
{
    UI32        StructSize;    // Must be set to sizeof(TResetFormAction)
    const void** Fields;        // Array of field pointers -> GetFieldEx2().
    UI32        FieldsCount;   // Number of fields in the array.
    LBOOL       Include;       // If true, the fields in the array must be
                               // reset. If false, these fields must be
                               // excluded.
    SI32        NextAction;    // -1 or next action handle to be executed if any
    TActionType NextActionType; // Only set if NextAction is >= 0.
};
```

The function returns the properties of a Reset Form Action. The member *StructSize* must be initialized with the structure size before the function can be called.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetResolution

Syntax:

```
SI32 pdfGetResolution(
    const PPDF* IPDF) // Instance pointer
```

The function returns the resolution in DPI (Dots per Inch), in which images are stored by DynaPDF. The value will be ignored if the property *SaveNewImageFormat* was set to false.

```
Default value = 150 (DPI)
```

GetSaveNewImageFormat

Syntax:

```
SI32 pdfGetSaveNewImageFormat(
    const PPDF* IPDF) // Instance pointer
```

If false (0), images are not downscaled if necessary to achieve the specified resolution (see SetResolution()).

```
Default value = 1 (true)
```

GetSeparationInfo

Syntax:

```
LBOOL pdfGetSeparationInfo(
    const PPDF* IPDF,         // Instance pointer
    char* ADDR Colorant,      // Colorant name
    TExtColorSpace ADDR CS) // Separation, DeviceN, or NChannel
```

The function returns the separation info stored in the current open page. The colorant name is returned UTF-8 Unicode format.

Return values:

If the function succeeds the return value is 1. If the function fails or if the page contains no separation info the return value is 0.

GetSigDict

Syntax:

```
LBOOL pdfGetSigDict(
    const PPDF* ISignature,       // Pointer of a signature dictionary
    struct TPDFSigDict* SigDict) // see below

struct TPDFSigDict
{
    UI32            StructSize;     // Must be set to sizeof(TPDFSigDict).
    const UI32*     ByteRange;      // ByteRange -> Byte offset followed by the corresponding length.
                                    // The byte ranges are required to create the digest. The values
                                    // are returned as is. Check whether the offsets and length
                                    // values are valid. There are normally at least two ranges.
                                    // Overlapping ranges are not allowed! Any error breaks
                                    // processing and the signature should be considered as invalid.
    UI32            ByteRangeCount; // The number of Offset / Length pairs.
    const BYTE*     Cert;           // X.509 Certificate when SubFilter is adbe.x509.rsa_sha1.
    UI32            CertLen;        // Length in bytes
    const UI32*     Changes;        // If set, an array of three integers that specify changes to the
                                    // document that have been made between the previous signature
                                    // and this signature in this order: the number of pages altered,
                                    // the number of fields altered, and the number of fields filled
                                    // in.
    const char*     ContactInfoA;  // Optional contact info string
    const UI16*     ContactInfoW;  // Optional contact info string
    const BYTE*     Contents;      // The signature. This is either a DER encoded PKCS#1 binary data
                                    // object or a DER-encoded PKCS#7 binary data object depending on
                                    // the used SubFilter.
    UI32            ContentsSize;  // Length in bytes.
    const char*     Filter;        // The name of the security handler, usually Adobe.PPKLite.
    const char*     LocationA;     // Optional location of the signer
    const UI16*     LocationW;     // Optional location of the signer
    const char*     SignTime;      // Date/Time string
    const char*     NameA;         // Optional signers name
    const UI16*     NameW;         // Optional signers name
    UI32            PropAuthTime;  // The number of seconds since the signer was last authenticated.
    const char*     PropAuthType;  // Optional -> The method that shall be used to authenticate the
                                    // signer. Valid values are PIN, Password, and Fingerprint.
    const char*     ReasonA;       // Optional reason
    const UI16*     ReasonW;       // Optional reason
    SI32            Revision;      // The version of the signature handler that was used to create
                                    // the signature.
    const char*     SubFilter;     // A name that describes the encoding of the signature value.
```

```
                                    // This should be adbe.x509.rsa_sha1, adbe.pkcs7.detached, or
                                    // adbe.pkcs7.sha1.
    SI32          Version;          // The version of the signature dictionary format.
};
```

The function returns the properties of a signature dictionary. This dictionary is only present if an imported PDF file was digitally signed. Note that the byte ranges refer to the original file that contained the signature. With the information of this dictionary it is possible to validate the signature with a cryptographic library like the Windows Crypt API, OpenSSL or similar libraries.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetSpaceWidth

Syntax:

```
double fntGetSpaceWidth(
    const PPDF* IFont, // Pointer to internal font object
    double FontSize)   // Font size to compute the space width
```

The function returns the width of the space character of the font depending on the font size. If the font contains no space character a default size is used to compute the space width.

The parameter *IFont* must be a valid pointer to a PDF font. Such a pointer is returned by GetPageText(), EnumDocFonts(), or by the content parser (see ParseContent()).

Return values:

If the function succeeds the return value is the space width. If the function fails the return value is zero. This function does not use the exception handling of DynaPDF. No error message is set on failure.

GetStrokeColor

Syntax:

```
UI32 pdfGetStrokeColor(
    const PPDF* IPDF) // Instance pointer
```

The function returns the current stroke color. The returned color value is not converted to the active color space. For example, if the color space will be changed with SetColorSpace(), the color value is still the same. Colors must be correctly defined in the current color space.

```
Default value = 0 (Black)
```

GetSubmitAction

Syntax:

```
LBOOL pdfGetSubmitAction(
    const PPDF* IPDF,                       // Instance pointer
    UI32 Handle,                            // SubmitFormAction handle
    struct TPDFSubmitFormAction* Value) // See below

struct TPDFSubmitFormAction
{
    UI32        StructSize;     // Must be set to sizeof(TSubmitFormAction)
    const char* CharSet;        // Optional charset in which the form should be submitted.
    const void** Fields;        // Array of field pointers -> GetFieldEx2().
    UI32        FieldsCount;    // Number of fields in the array.
    TSubmitFlags Flags;         // Various flags, see CreateSubmitAction().
    const char* URL;            // The URL of the script that will process the
                                // submission.
    SI32        NextAction;     // -1 or next action handle to be executed if any
    TActionType NextActionType; // Only set if NextAction is >= 0.
};
```

The function returns the properties of a Submit Form Action. The member *StructSize* must be initialized with the structure size before the function can be called.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetSysFontInfo

Syntax:

```
SI32 pdfGetSysFontInfo(
    const PPDF* IPDF,        // Instance pointer
    UI32 Handle,             // Next font handle or 0 for the first call
    struct TPDFSysFont* Out) // Out -> See below

struct TPDFSysFont
{
    UI32            StructSize;     // Must be set to sizeof(TPDFSysFont)
    TFontBaseType   BaseType;       // Font type
    const char*     CIDOrdering;    // OpenType CID fonts only
    const char*     CIDRegistry;    // OpenType CID fonts only
    UI32            CIDSupplement;  // OpenType CID fonts only
    UI32            DataOffset;     // Data offset
```

```
    const UI16*         FamilyName;        // Family name
    const char*         FilePathA;         // Font file path (Ansi string on Windows, UTF-8 otherwise
    const UI16*         FilePathW;         // Font file path
    UI32                FileSize;          // File size in bytes
    TEnumFontProcFlags  Flags;             // Bitmask
    const UI16*         FullName;          // Full name
    UI32                Length1;           // Length of the clear text portion of a Type1 font
    UI32                Length2;           // Length of the eexec encrypted portion of a Type1 font
    const char*         PostScriptNameA;   // Postscript mame
    const UI16*         PostScriptNameW;   // Postscript mame
    SI32                Index;             // Zero based font index if stored in a TrueType collection
    LBOOL               IsFixedPitch;      // If true, the font is a fixed pitch font
    TFStyle             Style;             // Font style
    TUnicodeRange1      UnicodeRange1;     // Bitmask
    TUnicodeRange2      UnicodeRange2;     // Bitmask
    TUnicodeRange3      UnicodeRange3;     // Bitmask
    TUnicodeRange4      UnicodeRange4;     // Bitmask
};
```

The function returns the most important properties of a system font. The member *StructSize* must be set to sizeof(TPDFSysFont) before the function can be called.

If you want to know how many fonts will be returned then call EnumHostFonts() with the parameter *EnumProc* set to NULL beforehand. The function returns the number of available fonts in this case.

However, also if the number of host fonts is known, it is not possible to loop over the available fonts from 0 to FontCount - 1 since the parameter *Handle* represents no array index.

In the first call set the parameter *Handle* to 0 or *Out* to NULL. In this case, the function returns the handle of the first font file. This value must be passed to the next call until the return value is zero.

Note that if *Out* is set in the first call, then the first font will already be returned.

Since the return value could be zero if only one system font is available, the caller must check whether *FamilyName* is non-NULL to determine whether a font was returned (all members are initialized with zero in this case). The family name is always set if a font was returned.

Example (C#):

```csharp
// This code adds the full names to a list box.
int next;
TPDFSysFont f = new TPDFSysFont();
next = 0;
next = pdf.GetSysFontInfo(next, ref f);
if (next >= 0 && f.FamilyName != null)
{
    listBox1.Items.Add(f.FullName);
}
while (next > 0)
{
    next = pdf.GetSysFontInfo(next, ref f);
    if (next >= 0)
    {
        listBox1.Items.Add(f.FullName);
    }
}
```

The function returns the fonts in the current font search order. That means if the search order is {fbtTrueType, fbtOpenType, fbtType1, fbtStdFont}, for example, then the function returns first all TrueType fonts sorted in ascending order, then all OpenType fonts, and so on.

Since standard fonts are no system fonts, these fonts are not returned.

Return values:

If the function succeeds the return value is greater or equal zero and the parameter *Out* was initialized or filled with values. If the function fails the return value is a negative error code. The parameter *Out* is **not** initialized in this case.

GetTabLen

Syntax:

```
SI32 pdfGetTabLen(
   const PPDF* IPDF) // Instance pointer
```

The function returns the tabulator length in spaces that is used to emulate tabulators during text formatting (see WriteFText() for further information).

Because tabulators are emulated with spaces they have no fixed with. The width of a tabulator depends on the width of the space character of the active font.

```
Default value = 3
```

GetTemplCount

Syntax:

```
SI32 pdfGetTemplCount(
   const PPDF* IPDF) // Instance pointer
```

The function returns the number of templates used by a page. The page must be opened with EditPage() beforehand.

Return values:

If the function succeeds the return value is the number of templates. If the function fails the return value is a negative error code.

GetTemplHandle

Syntax:

```
SI32 pdfGetTemplHandle(
   const PPDF* IPDF) // Instance pointer
```

The function returns the handle of the current open template or a negative error code on failure.

GetTemplHeight

Syntax:

```
double pdfGetTemplHeight(
    const PPDF* IPDF, // Instance pointer
    SI32 Handle)      // Template handle
```

The function returns the height of a template. The parameter *Handle* must be a valid template handle.

Return values:

If the function succeeds the return value is the height of the template. If the function fails the return value is a negative error code.

GetTemplWidth

Syntax:

```
double pdfGetTemplWidth(
    const PPDF* IPDF, // Instance pointer
    SI32 Handle)      // Template handle
```

The function returns the width of a template. The parameter *Handle* must be a valid template handle.

Return values:

If the function succeeds the return value is the width of the template. If the function fails the return value is a negative error code.

GetTextDrawMode

Syntax:

```
SI32 pdfGetTextDrawMode(
    const PPDF* IPDF) // Instance pointer
typedef enum
{
    dmNormal        = 0,
    dmStroke        = 1,
    dmFillStroke    = 2,
    dmInvisible     = 3,
    dmFillClip      = 4,
    dmStrokeClip    = 5,
    dmFillStrokeClip = 6,
    dmClipping      = 7
}TDrawMode;
```

The function returns the text draw mode. Se draw modes are described in detail under SetTextDrawMode().

GetTextFieldValue

Syntax:

```
LBOOL pdfGetTextFieldValue(
   const PPDF* IPDF,            // Instance pointer
   UI32 AField,                 // Field handle
   char* ADDR Value,            // Field value
   LBOOL ADDR ValIsUnicode,     // If true, Value is a Unicode string
   char* ADDR DefValue,         // Default value
   LBOOL ADDR DefValIsUnicode)  // If true, DeValue is a Unicode string
```

The function returns the value and default of a text field. Both values can be in Ansi or Unicode format (UTF-16).

Return value:

If the function succeeds the return value is 1, if the function fails the return value is 0.

GetTextRect

Syntax:

```
SI32 pdfGetTextRect(
   const PPDF* IPDF,    // Instance pointer
   double ADDR PosX,    // X-Coordinate of the rectangle
   double ADDR PosY,    // Y-Coordinate of the rectangle
   double ADDR Width,   // Width of the rectangle
   double ADDR Height)  // Height of the rectangle
```

The function retrieves the bounding rectangle to output formatted text. No parameter of the function must be NULL. See also SetTextRect(), WriteFText().

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetTextRise

Syntax:

```
double pdfGetTextRise(
   const PPDF* IPDF) // Instance pointer
```

The function returns the current text rise used to output text. Text rise specifies the distance, in unscaled text space units, to move the baseline up or down from its default location. Positive values of text rise move the baseline up. Adjustments to the baseline are useful for drawing superscripts or subscripts. The default location of the baseline can be restored by setting the text rise to 0. The figure below illustrates the effect of the text rise. Text rise always applies to the vertical coordinate in text space. See also SetTextRise().

The text moves around the baseline.

```
Default value = 0
```

GetTextScaling

Syntax:

```
double pdfGetTextScaling(
    const PPDF* IPDF) // Instance pointer
```

The function returns the current value of horizontal text scaling. The scaling value adjusts the width of glyphs by stretching or compressing them in the horizontal direction. Its value is specified as a percentage of the normal width of the glyphs, with 100 being the normal width. See also SetTextScaling().

```
Default value = 100
```

Value = 100	Word
Value = 50	WordWord

GetTextWidth

Syntax:

```
double pdfGetTextWidth(
    const PPDF* IPDF,  // Instance pointer
    const char* AText) // Null-terminated text string
```

The function computes the width of a text string in horizontal writing mode and the height in vertical writing mode. The graphics state parameters character spacing, word spacing, text scaling, and the current font size are all considered. A font must be set before this function can be used.

The function computes the visible width of the string. This is the width **excluding** the last character spacing. The bounding box of the string is the text width **plus** the current character spacing. This behaviour must be taken into account when writing right aligned text with WriteFText(). Due to certain formatting rules WriteFText() must use the real bounding box of the text.

Visible Text Width (character spacing = 5.0)

C h a r a c t e r   S p a c i n g

Real Bounding Box (character spacing = 5.0)

C h a r a c t e r   S p a c i n g

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

CJK to Unicode code pages are supported by the Ansi version only because CJK strings must be converted to Unicode beforehand. These conversion algorithms are available in the Ansi version only. However, native CJK character sets are also supported with the wide string version in combination with a CJK font. See SetFont() for further information.

Return values:

If the function succeeds the return value is the string width in un-scaled units. If the function fails the return value is a negative error code.

GetTextWidth (Font API)

Syntax:

```
double fntGetTextWidth(
  const void* IFont, // Pointer of the font object
  const BYTE* Text,  // PDF Text string
  UI32 Len,          // Length of the string
  float CharSpacing, // Current character spacing
  float WordSpacing, // Current word spacing
  float TextScale)   // Current text scaling
```

The function can be used to calculate the width of a sub string that was returned by the content parser (see ParseContent() for further information). The parameter IFont must be a valid pointer of a font object that was provided in the TSetFont callback function. Note that this function is optimized for speed and does not use the normal error handling of DynaPDF. The parameters CharSpacing, WordSpacing, and TextScale must be taken from the current graphics state.

The function returns the real text width of a string:

Visible Text Width (character spacing = 5.0)

C h a r a c t e r   S p a c i n g

Real Text Width (character spacing = 5.0)

C h a r a c t e r   S p a c i n g

Return values:

If the function succeeds the return value is the string width measured in text space. If the parameter IFont is set NULL the return value is 0.0.

GetTextWidthEx

Syntax:

```
double pdfGetTextWidthEx(
   const PPDF* IPDF,  // Instance pointer
   const char* AText, // Text string
   UI32 Len)          // Length of the string in characters
```

The function computes the width of a text string. The graphics state parameters character spacing, word spacing, text scaling, and the current font size are all considered. A font must be set before this function can be used.

The function computes the visible width of the string. This is the width **excluding** the last character spacing. The bounding box of the string is the text width **plus** the current character spacing. This behaviour must be taken into account when writing right aligned text with WriteFText(). Due to certain formatting rules WriteFText() uses the bounding box of a text string to output text.

Visible Text Width (character spacing = 5.0)

C h a r a c t e r   S p a c i n g

Real Bounding Box (character spacing = 5.0)

C h a r a c t e r   S p a c i n g

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

CJK to Unicode code pages are supported by the Ansi version only because CJK strings must be converted to Unicode beforehand. These conversion algorithms are available in the Ansi version only. However, native CJK character sets are supported by the Unicode version in combination with a CJK font. See SetFont() for further information.

Return values:

If the function succeeds the return value is the string width in un-scaled units. If the function fails the return value is a negative error code.

GetTransparentColor

Syntax:

```
UI32 pdfGetTransparentColor(
   const PPDF* IPDF) // Instance pointer
```

The function returns the transparent color value that is used for newly inserted images.

```
Default value = 0xFFFFFF (RGB White)
```

GetTrapped

Syntax:

```
LBOOL pdfGetTrapped(
   const PPDF* IPDF) // Instance pointer
```

The function returns the trapped key of the document. The default value is false. However, the real default value is unknown that means the key is not written to the file. The trapped key must be set with SetTrapped() if it should written to the file.

GetTypoLeading

Syntax:

```
double pdfGetTypoLeading(
   const PPDF* IPDF) // Instance pointer
```

The function returns the typographic leading or line height of a font. This is sTypoAscender – sTypoDescender + sTypoLineGap of the OS/2 table of TrueType and OpenType fonts. The value is scaled to the current font size. Type1 fonts do not support these metrics.

The function can be called after a font was activated in the graphics state, e.g. with SetFont(), SetFontEx(), SetCIDFont(), or ChangeFont().

If typographic metrics are not available the return value is Ascent – Descent. If the value is smaller than the font size then the font size is returned.

Return values:

If the function succeeds the return value is the typographic leading of the active font, this is a value greater zero. If the function fails the return value is a negative error code.

GetURIAction

Syntax:

```
LBOOL pdfGetURIAction(
   const PPDF* IPDF,                  // Instance pointer
   UI32 Handle,                       // Action handle
   struct TPDFURIAction* Action) // see below

struct TPDFURIAction
{
   UI32       StructSize;    // Must be set to sizeof(TPDFURIAction)
   const char* BaseURL;      // Optional, if defined in the Catalog object.
   LBOOL      IsMap;         // A flag specifying whether to track the mouse position
                             // when the URI is resolved: e.g. http://test.org?50,70.
   const char* URI;          // Uniform Resource Identifier.
   SI32       NextAction;    // -1 or next action handle to be executed if any.
   TActionType NextActionType; // Only set if NextAction is >= 0.
};
```

The function retrieves the properties of a URI action. The member StructSize must be set to sizeof(TPDFURIAction) before the function can be called.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetUseExactPwd

Syntax:

```
LBOOL pdfGetUseExactPwd(
    const PPDF* IPDF) // Instance pointer
```

If the property *UseExactPwd* is false, an encrypted PDF file can always be decrypted, if either the open or owner password in the file is an empty string. If true, the open or owner password must be known to open the PDF file (see also SetUseExactPwd()).

```
Default value = 1 (true)
```

Remarks:

If your application should allow the modification of encrypted PDF files, you may check the access permissions to grant user rights, if the file was opened with the open password instead of the owner password (see also GetUserRights()).

Due to the license agreement of Adobe, all manufacturers of applications which make the treatment of encrypted PDF files possible, must respect the access permissions of a PDF file, if the file was opened with the open password.

Only if the file was opened with the owner password, all rights should be granted. See *PDF Reference 1.5* for further information.

The property *UseExactPwd* should be true, if the application is a commercial software product.

GetUseGlobalImpFiles

Syntax:

```
LBOOL pdfGetUseGlobalImpFiles(
    const PPDF* IPDF) // Instance pointer
```

The property specifies whether import files should be loaded permanent into memory, e.g. to enable splitting of large PDF files. An open import file will not be closed when CloseFile() or FreePDF() is called when the property is true. See also SetUseGlobalImpFiles().

```
Default value = 0 (false)
```

GetUserRights

Syntax:

```
SI32 pdfGetUserRights(
    const PPDF* IPDF) // Instance pointer

typedef SI32 TRestrictions;
#define rsDenyNothing      0x00000000 // Editing is not restricted
#define rsDenyAll          0x00000F3C // Deny anything
#define rsPrint            0x00000004 // Deny printing
#define rsModify           0x00000008 // Deny modification of contents
#define rsCopyObj          0x00000010 // Deny copying of contents
#define rsAddObj           0x00000020 // No commenting
/* 128/256 bit only -> not meaningful with 40 bit encryption */
#define rsFillInFormFields 0x00000100 // requires rsModify + rsAddObj
#define rsExtractObj       0x00000200 // Extract text, images, vector data
#define rsAssemble         0x00000400 // Text access for screen readers
#define rsPrintHighRes     0x00000800 // Disable high res. printing
#define rsExlMetadata      0x00001000 // Metadata streams are unencrypted?
#define rsEmbFilesOnly     0x00002000 // If set, rsExlMetadata is set too
```

The function returns the encryption flags of the last imported PDF file or **-1** if the file was not encrypted. These flags must be considered when opening an encrypted PDF with user privileges, that is, when the open password was used to decrypt the file. The print flags rsPrint and rsPrintHighRes can be ignored for editing, but the file must be encrypted again with the same encryption flags.

The encryption flags are described in detail at CloseFileEx(). If no file was imported beforehand the return value is zero. Use GetInIsEncrypted() to determine whether the file was encrypted.

GetUserUnit

Syntax:

```
float pdfGetUserUnit(
    const PPDF* IPDF) // Instance pointer
```

The function returns the user unit of the current open page. A user unit acts like a scaling factor. The page format and all page coordinates are multiplied with this factor in a viewer application. The default size of a PDF unit is 1/72 inch and the default user unit is 1.0. User units can be useful if the page format would be too large to be expressed in standard PDF units. The largest page format in PDF is limited to 14400 units or 200 inches. This limit can be extended with the user unit.

The largest value that is supported is 75.0 which results in a maximum page format of 15,000 x 15,000 inches or 1,800,000 units.

GetUseStdFonts

Syntax:

```
LBOOL pdfGetUseStdFonts(
    const PPDF* IPDF) // Instance pointer
```

The function returns 1 (true) if the 14 standard fonts are enabled. PDF viewer applications support 14 standard fonts, these fonts are not embedded by default, also if the parameter Embed of the function SetFont() is true and if the font file is available.

If a standard font should be embedded it is recommended to disable the internal fonts supported by DynaPDF. See also SetFont() or SetUseStdFonts().

```
Default value = 1 (true)
```

GetUseSystemFonts

Syntax:

```
LBOOL pdfGetUseSystemFonts(
    const PPDF* IPDF) // Instance pointer
```

The property specifies whether the default font directory of the operating system should be added to the list of font search paths. On Windows this is the %Windows%/Fonts directory plus additional fonts which can be named in the Registry.

On Mac OS X the following font directories will be added to the list of font search paths (in this order):

```
~/Library/Fonts/
/Library/Fonts/
/System/Library/Fonts/
```

System fonts are loaded on demand, e.g. when a function tries to load a font. To improve processing speed the internal font cache is hold in memory until the PDF instance will be deleted. Additional font search paths can be added with AddFontSearchPath().

Because Linux and UNIX operating system support no default font directory the property is ignored.

Return values:

The return value is 1 if systems fonts are enabled, or zero otherwise.

GetUsesTransparency

Syntax:

```
SI32 pdfGetUsesTransparency(
    const PPDF* IPDF, // Instance pointer
    SI32 PageNum)     // Page number or -1 to check all pages
```

The function checks whether a page or the entire document uses transparency. This is no quick check as GetDocUsesTransparency() applies. The function parses the page or pages to determine whether transparent objects are really used.

To check whether a specific page uses transparency set the parameter *PageNum* to the wished page number. The first page is denoted by 1. To check the entire PDF file set *PageNum* to -1.

Return values:

The return value is a bit mask on success (a positive integer value), or a negative error code on failure.

The following flags are defined:

- 0: The page or document uses no transparency.
- 1: The content stream of a page contains transparent objects.
- 2: A page defines the blending color space (Group dictionary).
- 4: A page contains transparent annotations or form fields.

The above values can occur in any combination. To check whether a specific flag was set use a binary and operator:

Example C++:

```
...
SI32 retval = pdfGetUsesTransparency(pdf, 1);
if (retval < 0)
   // an error occurred
else if (retval == 0)
   // no transparency used
else
{
   if (retval & 1)
   {
      // The page or document uses transparency
   }
   if (retval & 2)
   {
      // At least one page defines the blending color space. If bit 1 is absent then no transparency
      // is used in the content stream of a page.
   }
   if (retval & 4)
   {
      // A page contains transparent annotations or form fields.
   }
}
...
```

GetUseTransparency

Syntax:

```
LBOOL pdfGetUseTransparency(
    const PPDF* IPDF) // Instance pointer
```

The property specifies whether images should get a transparent background. If true, the transparent color (see SetTransparentColor() is used to mask an image. The property has no effect for JPEG compressed images.

```
Default value = 1 (true)
```

GetUseVisibleCoords

Syntax:

```
LBOOL pdfGetUseVisibleCoords(
    const PPDF* IPDF) // Instance pointer
```

The property specifies whether DynaPDF should consider the crop box to calculate the position of an object. The crop box crops a page, but the paper format (media box) is left unchanged. To move the coordinates into the visible area, set the property to true. Coordinates can then be used as if no crop box was defined. The bounding boxes are described in detail at SetBBox().

GetViewerPreferences

Syntax:

```
LBOOL pdfGetViewerPreferences(
    const PPDF* IPDF,      // Instance pointer
    SI32 ADDR Preference, // Preference
    SI32 ADDR AddVal)      // Parameter of the preference if any

typedef SI32 TViewerPreference;
#define vpUseNone             0x00000000 // No preference is set
#define vpHideToolBar         0x00000001 // No parameter
#define vpHideMenuBar         0x00000002 // No parameter
#define vpHideWindowUI        0x00000004 // No parameter
#define vpFitWindow           0x00000008 // No parameter
#define vpCenterWindow        0x00000010 // No parameter
#define vpDisplayDocTitle     0x00000020 // (PDF 1.4) No parameter
#define vpNonFullScrPageMode  0x00000040 // Key, values see below
#define vpDirection           0x00000080 // (PDF 1.3)
#define vpViewArea            0x00000100 // (PDF 1.4)
#define vpViewClip            0x00000200 // (PDF 1.4)
#define vpPrintArea           0x00000400 // (PDF 1.4)
#define vpPrintClip           0x00000800 // (PDF 1.4)

typedef SI32 TViewPrefAddVal;
#define avNone                0x00000000
#define avNonFullScrUseNone   0x00000001
#define avNonFullScrUseOutlines 0x00000002
#define avNonFullScrUseThumbs 0x00000004
#define avNonFullScrUseOC     0x00000400 // PDF 1.6
#define avDirectionL2R        0x00000008
#define avDirectionR2L        0x00000010
#define avViewPrintArtBox     0x00000020
#define avViewPrintBleedBox   0x00000040
#define avViewPrintCropBox    0x00000080
```

```
#define avViewPrintMediaBox      0x00000100
#define avViewPrintTrimBox       0x00000200

#define AV_NON_FULL_SRC_MASK     0x00000005
#define AV_DIRECTION_MASK        0x00000018
#define AV_VIEW_PRINT_MASK       0x000003E0
```

The function retrieves the viewer preferences specified in the document. The parameters *Preference* and *AddVal* are bit masks; they must be masked out with a bitwise and operator. Preferences in red color are keys which have a corresponding value which must be taken from the parameter *AddVal*.

Note that more than one preference can be set at time.

The mask values can be used to easily mask out a specific parameter, see the example below.

Example (C++):

```
...
SI32 pref = 0, value = 0;
pdfGetViewerPreferences(pdf, pref, value);
if ((pref & vpDirection) && (value & avDirectionR2L))
   printf("Right to left reading order was defined!\n");
if (pref & AV_NON_FULL_SRC_MASK)
{
   printf("Non full screen mode is: \n");
   switch(value & AV_NON_FULL_SRC_MASK)
   {
      case avNonFullScrUseNone:     printf("Use none\n");
      case avNonFullScrUseOutlines: printf("Use outlines\n");
      case avNonFullScrUseThumbs:   printf("Use thumbs\n");
   }
}
...
```

GetViewport

```
LBOOL pdfGetViewport(
    const PPDF* IPDF,        // Instance pointer
    UI32 PageNum,            // Page number
    UI32 Index,              // Viewport index
    struct TPDFViewport* VP) // See below

struct TPDFViewport
{
    UI32        StructSize; // Must be set to sizeof(TPDFViewport)
    TFltRect    BBox;       // Bounding box
    IMSR*       Measure;    // Optional -> GetMeasureObj()
    const char* NameA;      // Optional name
    const UI16* NameW;      // Optional name
    IPTD*       PtData;     // Pointer of a Point Data dictionary -> GetPtDataObj().
};
```

The functions returns a viewport that is associated with a page. A viewport is a rectangular region of a page, that specifies usually a measure coordinate system or points in a geospatical coordinate system.

Since viewports might overlap, to determine the viewport to use for any point on a page, the viewports in the array shall be examined, starting with the last one and iterating in reverse, and the first one whose BBox entry contains the point shall be chosen.

The member *StructSize* must be set to sizeof(TPDFViewport) before the function can be called. To determine the number of viewports associated with a page call GetViewportCount().

To determine how viewports and measure dictionaries can be used, please have a look into the PDF Reference 2.0, ISO/DIS 32000-2, Section Measurement properties.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetViewportCount

Syntax:

```
SI32 pdfGetViewportCount(
    const PPDF* IPDF, // Instance pointer
    UI32 PageNum)     // Page number
```

The function returns the number of viewports which are associated with a page. The first page has the page number 1.

Return values:

If the function succeeds the return value is the number of viewports, a number greater or equal zero. If the function fails the return value is a negative error code. The function can only fail if the page number is invalid.

GetWMFDefExtent

Syntax:

```
LBOOL pdfGetWMFDefExtent(
   const PPDF* IPDF, // Instance pointer
   UI32 ADDR Width,  // Width in 0.01 millimetres units
   UI32 ADDR Height) // Height in 0.01 millimetres units
```

The function retrieves the default size which is used to convert non-portable WMF files to EMF. See InsertMetafile() for further information.

```
Default value:
Width  = 0;
Height = 0;
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetWMFPixelPerInch

Syntax:

```
SI32 pdfGetWMFPixelPerInch(
   const PPDF* IPDF) // Instance pointer
```

The function returns the default pixels per inch of the y-axis which are used to convert portable WMF files to EMF. DynaPDF uses a corrected value to get exact proportions. See InsertMetafile() for further information.

```
Default value = 2400
```

GetWordSpacing

Syntax:

```
double pdfGetWordSpacing(
   const PPDF* IPDF) // Instance pointer
```

The function returns the current word spacing.

```
Default value = 0
```

Value = 0	Word Space
Value = 80	Word Space

GetXFAStream

Syntax:

```
LBOOL pdfGetXFAStream(
    const PPDF* IPDF,          // Instance pointer
    UI32 Index,                // Array index
    struct TPDFXFAStream* Out) // Can be NULL. See description.

struct TPDFXFAStream
{
    UI32        StructSize;  // Must be set to sizeof(TPDFXFAStream)
    const BYTE* Buffer;      // XML Stream
    UI32        BufSize;     // Buffer size in bytes
    const char* NameA;       // Stream name
    const UI16* NameW;       // Either the Ansi or Unicode name is set but
                             // never both
};
```

The function returns an XFA stream. The parameter *Index* is the array index. The function GetXFAStreamCount() returns the number of available streams.

The function must decompress the stream before it can be returned. To compress the stream again call the function a second time and set the parameter *Out* to NULL. The function compresses the stream in this case. This step is recommended to reduce the memory usage. Note that the returned pointer become invalid when the stream will be compressed.

An XFA form consist of several XML streams. The stream name must be used to determine which stream was returned.

Return value:

If the function succeeds the return value is 1. If the function fails the return value is 0.

GetXFAStreamCount

Syntax:

```
SI32 pdfGetXFAStreamCount(
    const PPDF* IPDF) // Instance pointer
```

The function returns the number of available XFA streams. XFA streams are available after a PDF file was imported.

Return value:

If the function succeeds the return value is the number of available XFA streams. If the function fails the return value is a negative error code.

HaveDPartRoot

Syntax:

```
LBOOL pdfHaveDPartRoot(
    const PPDF* IPDF) // Instance pointer
```

The function checks whether the PDF file in memory contains a *DPartRoot* dictionary.

DPart stands for Document Parts. *DParts* are a feature of PDF/VT, a PDF format for the printing industry.

The function should be used to determine whether a *DPart* structure is already in memory, before calling CreateDPartRoot() since the function would fail if this is the case.

An existing *DPart* structure can be extended with additional *DPart* nodes, but it is not yet possible to edit existing nodes.

Return values:

If the document contains a *DPartRoot* dictionary, the return value is 1 or 0 otherwise.

HaveOpenDoc

Syntax:

```
LBOOL pdfHaveOpenDoc(
    const PPDF* IPDF) // Instance pointer
```

If an error occurred it is not always clear whether the PDF file was already deleted or if it is still in memory. Therefore, this function can be used to determine whether a PDF file is still in memory.

Return values:

If an open PDF file is in memory the return value is 1. If no open PDF file is in memory the return value is 0.

HaveOpenPage

Syntax:

```
LBOOL pdfHaveOpenPage(
    const PPDF* IPDF) // Instance pointer
```

The function returns true (1) if an open page is in memory. Otherwise the return value is false (0).

HighlightAnnot

Syntax:

```
SI32 pdfHighlightAnnot(
  const PPDF* IPDF,      // Instance pointer
  TAnnotType SubType,    // Allowed values are atHighlight, atSquiggly,
                         // atStrikeOut, or atUnderline
  double PosX,           // X-Coordinate
  double PosY,           // Y-Coordinate
  double Width,          // Annotation height
  double Height,         // Annotation width
  UI32 Color,            // Highlight color
  const char* Author,    // Optional author
  const char* Subject,   // Optional subject (PDF 1.5)
  const char* Comment)   // Optional comment
```

The function creates a Highlight annotation. Highlight annotations are used to mark text on a PDF page.

If the coordinate system is bottom-up the point PosX, PosY defines the lower left corner of the bounding rectangle. If the coordinate system is top-down it defines the upper left corner.

The parameter *Color* specifies the highlight color; it must be defined in RGB color space. The line width of a Strikout or Underline annotation is not adjustable. The line width depends on the size of the annotation and is automatically calculated.

The active color space must be set to DeviceRGB before creating the annotation. See SetColorSpace() for further information.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

ImportBookmarks

Syntax:

```
SI32 pdfImportBookmarks(
   const PPDF* IPDF) // Instance pointer
```

This function imports the outline tree of the currently opened import file (see OpenImportFile() or OpenImportBuffer()).

Return values:

If the function succeeds the return value is the number of imported bookmarks. If the function fails the return value is a negative error code.

ImportCatalogObjects

Syntax:

```
LBOOL pdfImportCatalogObjects(
   const PPDF* IPDF) // Instance pointer
```

The function imports global objects of the currently open import file such as bookmarks, JavaScripts, embedded files, open actions, invisible page templates, rendering intents, the document info entries, and certain other global properties such as the page mode or page layout.

These objects are normally imported by ImportPDFFile(). However, in cases where this function cannot be used, it is possible to import the global objects manually if necessary.

Remarks:

The function considers the current import flags. If certain objects are not imported, check whether the right import flags are set. See also SetImportFlags().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ImportDocInfo

Syntax:

```
LBOOL pdfImportDocInfo(
   const PPDF* IPDF) // Instance pointer
```

The function imports the document info entries from the currently opened import file. Already existing entries are not overridden.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ImportEncryptionSettings

Syntax:

```
LBOOL pdfImportEncryptionSettings(
    const PPDF* IPDF) // Instance pointer
```

The function imports the encryption settings from the currently open import file (see OpenImportFile() or OpenImportBuffer() for further information).

If the PDF file is not encrypted, the function does nothing. To determine whether the open PDF file is encrypted call GetInIsEncrypted().

The encryption settings should be imported to make sure that the PDF file can be saved with the same settings after it was edited. The user rights should be considered if the file was opened with the open password, also if the password was just an empty string. See GetUserRights() for further information.

If the file was opened with the owner password, then all editing rights should be grated.

The PDF file must be closed with CloseFile() or CloseAndSignFile() for example. Don't use CloseFileEx() for example since this function would override the encryption settings.

If the file should be saved unencrypted call ResetEncryptionSettings() before closing the file.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ImportOCProperties

Syntax:

```
LBOOL pdfImportOCProperties(
    const PPDF* IPDF) // Instance pointer
```

The function imports the global Optional Content Properties of the curent open import file (see OpenImportFile() fur further information).

OC properties are normally automatically imported by functions like ImportPDFFile() or when importing a page that contains layers. However, a viewer application must be able to access the OC properties right after the PDF file was opened so that the layer configuration and the layer tree can be loaded (see LoadLayerConfig() and GetOCUINode() for further information).

The function can be called multiple times without causing unwanted side effects.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ImportPage

Syntax:

```
SI32 pdfImportPage(
    const PPDF* IPDF, // Instance pointer
    UI32 PageNum)     // Page number
```

The function imports a page from an external PDF file and converts it to a template. The external PDF file must be opened with OpenImportFile() or OpenImportBuffer() beforehand. The parameter *PageNum* specifies the page number that should be imported; the first page has the number 1.

The conversion of pages to templates is required if an imported page should be scaled or placed on a specific position or if an imported page should be used on more than one destination page, e.g. as a custom page background.

However, in cases where the conversion of pages to templates is not necessary the usage of ImportPageEx() within an open page is preferred. In this case it is also possible to import interactive objects with the page such as annotations or form fields. Such objects cannot be imported with ImportPage() because templates do not support interactive objects.

The new template is **not** placed on the current open page if any. To make the template visible the template must be placed on a page or other template with PlaceTemplate() or PlaceTemplateEx(). The template can be used on arbitrary pages and arbitrary positions. It is also possible to scale a template like an image. See PlaceTemplateEx() for further information.

Imported pages are often called watermark. A watermark is usually a simple text or some kind of graphics that was imported from a PDF file. Such a graphics can be placed in background or in foreground of a page. To place a template in background, add the template as first object to the desired page. If the template should appear in foreground, add it as last object. The background of an imported page is transparent as long as it doesn't contain any opaque objects which blank out the entire background. Scanned images which are converted to PDF are often not transparent. Those pages must be placed in background to avoid overlaying objects.

It is possible to import an external page outside of an open page, and it is also possible to import a page outside of an open output PDF file. However, all templates will be deleted when CloseFile() is called the next time. It is not possible to import an external page as template and hold it in memory for use with more than one PDF file.

Bounding boxes

As mentioned earlier, imported pages are converted to templates by this function. A template has only one bounding box, but a PDF page has 5 bounding boxes (see SetBBox() for a full explanation). The most important bounding boxes are the media box and crop box of the original page.

A bounding box is defined as a rectangle giving the coordinates pair of diagonally opposite corners. The media box of a page is usually expressed in a normalized form where the coordinates of the lower-left point are set to 0.

However, the normalized form is used in 99% of all available documents but it is also possible that a document uses a non-normalized form. For instance, the media box can look like this:

```
mediabox.Left   = -100.0; // lower-left  x
mediabox.Bottom = -100.0; // lower-left  y
mediabox.Right  =  200.0; // upper-right x
mediabox.Top    =  200.0; // upper-right y
```

The size of the page that uses such a bounding box is 300 x 300 units. The problem is that the coordinate origin lies at -100, -100. The page can also contain a crop box and the original page can be rotated. This makes the handling more complicated.

When placing such a template with PlaceTemplate() on a page without further considerations the result will be wrong because PlaceTemplate() does not consider the coordinate origin or the original page orientation.

To simplify the work with imported templates PlaceTemplateEx() should be used instead. This function considers the coordinate origin, crop box, and it preserves the original orientation when the template is placed on a page. See PlaceTemplateEx() for further information.

Return values:

If the function succeeds the return value is a template handle, a value greater or equal zero. If the function fails the return value is a negative error code.

ImportPageEx

Syntax:

```
SI32 pdfImportPageEx(
   const PPDF* IPDF, // Instance pointer
   UI32 PageNum,     // Page number
   double ScaleX,    // Horizontal scaling factor (1.0 = no scaling)
   double ScaleY)    // Vertical scaling factor (1.0 = no scaling)
```

The function imports a PDF page of an external PDF file incl. interactive objects such as annotations, form fields and so on, if any. The external PDF must be opened with the function OpenImportFile() or OpenImportBuffer() beforehand.

Link annotations and article beads are not imported by default because the destinations of such annotations become invalid if the remaining pages are not imported in the same order as they appeared in the original PDF file. Web links are affected too because it is not possible to distinguish between a normal page link and web link annotations during import.

However, if you can make sure that all pages of a PDF file will be imported in the same order as in the original file you can set the import flag ifAllPageObjects (see SetImportFlags()) to disable the normal import behaviour. If this flag is set all annotation types will be imported. However, article beads are always discarded by this function because these objects depend on other global objects which cannot be resolved when importing pages manually. Article beads can only be preserved if the entire PDF file is imported with ImportPDFFile().

The function can be used in two ways, inside an open page or outside.

Calling the function inside of an open page

If the function is called inside an open page, the external page will be imported and scaled by using the scaling factors *ScaleX* and *ScaleY*, if the import flag ifImportAsPage is **not** set (see SetImportFlags() for further information). All bounding boxes of the page will be scaled too and the orientation will be set to the value of the source page.

An external page can only be scaled to an arbitrary format if it is converted to a template. This conversion is done if the import flag ifImportAsPage is not set (default).

However, if a page must not be scaled or used as a template on multiple pages the flag ifImportAsPage should be set to avoid the conversion of pages to templates. In this case, the scaling factors will be ignored and the page is imported as is.

If an imported page contains a crop box (see GetBBox()) the property UseVisibleCoords can be used to move all coordinates into the visible area of the crop box. If the property is true, all functions consider the coordinate origin of the crop box to calculate the position of an object.

Calling the function outside of an open page

If the function is called outside of an open page, no interactive objects are imported and the scaling factors *ScaleX* and *ScaleY* are ignored. The page becomes converted to a template. The function works then exactly like ImportPage(). The resulting template can be placed on one or more pages with PlaceTemplate() or better PlaceTemplateEx(). The latter version is preferred because it considers the coordinate origin of the template as well as a maybe existing crop box and it preserves the original page orientation.

Remarks:

This function is available in an Ansi and Unicode compatible version.

Return values:

If the function succeeds and if the imported page was converted to a template, the return value is a template handle, a value greater or equal zero.

If the imported page was not converted to a template and if the function succeeds, the return value is zero. If the function fails the return value is a negative error code.

ImportPDFFile

Syntax:

```
SI32 pdfImportPDFFile(
    const PPDF* IPDF, // Instance pointer
    UI32 DestPage,    // Add the new pages at this position
    double ScaleX,    // Horizontal scaling factor (1.0 = no scaling)
    double ScaleY)    // Vertical scaling factor (1.0 = no scaling)
```

The function imports an external PDF file including interactive objects such as annotations, bookmarks, form fields, and so on. DynaPDF supports a large set of flags to enable import of required objects only. Unwanted objects can be removed by the function if necessary. The import flags are described in detail at SetImportFlags() and SetImportFlags2().

The parameter *DestPage* specifies the destination page on which the new pages will be inserted or added if the pages already exist. It is possible to import multiple PDF files on the same or overlapping destination page.

Example 1:

```
...
pdfOpenImportFile(pdf, "c:/test1.pdf");
pdfImportPDFFile(pdf, 1, 1.0, 1.0); // Start import at page one
// The previous PDF file will be closed automatically.
pdfOpenImportFile(pdf, "c:/test2.pdf");
// No we have overlapping pages. For instance, if both PDF files
// contain 3 pages, then we get 3 pages in which the pages of the second
// PDF file are placed on top of the already existing ones.
pdfImportPDFFile(pdf, 1, 1.0, 1.0);
pdfCloseImportFile(pdf);
...
```

Example 2:

PDF files can be merged as follows:

```
...
pdfOpenImportFile(pdf, "c:/test1.pdf");
SI32 nextPage;
// We avoid the conversion of pages to templates in this case.
pdfSetImportFlags(ifImportAll | ifImportAsPage);
nextPage = pdfImportPDFFile(pdf, 1, 1.0, 1.0); // Start import at page one
// The previous PDF file will be automatically closed.
pdfOpenImportFile(pdf, "c:/test2.pdf");
// Add the new file behind the previous one.
pdfImportPDFFile(pdf, nextPage + 1, 1, 1);
pdfCloseImportFile(pdf);
...
```

Imported pages are converted to templates by default. This conversion is done by default mainly to achieve the same runtime behaviour in comparison to older versions of DynaPDF and to enable

scaling of PDF pages (scaling is only possible with templates). So, if you want to scale the pages to another page format then the conversion of pages to templates is required.

However, in most cases scaling is not required. The conversion of pages to templates should be avoided in this case by setting the flag ifImportAsPage:

```
pdfSetImportFlags(pdf, ifImportAll | ifImportAsPage);
```

The advantage is that non-converted pages are a little bit smaller and a PDF file can be imported and written to disk arbitrary often without increasing the file size or the nesting depth of templates.

Especially the nesting depth of templates can be an important aspect if a PDF file must edited multiple times. Note that each time a PDF file is imported, while pages are converted to templates, the nesting depth of templates becomes incremented. So, if you would import and save a PDF file in a loop, while each time pages are converted to templates, the resulting PDF file becomes larger and larger, and the nesting depth of templates becomes higher and higher. A nesting depth of more than 8 can already cause printing problems. Viewer applications support usually much higher nesting depths but the memory usage becomes then very large and the rendering speed is decreased.

Return values:

If the function succeeds the return value is the last page number that was touched. For example, if the document in memory contains already 20 pages, and if we import a PDF file with 23 pages, and if the destination page is 21 the return value should be 43 because this is the last page that was invoked during import.

If the function fails the return value is a negative error code.

InitBarcode2

Syntax:

```
LBOOL pdfInitBarcode2(
   struct TPDFBarcode2* Barcode) // Barcode structure
```

The function initializes the *TPDFBarcode2* structure with default values. The member *StructSize* must be set to sizeof(TPDFBarcode2) before the function can be called. The structure size is used to identify different versions of the structure.

Please note that many members of the structure must be initialized with non-zero values. Initializing the structure with zero is no meaningful initialization.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InitColorManagement

```
LBOOL pdfInitColorManagement(
    const PPDF* IPDF,                    // Instance pointer
    struct TPDFColorProfiles* Profiles, // Color profiles or NULL
    TPDFColorSpace DestSpace,            // Destination color space
    TPDFInitCMFlags Flags)               // See below
```

```
struct TPDFColorProfiles
{
    UI32        StructSize;    // Must be set to sizeof(TPDFColorProfile)
    const char* DefInGrayA;    // Optional
    const UI16* DefInGrayW;    // Optional
    const char* DefInRGBA;     // Optional, sRGB is the default.
    const UI16* DefInRGBW;     // Optional
    const char* DefInCMYKA;    // Optional, but this is the most important
    const UI16* DefInCMYKW;    // profile!
    const char* DeviceProfileA; // Optional, must be compatible with the
    const UI16* DeviceProfileW; // output color space. Default is sRGB.
    const char* SoftProofA;     // Optional, emulates another output device.
    const UI16* SoftProofW;     // Optional.
};
```

```
typedef enum
{
    icmDefault        = 0, // Default rules.
    icmBPCompensation = 1, // Black point compensation preserves the black
                           // point when converting CMYK colors to
                           // different color spaces.
    icmCheckBlackPoint = 2 // If set, soft proofing will be disabled if the
                           // black point of the output intent is probably
                           // invalid. The limit is 35 for L* measured in
                           // Lab color space.
}TPDFInitCMFlags;
```

The function enables color management in the functions RenderPage(), RenderPageEx(), RenderPageToImage(), and RenderPDFFile(). All color profiles are optional. Default profiles for DeviceGray and DeviceRGB color spaces (sRGB is default) are automatically created if not provided.

The most important color profile is the CMYK profile because it is not possible to create such a profile with build-in functions. If no CMYK profile is set DynaPDF renders CMYK only with color management if the file contains corresponding ICCBased color spaces or an embedded CMYK output intent.

The file paths can be defined in Ansi or Unicode format (UTF-16). Ansi strings are interpreted in the code page 1252 on Windows and in UTF-8 Unicode on non-Windows operating systems.

It is also possible to load the ICC profiles from file buffers. See InitColorManagementEx() for further information.

The SoftProof profile specifies an arbitrary output color space that should be simulated.

To disable color management set the parameter *Profile* to NULL. Because Visual Basic 6 doesn't accept vbNull for this parameter, call DisableColorManagement() instead. This function exists in the VB 6 interface only.

Although Black Point Compensation (BPC) is not enabled by default, it is recommended to enable it because it improves the rendering quality of CMYK images a lot.

Remarks:

All profile paths must be absolute paths. Otherwise it is maybe not possible to reload a profile if necessary. DynaPDF must be able to reload the profiles if a PDF file contains ICCBased color spaces.

Initializing the color management requires a considerable amount of processing time. It is strongly recommended to use one PDF instance as long as possible so that it must not be initialized again when another PDF file will be rendered.

The color management can be initialized right after the PDF instance was created.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InitColorManagementEx

Syntax:

```
LBOOL pdfInitColorManagementEx(
    const PPDF* IPDF,                      // Instance pointer
    struct TPDFColorProfilesEx* Profiles, // Color profiles or NULL
    TPDFColorSpace DestSpace,              // Destination color space
    TPDFInitCMFlags Flags)                 // See below
struct TPDFColorProfilesEx
{
    UI32        StructSize;      // Must be set to sizeof(TPDFColorProfilesEx)
    const void* DefInGray;       // Optional
    UI32        DefInGrayLen;    // Optional
    const void* DefInRGB;        // Optional
    UI32        DefInRGBLen;     // Optional
    const void* DefInCMYK;       // Optional, but this is the most important
    UI32        DefInCMYKLen;    // but this is the most important profile.
    const void* DeviceProfile;   // Optional, must be compatible with the output color
    UI32        DeviceProfileLen; // space. Default is sRGB.
    const void* SoftProof;       // Optional, emulates another output device.
    UI32        SoftProofLen;    // Optional.
};
typedef enum
{
    icmDefault        = 0, // Default rules.
    icmBPCompensation = 1, // Black point compensation preserves the black
                           // point when converting CMYK colors to
                           // different color spaces.
    icmCheckBlackPoint = 2 // If set, soft proofing will be disabled if the
                           // black point of the output intent is probably
                           // invalid. The limit is 35 for L* measured in
                           // Lab color space.
}TPDFInitCMFlags;
```

The function enables color management in the functions RenderPage(), RenderPageEx(), RenderPageToImage(), and RenderPDFFile() exactly like InitColorManagement() but accepts ICC profile buffers instead of file paths. See InitColorManagement() for further information.

To disable color management set the parameter *Profiles* to NULL.

Remarks:

Initializing the color management requires a considerable amount of processing time. It is strongly recommended to use one PDF instance as long as possible so that it must not be initialized again when another PDF file will be rendered.

The color management can be initialized right after the PDF instance was created.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InitExtGState

Syntax:

```
LBOOL pdfInitExtGState(
   struct TPDFExtGState* GS) // Structure to be initialized
```

The function initializes the structure *GS* with default values. The initialization is strongly recommended before creating an extended graphics state object. See also CreateExtGState().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InitHeaderFooter

Syntax:

```
LBOOL pdfInitHeaderFooter(
   struct TPDFHeaderFooter* Init, // Adress of TPDFHeaderFooter structure
   struct TPDFHdrFtr* HFArray,    // Array of TPDFHdrFtr structures
   UI32 Count)                    // Number of values in the array
```

The function initializes the variables *Init* and *HFArray* with default values.

The member *StructSize* of the structure TPDFHeaderFooter must be initialized to sizeof(TPDFHeaderFooter) before the function can be called. The structure size is used to identify different versions of the structure.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InitOCGContUsage

Syntax:

```
LBOOL pdfInitOCGContUsage(
   struct TPDFOCGContUsage* Value) // Address of TPDFOCGContUsage
                                   // structure
```

The function initializes the TPDFOCGContUsage structure with default values. The member *StructSize* must be set to sizeof(TPDFOCGContUsage) before the function can be called. The structure size is used to identify different versions of the structure.

Once the structure was initialized, one or more members can be changed and passed to SetOCGContUsage().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InitStack

Syntax:

```
LBOOL pdfInitStack(
   const PPDF* IPDF,        // Instance pointer
   struct TPDFStack* Stack) // Stack variable to be initialized
```

The function initializes the variable *Stack* with default values and prepares the editing of a content stream. This function must always be called before a content stream can be parsed with the function GetPageText(). See also GetPageText().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InkAnnot

Syntax:

```
SI32 pdfInkAnnot(
   const PPDF* IPDF,          // Instance pointer
   struct TFltPoint* Points,  // Array of points
   UI32 NumPoints,            // Number of points (must be greater 1)
   double LineWidth,          // Line width
   UI32 Color,                // Color
   TPDFColorSpace CS,         // Color space in which Color is defined
   const char* Author,        // Optional author
   const char* Subject,       // Optional subject
   const char* Content)       // Optional content
```

The function creates an Ink annotation. An ink annotation (PDF 1.3) represents a freehand "scribble" composed of one or more disjoint paths. The parameter *Points* represents a stroked path. More paths can be added with AddInkList().

The points are interpreted in currect user space. Any transformation that was applied on the coordinate system will be taken into account.

The points are connected with bezier curves to achieve a smooth transition between points. If the points should be connected with straight lines, then create a PolyLine annotation instead (see PolyLineAnnot()).

This annotation type has an associated PopUp annotation that displays the string *Content* in a floating window. The initial window state of the associated PopUp annotation is closed by default but the state can be changed with SetAnnotOpenState() if necessary.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

InsertBarcode

Syntax:

```
SI32 pdfInsertBarcode(
    const PPDF* IPDF,              // Instance pointer
    double PosX,                   // X-Coordinate of destination rectangle
    double PosY,                   // Y-Coordinate of destination rectangle
    double Width,                  // Width of destination rectangle
    double Height,                 // Height of destination rectangle
    TCellAlign HAlign,             // Horizontal alignment
    TCellAlign VAlign,             // Vertical alignment
    struct TPDFBarcode2* Barcode)  // See below
```

```
typedef enum TPDFBarcodeType
{
    bctNotInitialized   = 0x0000, // No valid value
    bctAustraliaPost    = 0x003F, // Australia Post Standard Customer
    bctAustraliaRedir   = 0x0044, // Australia Post Redirection
    bctAustraliaReply   = 0x0042, // Australia Post Reply Paid
    bctAustraliaRout    = 0x0043, // Australia Post Routing
    bctAztec            = 0x005C, // Aztec Code
    bctAztecRunes       = 0x0080, // Aztec Runes
    bctC2Of5IATA        = 0x0004, // Code 2 of 5 IATA
    bctC2Of5Industrial  = 0x0007, // Code 2 of 5 Industrial
    bctC2Of5Interleaved = 0x0003, // Interleaved 2 of 5
    bctC2Of5Logic       = 0x0006, // Code 2 of 5 Data Logic
    bctC2Of5Matrix      = 0x0002, // Standard Code 2 of 5
    bctChannelCode      = 0x008C, // Channel Code
    bctCodabar          = 0x0012, // Codabar
    bctCodablockF       = 0x004A, // Codablock-F
    bctCode11           = 0x0001, // Code 11
    bctCode128          = 0x0014, // Code 128 (automatic subset switching)
    bctCode128B         = 0x003C, // Code 128 (Subset B)
    bctCode16K          = 0x0017, // Code 16K
    bctCode32           = 0x0081, // Code 32
    bctCode39           = 0x0008, // Code 3 of 9 (Code 39)
    bctCode49           = 0x0018, // Code 49
    bctCode93           = 0x0019, // Code 93
    bctCodeOne          = 0x008D, // Code One
    bctDAFT             = 0x005D, // DAFT Code
    bctDataBarOmniTrunc = 0x001D, // GS1 DataBar-14 Omnidirectional and truncated / RSS14
    bctDataBarExpStacked = 0x0051, // GS1 DataBar Expanded Stacked
    bctDataBarExpanded  = 0x001F, // GS1 DataBar Expanded / RSS_EXP
    bctDataBarLimited   = 0x001E, // GS1 DataBar Limited  / RSS_LTD
    bctDataBarStacked   = 0x004F, // GS1 DataBar-14 Stacked
    bctDataBarStackedO  = 0x0050, // GS1 DataBar-14 Stacked Omnidirectional
    bctDataMatrix       = 0x0047, // Data Matrix ECC200
    bctDotCode          = 0x0073, // DotCode
    bctDPD              = 0x0060, // DPD code
    bctDPIdentCode      = 0x0016, // Deutsche Post Identcode
    bctDPLeitcode       = 0x0015, // Deutsche Post Leitcode
    bctEAN128           = 0x0010, // GS1-128 (UCC.EAN-128)
    bctEAN128_CC        = 0x0083, // Composite Symbol with GS1-128 linear component
    bctEAN14            = 0x0048, // EAN-14
    bctEANX             = 0x000D, // EAN
    bctEANX_CC          = 0x0082, // Composite Symbol with EAN linear component
    bctEANXCheck        = 0x000E, // EAN + Check Digit
    bctExtCode39        = 0x0009, // Extended Code 3 of 9 (Code 39+)
    bctFIM              = 0x0031, // FIM
    bctFlattermarken    = 0x001C, // Flattermarken
    bctHIBC_Aztec       = 0x0070, // HIBC Aztec Code
    bctHIBC_CodablockF  = 0x006E, // HIBC Codablock-F
    bctHIBC_Code128     = 0x0062, // HIBC Code 128
```

```
    bctHIBC_Code39       = 0x0063, // HIBC Code 39
    bctHIBC_DataMatrix   = 0x0066, // HIBC Data Matrix ECC200
    bctHIBC_MicroPDF417  = 0x006C, // HIBC MicroPDF417
    bctHIBC_PDF417       = 0x006A, // HIBC PDF417
    bctHIBC_QR           = 0x0068, // HIBC QR Code
    bctISBNX             = 0x0045, // ISBN (EAN-13 with verification stage)
    bctITF14             = 0x0059, // ITF-14
    bctJapanPost         = 0x004C, // Japanese Postal Code
    bctKIX               = 0x005A, // Dutch Post KIX Code
    bctKoreaPost         = 0x004D, // Korea Post
    bctLOGMARS           = 0x0032, // LOGMARS
    bctMailmark          = 0x0079, // Royal Mail 4-State Mailmark
    bctMaxicode          = 0x0039, // Maxicode
    bctMicroPDF417       = 0x0054, // Micro PDF417
    bctMicroQR           = 0x0061, // Micro QR Code
    bctMSIPlessey        = 0x0047, // MSI Plessey
    bctNVE18             = 0x004B, // NVE-18
    bctPDF417            = 0x0037, // PDF417
    bctPDF417Truncated   = 0x0038, // PDF417 Truncated
    bctPharmaOneTrack    = 0x0033, // Pharmacode One-Track
    bctPharmaTwoTrack    = 0x0035, // Pharmacode Two-Track
    bctPLANET            = 0x0052, // PLANET
    bctPlessey           = 0x0056, // Plessey
    bctPostNet           = 0x0028, // PostNet
    bctPZN               = 0x0034, // PZN
    bctQRCode            = 0x003A, // QR Code
    bctRMQR              = 0x0091, // Rectangular Micro QR Code (rMQR)
    bctRoyalMail4State   = 0x0046, // Royal Mail 4 State (RM4SCC)
    bctRSS_EXP_CC        = 0x0086, // Composite Symbol with GS1 DataBar Extended component
    bctRSS_EXPSTACK_CC   = 0x008B, // Composite Symbol with GS1 DataBar Expanded Stacked component
    bctRSS_LTD_CC        = 0x0085, // Composite Symbol with GS1 DataBar Limited component
    bctRSS14_CC          = 0x0084, // Composite Symbol with GS1 DataBar-14 linear component
    bctRSS14Stacked_CC   = 0x0089, // Composite Symbol with GS1 DataBar-14 Stacked component
    bctRSS14StackOMNI_CC = 0x008A, // Composite Symbol with GS1 DataBar-14 Stacked Omnidirectional
    bctTelepen           = 0x0020, // Telepen Alpha
    bctTelepenNumeric    = 0x0057, // Telepen Numeric
    bctUltracode         = 0x0090, // Ultracode
    bctUPCA              = 0x0022, // UPC A
    bctUPCA_CC           = 0x0087, // Composite Symbol with UPC A linear component
    bctUPCACheckDigit    = 0x0023, // UPC A + Check Digit
    bctUPCE              = 0x0025, // UCP E
    bctUPCE_CC           = 0x0088, // Composite Symbol with UPC E linear component
    bctUPCECheckDigit    = 0x0026, // UPC E + Check Digit
    bctUPNQR             = 0x008F, // UPNQR (Univerzalni Placilni Nalog QR)
    bctUSPSOneCode       = 0x0055, // USPS OneCode
    bctVIN               = 0x0049  // Vehicle Identification Number (America)
}TBarcodeType;

typedef enum TPDFBarcodeDataType
{
    bcdtBinary     = 0, // Binary data.
    bcdtUnicode    = 1, // UTF-8.
    bcdtGS1Mode    = 2, // Encodes GS1 data using FNC1 characters.
    bcdtEscapeMode = 8  // Scan input data for escape sequences. This flag can be combined with the
                        // other constants, e.g. TBarcodeDataType(bcdtUnicode | bcdtEscapeMode).
}TBarcodeDataType;

// The following flags can be combined with a binary or operator, e.g.
TBarcodeOptions(bcoNoASCII | bcoDottyMode).
typedef enum TPDFBarcodeOptions
{
    bcoDefault       = 0x00000000, // Use default settings.
    bcoNoASCII       = 0x00000001, // Consider non-ASCII characters when creating the barcode.
    bcoBind          = 0x00000002, // Boundary bars above and below the symbol and between rows.
    bcoBox           = 0x00000004, // Add a box surrounding the symbol and whitespace.
    bcoReaderInit    = 0x00000010, // Add a reader initialisation symbol before encoding.
    bcoSmallText     = 0x00000020, // Use a smaller font for human readable text.
    bcoBoldText      = 0x00000040, // Embolden human readable text.
```

```
    bcoDottyMode         = 0x00000100, // Plot a matrix symbol using dots rather than squares.
    bcoGS1_GS_Separator = 0x00000200, // Use GS instead FNC1 as GS1 separator.
    bcoImageOutput       = 0x01000000, // Draw the barcode as an image.
    bcoUseActiveFont     = 0x02000000  // Vector output only. If set, text is output with the active
                                       // font. Helvetica is used otherwise.
}TBarcodeOptions;
struct TPDFBarcode2
{
    UI32                 StructSize;   // Must be set to sizeof(TPDFBarcode2).
    TPDFBarcodeType      BarcodeType;  // The type of barcode that should be created.
    UI32                 BgColor;      // Background color -> Default NO_COLOR (transparent).
    UI32                 BorderWidth;  // Border width -> default 0.
    const char*          Data;         // Required: Data string.
    TPDFBarcodeDataType DataType;      // Data type and escape flag.
    float                DotSize;      // Dotty mode only -> Default 4.0/5.0.
    SI32                 Eci;          // Extended Channel Interpretation mode -> Default 0.
    UI32                 FgColor;      // RGB Foreground color -> Default 0.
    float                FSizeFactor;  // Font size correction factor -> Default 0.7.
    SI32                 Option1;      // Various, depends on barcode type. -> Default -1.
    SI32                 Option2;      // Various, depends on barcode type. -> Default 0.
    SI32                 Option3;      // Various, depends on barcode type. -> Default 0.
    TPDFBarcodeOptions   Options;      // Output options.
    SI32                 Orientation;  // Supported values: 0, 90, -90, 180, -180, 270, -270.
    const char*          Primary;      // Composite codes only. Primary data message (max 127 bytes).
    float                Scale;        // Image based barcodes only. Default 1.0.
    LBOOL                ShowText;     // Show human readable text? Default true.
    UI32                 SpaceWidth;   // Whitespace width in pixels -> Default 0 (auto).
    UI32                 SymbHeight;   // Default 50.
    float                TextOffsetY;  // Offset to correct the y-coordinate of text -> Default 0.
    struct TFltRect      OuterBorder;  // Optional border around the scaled barcode measured in user
                                       // space. The border is drawn in BgColor.
};
```

The function inserts a barcode to the current open page. The creation of a barcode starts with the initialisation of the structure *TPDFBarcode2* with InitBarcode2(). However, before we can call InitBarcode2(), *StructSize* must be set to sizeof(TPDFBarcode2) (C/C++ only). This is required because the structure size is used to identify different versions of the structure.

After the structure was initialized, the barcode type and data must be set. Composite barcodes require also a primary message (member *Primary*) but that's it, most barcode types can already be created with this minimal initialisation.

Although it is possible to create barcodes with pure default values, we want now take a deeper look into the barcode options and available flags.

Output position and orientation

Because every barcode has a distinct size that is usually unknown, a barcode is placed into a destination rectangle. If the coordinate system is bottom-up the point *PosX, PosY* defines the lower left corner of the output rectangle. If the coordinate system is top-down it defines the upper left corner.

The barcode is horizontally and vertically aligned into the output rectangle according to the values of *HAlign* and *VAlign*.

The barcode can also be rotated in 90 degrees steps with the member *Orientation*. Note that the barcode will be rotated and not the output rectangle!

Vector vs. Image Barcodes

All barcode types, with exception of Ultracode, can be created as vector graphic (default) or as an image. Vector based barcodes offer the highest output quality but it is not always recommended to use this format. For example, image based barcodes can easily be extracted and analyzed with a barcode reader. This is not possible with vector based barcodes.

An image based barcode will be created if the flag *bcoImageOutput* is set. If *BgColor* is set to white or NO_COLOR (default) and if the barcode type is not Ultracode, a 1 bit image will be created and compressed with flate if the resulting binary size is less than 500 bytes. If the image is larger, JBIG2 compression will be used. The resulting image is often smaller as the vector version of the same barcode.

Output resolution

Image based barcodes are not necessarily resolution dependent. This is due to the fact that most barcodes consist of rectangles only. This special kind of image data can be scaled with almost no loss of quality.

However, the output resolution plays a rule if a barcode contains non-rectangular areas like dots, circles, or text. DotCode and Maxicode are examples, as well as all barcodes with human readable text.

The current implementation of the barcode library is not able to scale text with the surrounding barcode. Therefore, it is recommended to draw the barcode as vector graphic if the text quality is not high enough. At time of publication it is not possible to draw the text portions independent of an image based barcode as it is the case in vector mode.

DotCode and Maxicode are the only barcodes which should be rendered in a higher resolution to increase the image quality.

The resolution can be increased with the member *Scale*. As the name suggests, *Scale* is just a scaling factor. Because a barcode has always a distinct size that is resolution independent, it is not possible to render a barcode in an arbitrary resolution, the scaling capabilities are limited.

Scale must be between 0.1 though 30.0 although values greater than 10 make probably no sense.

Background and foreground colors

If *BgColor* is set to NO_COLOR (default) the barcode is drawn transparent. In image mode the barcode is rendered as an image mask. In vector mode, no background rectangle is drawn in the size of the barcode.

The foregound and background color must be specified in DeviceRGB. RGB values can be created with the macro PDF_RGB() or with common functions like rgb() which are available in most programming languages. Other color spaces are not yet supported for barcodes.

Human readable text

Many barcodes consist of the barcode and a human readable text portion that is mostly drawn below the barcode. In image mode it is possible to adjust the font size and font weight with the flags *bcoSmallText* and *bcoBoldText*. More options are not available in image mode.

In vector mode, we have more control over the text. The default font for text is the standard font Helvetica but it is possible to change it to an arbitrary font by setting the flag *bsoUseActiveFont*.

When this flag is set the active font is used to output the text. This font should be loaded with the code page *cp1252* or *cpUnicode*. If no font is active the function falls back again to Helvetica. This font is always available unless it was disabled with SetUseStdFonts() or SetFontSearchOrder(). Note that the function fails with a fatal error if no font can be loaded when text must be drawn.

The function tries to achieve an almost identical result in comparison to the image version of a barcode. This required a bit fine tuning by adjusting the font size and vertical text position. The font size scaling factor (member *FSizeFactor*) is set to 0.7 by default but it can be set to values between 0.3 through 3.0. However, the text cannot become wider as the barcode. The font size will be decreased if the text does not fit into the barcode width. Setting the scaling factor to an unneccesary large value increases the processing time.

The vertical text position can be adjusted with the member *TextOffsetY*.

Outer border

The member *OuterBorder* can be used to extend the background rectangle. Each side of *OuterBorder* specifies the line width of the corresponding side of the rectangle. *OuterBorder* is measured in user space whereas *BorderWidth* is measured in pixels of the reference grid (see Scaling Details below).

Note that the background rectangle is drawn only if *BgColor* is set to another value than NO_COLOR (default).

Scaling details

Barcodes are rendered internally to a virtual integer raster that we call reference grid. Although the width and height of the reference grid is mostly unknown, the height of lines or bars of 1D codes can be controlled with the member *SymbHeight* and the amount of space between bars can be adjusted with *SpaceWidth*. Both values are measured in pixels of the reference grid.

The default symbol height of 1D barcodes is 50. The default space width is just zero since meaningful values depend on the barcode type.

The full height of a 1D barcode is the height of the bars plus optional text, that is mostly drawn below the code. The default font size is 8 for image based barcodes and 9 for vector based barcodes.

Although not every part of a barcode can be scaled freely, with *SymbHeight* and *SpaceWidth* it is possible to control the proportions of a barcode precise enough to achieve reasonable results.

Special options

A few barcode types support additional flags or properties which can be set with the members *Option1* through *Option3*.

Unfortunately, the Zint library is completely undocumented in regard to barcode options. The following values were taken from the source codes and maybe incomplete:

Barcode type	Member to set	Description	Supported range
bctAztec	Option1	Error correction level	-1..4
bctAztec	Option2	Symbol size	0..36
bctCodablockF	Option1	Number of rows	-1..44
bctCodablockF	Option2	Number of columns	-1, 0, 9..67
bctDataMatrix	Option3	Force square output	100
bctDataMatrix	Option3	Enable Rectangular Extension	101
bctDataMatrix	Options	Use GS1 separator instead of FNC1	bcoGS1_GS_Separator
bctMaxicode	Option1	Maxicode mode	-1, 2..6
bctCodabar	Option2	Add checksum	0, 1
bctCode32	Option2	Add checksum	0, 1
bctMSIPlessey	Option2	Symbol size	0..4
bctPharmaOneTrack	Option2	Add checksum	0, 1
bctPharmaTwoTrack	Option2	Add checksum	0, 1
bctPDF417	Option1	Security value	-1..8
bctPDF417	Option2	Number of columns	0..30
bctQRCode	Option1	Error correction level	-1, 1..4
bctQRCode	Option2	Symbol size	0..40
bctQRCode	Option3	Full multibyte -> Kanji mode	unsupported
bctRSS_EXPSTACK_CC	Option1	Component linkage flag	0, 2
bctRSS_EXPSTACK_CC	Option2	Number of columns	0..11
bctUltracode	Option1	Error correction level	-1..6
bctUltracode	Option3	Ultra compression	128

Return values:

The return value depends on whether an image or vector graphic was created. If the flag *bcoImageOutput* is set or if the barcode type is *bctUltracode*, the return value is the image handle, a value greater or equal zero on success or a negative error code on failure.

In vector mode the return value is zero on success or a negative error code on failure.

InsertBMPFromBuffer (obsolete)

Syntax:

```
SI32 pdfInsertBMPFromBuffer(
   const PPDF* IPDF,    // Instance pointer
   double PosX,         // X-Coordinate of destination rectangle
   double PosY,         // Y-Coordinate of destination rectangle
   double ScaleWidth,   // Scaled width of destination rectangle
   double ScaleHeight,  // Scaled height of destination rectangle
   const void* Buffer)  // Source buffer of the DIB
```

The function inserts a Device Independent Bitmap from a file buffer. The parameter *Buffer* requires a normal file buffer as bitmap image.

This function is marked as obsolete, please use InsertImageFromBuffer() if possible.

Return values:

If the function succeeds the return value is the image handle, a value greater or equal zero. If the function fails the return value is a negative error code.

InsertBMPFromHandle

Syntax:

```
SI32 pdfInsertBMPFromHandle(
   const PPDF* IPDF,    // Instance pointer
   double PosX,         // X-Coordinate of destination rectangle
   double PosY,         // Y-Coordinate of destination rectangle
   double ScaleWidth,   // Scaled width of destination rectangle
   double ScaleHeight,  // Scaled height of destination rectangle
   void* hBitmap);      // Windows HBITMAP handle
```

This function inserts a bitmap in the same way as InsertImage() but accepts a HBITMAP handle as input. The bitmap must not be selected into a device context when calling this function.

See also InsertImage().

Return values:

If the function succeeds the return value is the image handle, a value greater or equal zero. If the function fails the return value is a negative error code.

InsertBookmark

Syntax:

```
SI32 pdfInsertBookmark(
   const PPDF* IPDF,   // Instance pointer
   const char* Title,  // Bookmark title
   SI32 Parent,        // Immediate parent bookmark
   UI32 DestPage,      // Destination page that should be opened
```

```
    LBOOL Open,          // Open or close the node when it contains children
    LBOOL AddChildren)   // Add all bookmarks below the new one as children
```

The function inserts a bookmark in an existing outline tree. It sets also the page mode to pmUseOutline (see SetPageMode()). If the outline tree should not be shown when opening the document then set the page mode back to pmUseNone or another value before closing the document.

Parent can be the handle of the parent bookmark after the new bookmark should be inserted or **-1**. If *Parent* is set to **-1** the bookmark is inserted as the first root node in the outline tree.

If the parameter *AddChildren* is set to true, the bookmarks below the new one are added as children to the new bookmark. This can be useful when merging PDF files and if the bookmarks from each PDF file should be separated.

Notice:

When inserting a bookmark the handles of all bookmarks below the new one are incremented by one.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252 only. To create a bookmark in an arbitrary encoding convert the string to Unicode with the function ConvToIncode() and use the Unicode version to create the bookmark.

Return values:

If the function succeeds the return value is the bookmark handle, a value greater or equal zero. If the function fails the return value is a negative error code.

InsertBookmarkEx

Syntax:

```
SI32 pdfInsertBookmarkEx(
    const PPDF* IPDF,  // Instance pointer
    const char* Title, // Bookmark title
    SI32 Parent,       // Immediate parent bookmark
    UI32 NamedDest,    // Named Destination handle
    LBOOL Open,        // Open or close the node when it contains children
    LBOOL AddChildren) // Add all bookmarks below the new one as children
```

The function inserts a bookmark in an existing outline tree. It sets also the page mode to pmUseOutline (see SetPageMode()). If the outline tree should not be shown when opening the document then set the page mode back to pmUseNone or another value before closing the document.

Parent can be the handle of the parent bookmark after the new bookmark should be inserted or **-1**. If *Parent* is set to **-1** the bookmark is inserted as the first root node in the outline tree.

If the parameter *AddChildren* is set to true, the bookmarks below the new one are added as children to the new bookmark. This can be useful when merging PDF files and if the bookmarks from each PDF file should be separated.

NamedDest must be a valid handle from a named destination, see CreateNamedDest() for further information. If the destination lies in another document then insert the bookmark with InsertBookmark(), create an extended Go To Remote Action with CreateGoToRActionEx(), and add the action finally to the bookmark with AddActionToObj().

Notice:

When inserting a bookmark the handles of all bookmarks below the new one are incremented by one.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252 only. To create a bookmark in an arbitrary encoding convert the string to Unicode with the function ConvToIncode() and use the Unicode version to create the bookmark.

Return values:

If the function succeeds the return value is the bookmark handle, a value greater or equal zero. If the function fails the return value is a negative error code.

InsertImage (obsolete)

Syntax:

```
SI32 pdfInsertImage(
   const PPDF* IPDF,    // Instance pointer
   double PosX,         // X-Coordinate of destination rectangle
   double PosY,         // Y-Coordinate of destination rectangle
   double ScaleWidth,   // Scaled width of destination rectangle
   double ScaleHeight,  // Scaled height of destination rectangle
   const char* AFile)   // File path
```

This function is marked as obsolete, please use the function InsertImageEx() if possible.

InsertImageEx

Syntax:

```
SI32 pdfInsertImageEx(
   const PPDF* IPDF,    // Instance pointer
   double PosX,         // X-Coordinate of destination rectangle
   double PosY,         // Y-Coordinate of destination rectangle
   double ScaleWidth,   // Scaled width of destination rectangle
   double ScaleHeight,  // Scaled height of destination rectangle
   const char* AFile,   // File path
   UI32 Index)          // Image index
```

The function inserts an image from a file. The parameter *Index* specifies the image index of a multi-page image. The first image index is denoted by 1. If the image file is not a multi-page image, the parameter *Index* will be ignored. To determine the number of images stored in an image file call the function GetImageCount().

The image will be compressed with the compression filter that was set with SetCompressionFilter(). The default filter is Flate, which is a lossless compression filter. JPEG images should be inserted in pass-though mode if possible, see the description of the JPEG filter below.

Supported image formats

The function does not depend on a correct file extension. It parses the image header to determine the image format. The following image formats are supported:

- BMP -> Device independent bitmaps with and without a valid file header. Embedded PNG and JPEG images are supported. Which filter should be used for bitmaps depends on type pf the image. Bitmaps are usually uncompressed or run-length encoded. Since Flate is a lossless compression filter, this filter can be used as a default if the filter cannot be changed in the application.

- GIF -> The best alternate compression filter for GIF images is Flate encoding. GIF images are treated as single page image. It is not possible to access a specific frame of a GIF animation.

- JPG -> JPEG images should be inserted in pass-though mode if possible. See the description of the compression filter below.

- JP2, JPC, or JPX -> These formats are JPEG 2000 compressed images.

- PBM, PGM, PNM, PPM -> ASCII and Binary encoded images.

- PNG -> Flate compressed images. Since Flate is the native compression filter of PNG images, this filter should be set when inserting the image.

- PSD -> Only the preview image will be processed.

- TIFF

How to get the image format?

The most important image parameters such as width, height, bits per pixel and the default compression filter can be retrieved with the function ReadImageFormat() or ReadImageFormatEx().

Color spaces

The handling of image color space is described in detail at Color Spaces and Images.

How to change the color depth?

DynaPDF can reduce the color depth of true color images to 256 color images. The color depth will be changed if the default color depth was set to 8 bit per pixel and if the used compression filter is Flate. The default color depth can be changed with the function SetDefBitsPerPixel().

Duplicate check

The function performs by default a duplicate check so that the same image can be inserted multiple times without increasing the resulting PDF file size. For file based images the function compares the file name and the output parameters like the scaled width and height, transparency settings and so on. For memory based images the entire image buffer is taken into account. The required processing time for the duplicate check is therefore considerable higher for memory based images.

However, the duplicate check for file based images can lead to wrong results, e.g. if the application creates temporary images with the same file name. In such cases it is possible to disable the duplicate check with the flag gfNoImageDuplCheck. The flag can be set with SetGStateFlags().

Compression Filters

At time of publication DynaPDF supports Flate, CCITT Fax G3, CCITT Fax G4, JBIG2, JPEG, and JPEG 2000 compression. The compression filter can be individually set with SetCompressionFilter().

Flate Encode

Flate encode (also called zip compression) is a loss-less compression filter. It can be used with all supported color spaces. It produces good compression rates for images with large uniform surfaces.

It is also the best compression filter for images with color depths less than 8 bits (except 1 bit). The compression level can be adjusted with the function SetCompressionLevel().

CCITT Fax G3/4

CCITT Fax G3 and G4 compression are lossless compression filters for 1 bit b/w images. These filter are relatively fast and achieve good compression rates. However, JBIG2 compression achieves much higher compression rates! Therefore, these filters should no longer be used.

JBIG2

JBIG2 is a lossless compression filter for 1 bit b/w images. It achieves on average 1.5 to 8 times higher compression rates than CCITT Fax G4 and it is relatively fast. This filter should be used by default for 1 bit black & white images. JBIG2 supports also a lossy compression method but this mode is not implemented in DynaPDF because errors in the resulting image are usually not acceptable.

JPEG Encode

JPEG is a lossy compression method which produces very good compression rates for color and gray scale images. It can be used in combination with all supported color spaces. The compression ratio and resulting quality can be adjusted with SetJPEGQuality(); the default quality is 70%.

JPEG images are inserted in pass-through mode if the image resolution is lower or equal to the specified resolution (see SetResolution()) and if the image color space matches the destination color space. DynaPDF removes embedded ICC profiles also in pass-through mode since PDF readers do not use such profiles when displaying an image. See Embedded ICC profiles below for further information.

The image scaler can be disabled with SetSaveNewImgageFormat(). It is also possible to explicitly disable the pass-through mode by setting the JPEG quality to a negative value. The image will be re-compressed in this case. This can be useful if a higher compression ratio should be achieved.

Real pass-through mode

JPEG images are rebuild by default, also in pass-through mode, to avoid issues with certain malformed images which cannot be displayed in Adobe's Acrobat or Reader and to remove embedded ICC profiles.

The image is not re-compressed when rebuilding the image structure but this action requires some processing time which can be avoided when it is known that the image is already Adobe Acrobat or Reader compatible and if the image contains no embedded ICC profile. To enable real pass-through mode set the flag *gfRealPassThrough* with SetGStateFlags(). The processing speed will be significantly improved in this case.

Embedded ICC profiles

JPEG images can contain embedded ICC profiles but PDF readers ignore such profiles. However, DynaPDF can extract the embedded ICC profile to create an ICC based color space for the image. To enable the usage of embedded ICC profiles set the flag *gfUseImageColorSpace* with SetGStateFlags(). See also Color Spaces and Images. The embedded ICC profile will be removed from the image stream so that no unnecessary data must be stored in the PDF file.

> Note that embedded ICC profiles will not be removed if real pass-through mode is enabled. Especially embedded CMYK profiles are very large and should be removed if possible. Real pass-through mode should be disabled in this case.

Optimized Huffmann encoding

Images are compressed with optimized Huffmann tables if the current compression level is set to *clMax* (see SetCompressionLevel()). This level produces better compression ratios without loosing quality. However, the optimized compression method is minimal slower than the normal one.

JPEG images are inserted in path-through mode if either the property SetSaveNewImageFormat() is set to false (default = true) or if no further conversion or downscaling is required.

JPEG 2000

JPEG 2000 is a compression method that supports loss-less and lossy compression. JPEG 2000 images support alpha transparency but most image viewers discard the alpha channel. This can lead to diffent results since DynaPDF considers the alpha channel by default. The alpha channel can be ignored if necessary by setting the flag *gfNoBitmapAlpha* with SetGStateFlags().

JPEG 2000 compression achieves very good compression rates for color and gray scale images. It can be used with all color depths and supported color spaces. However, note that most current image viewers do not support CMYK or CMYKA images. Already RGBA or GrayA images are incorrectly displayed by most popular image viewers.

If you create GrayA, RGBA, CMYK, or CMYKA images, e.g. with the rendering engine of DynaPDF, be aware that many image viewers are not able to display such images error free.

Setting the image quality

The image quality can be set with SetJPEGQuality() in the very same way as for JPEG images.

However, the JPEG 2000 encoder supports two different ranges: 0..100 and 0..-1000. A positive value represents a percentage. A negative number is used as absolute value that is divided by 10. It disables also the pass-through mode of the JPEG and JPEG 2000 decoders.

The loss-less variant is used if the JPEG quality is set to 0, 100, or -1000.

Quality comparison against JPEG

JPEG 2000 achieves in most cases a sligthly better result in comparison to JPEG for a given compression ratio. Especially images with larger uniform colored areas achieve better results with JPEG 2000 compression because this filter does not produce typical compression artifacts due to color quantization which are typical for JPEG compression.

However, one thing that must be considered is the fact that Adobe's Acrobat uses another rendering technique to display JPEG 2000 images. This leads to worse results because JPEG 2000 images are rendered a bit blurry while JPEG images appear sharp. For a correct comparison render the file with DynaPDF or compare images directly with an image viewer.

Embedded ICC profiles

JPEG 2000 images can contain an embedded ICC profile that can be used to create an ICC based color space for the image (ICC based color spaces are device independent). To enable the usage of embedded ICC profiles set the flag *gfUseImageColorSpace* with SetGStateFlags(). See also Color Spaces and Images.

Pass-through mode

JPEG 2000 images can be inserted in pass-through mode. Pass-through mode is not enabled by default since not all features are known which are unsupported in Adobe's Acrobat. To enable pass-through mode set the flag *gfRealPassThrough* with SetGStateFlags().

Note that this flag just enables pass-through mode. That means the image is inserted in pass-through mode if no scaling is required and if the image is probably compatible with Adobe's Acrobat. Images in known incompatible formats are re-compressed as usual.

The output should be tested with Adobe's Acrobat or Reader when pass-through mode is enabled.

Why does DynaPDF not support LZW compression?

LZW is a very fast lossless compression filter that can be used for sampled images and arbitrary binary data. Flate encoded output is usually more compact than LZW encoded output for the same input, but Flate encoding is considerably slower than LZW. However, LZW encoding is marked as obsolete by Adobe. Because it is not clear whether this filter will be supported in future versions of Adobe's Acrobat, it will not be added to DynaPDF.

TIFF images

The TIFF image format is the most complex format of all available image formats. DynaPDF supports a full featured implementation of the TIFF format incl. most currently available compression methods and color spaces. TIFF images can be organized in scanlines, tiles or stripes, all three formats are fully supported by DynaPDF.

If the image is a multi-page image, a specific image can be selected with the parameter *Index*. The first image has the index 1. The number of available images inside a multi-page image can be determined with the function GetImageCount().

TIFF images can be defined in device dependent and device independent color spaces, such as Lab, CalGray, CalRGB, or ICC based color spaces. If the original color space should be preserved if possible, set the flag gfUseImageColorSpace with SetGStateFlags(). See also Color Spaces and Images.

1 Bit TIFF images

1 bit TIFF images use sometimes different resolutions for the x- and y-axis. DynaPDF considers the resolution information and scales the image according to the resolution information that is stored in the image file. Note that the resolution information is only considered for 1 bit images. No adjustment will be applied for higher bit depths since most image viewers do the same.

It is important to know that ReadImageFormat() or ReadImageFormat2() return the logical size of 1 bit TIFF images. That means, the resolution information was already taken into account. The physical size of a 1 bit TIFF image can be calculated as follows:

```
SI32 bits, useZip;
UI32 resX = 0, resY = 0, physHeight, physWidth, w, h;

pdfReadImageFormat2(pdf, "test.tif", 1, w, h, bits, useZip);
pdfReadImageResolution(pdf, "test.tif", 1, resX, resY);

if (resX != resY && resX > 0 && resY > 0)
{
   if (resX > resY)
   {
      physWidth  = w;
      physHeight = h / (resX / resY);
   }else
   {
      physWidth  = w / (resY / resX);
      physHeight = h;
   }
}else
{
   physHeight = h;
   physWidth  = w;
}
```

Note that the above calculation uses integer arithmetic. Note also that you don't need to consider the physical size when inserting such an image. The physical size should be used to determine the paper format when converting scanned faxes to PDF since the logical format produces often no exact match with available paper formats when calculated with the given resolution information.

Bitmap images

Bitmaps support alpha transparency but most image viewers discard the alpha channel. This can lead to diffent results since DynaPDF considers the alpha channel in 32 bit bitmaps by default.

Since many 32 bit bitmaps contain an invalid alpha channel that is zero for all pixels, it is possible to fully discard the alpha channel as most image viewers would do. This can be achieved by setting the flag gfNoBitmapAlpha with SetGStateFlags().

GIF images

GIF images are LZW compressed images which are widely used in the internet. LZW is a loss-less compression filter. Non-transparent GIF images are recompressed with the current compression filter when inserting the image. The optimal compression filter for GIF images is Flate, and therefore, the compression filter should be set to Flate before inserting such an image (see SetCompressionFilter()).

Transparent GIF images are handled differently. GIF images use color key masking to achieve the transparency effect. This kind of masking depends on exact color values and therefore, it is not possible to compress such images with JPEG or JPEG 2000 compression without invalidating the transparency information. To achieve correct results DynaPDF changes the compression filter for transparent GIF images automatically to Flate and the image will not be downscaled independent of the used settings.

Note also that transparent GIF images will be inserted transparent independent of the current transparency settings. DynaPDF considers also the image resolution information if the image uses different values for the x- and y-axis.

How to calculate the image size?

The calculation of the image size should be easy as possible. In most cases, images must be inserted with exact proportions. Therefore, all image functions support two special values to make the calculation easier. The width and height can be calculated as follows:

- If *ScaleWidth* or *ScaleHeight* is **-1** the function uses the original width or height from the image. If both parameters are **-1** the image will be inserted with a resolution of 72 DPI.

- If *ScaleWidth* **or** *ScaleHeight* is 0, the missing value is calculated in relation to the given value of *ScaleHeight* or *ScaleWidth* to preserve the image's aspect ratio. The resulting output is an image with exact proportions relative to its original size.

- If *ScaleWidth* **and** *ScaleHeight* is 0, the original size is used (same effect as -1).

- A negative value of *Width* or *Height* mirrors the image on the x- and or y-axis.

Image Resolution

Images will be scaled to the defined resolution (see SetResolution()) if the original size is larger than required, and if the property *SaveNewImageFormat* is set to true (default).

Images are downscaled with a high quality scaling algorithm to avoid loss of image quality. However, 1 bit images are never downscaled because the loss of quality can be very large.

If the image resolution is only a little bit larger than required, the new image can require more disk space than the original unscaled image. This is an effect of the color interpolation because scaled images contain usually more colors than the original image.

However, to avoid the above effect the original image must be at least 5% larger than required before the image will be scaled.

Transparent images (Color Key Masking)

DynaPDF masks images without an alpha channel with color key masking by default. Color key masking is natively supported in GIF images only. Therefore, it is recommended to disable image transparency by default since it is automatically applied on GIF images if necessary.

That color key masking is enabled by default is an artifact of older DynaPDF versions which didn't support alpha transparency. Use the function SetUseTransparency() to enable or disable color key masking.

Notice:

> If an image will be downscaled to a lower resolution, it is possible that the transparent color does no longer produce the wished transparency effect. This can occur due to color interpolation during downscaling. To avoid such issues set the resolution to a higher value or disable downscaling with the function SetSaveNewImageFormat(). Color key masking does also not properly work with JPEG or JPEG 2000 compressed images since these filters are lossy filters which change color values.

Images with an alpha channel

Images which contain an alpha channel are always inserted transparent, independent of the current value of SetUseTransparency(). If an image with an alpha channel should appear opaque then draw a rectangle in the wished background color in background of it (just before inserting the image).

Remarks:

To insert an image from a file buffer use the function InsertImageFromBuffer(). It is possible to use one image on several pages or positions. To insert an image multiple times use the function PlaceImage(). However, once an image was inserted, the physical dimension cannot be changed anymore; it can only be scaled to different sizes. If an image should be used with different dimensions, insert the largest version first or insert it again with this function.

DynaPDF checks whether an image was already inserted before inserting an image again. The duplicate check is done by using a MD5 hash of the file name and the used parameters to insert the image. If all parameters are the same, the already available image is used instead.

Return values:

If the function succeeds the return value is the image handle, a value greater or equal zero. If the function fails the return value is a negative error code.

InsertImageFromBuffer

Syntax:

```
SI32 pdfInsertImageFromBuffer(
   const PPDF* IPDF,    // Instance pointer
   double PosX,         // X-Coordinate of destination rectangle
   double PosY,         // Y-Coordinate of destination rectangle
   double ScaleWidth,   // Scaled width of destination rectangle
   double ScaleHeight,  // Scaled height of destination rectangle
   const void* Buffer,  // Pointer to image buffer
   UI32 BufSize,        // Buffer size
   UI32 Index)          // Image index of a multi page image
```

The function inserts an image in exactly the same way as InsertImageEx(), but it accepts a file buffer as input. A specific image of a multi-page image can be selected with the parameter *Index*; the first image is denoted by the index 1. If the image file is not a multi-page image, the parameter *Index* will be ignored. To determine the number of images stored in an image file use the function GetImageCount() or GetImageCountEx(). The usage of the function is described in detail at InsertImageEx().

Remarks:

DynaPDF applies always a duplicate check when inserting an image. To enable an efficient duplicate check a MD5 hash is build from all image parameters and from the image buffer. Although the calculation of a MD5 hash is very fast, it is significantly slower in comparison to file based images since only the file name must be taken into account in this case. To improve processing speed the usage of InsertImageEx() should be preferred.

Return values:

If the function succeeds the return value is the image handle, a value greater or equal zero. If the function fails the return value is a negative error code.

InsertMetafile

Syntax:

```
LBOOL pdfInsertMetafile(
   const PPDF* IPDF,     // Instance pointer
   const char* FileName, // File path to EMF or WMF file
   double PosX,          // X-Coordinate of output rectangle
   double PosY,          // Y-Coordinate of output rectangle
   double Width,         // Width of output rectangle
   double Height)        // Height of output rectangle
```

The function converts an Enhanced Meta File (EMF) or Windows Meta File (WMF) to native PDF vector graphics. The function requires an open page or template into which the metafile can be drawn (see Append() or EditPage()). If a metafile should be used as fixed background on multiple pages, then insert it into a template and place this template on the pages where it should be used (see BeginTemplate() for further information). It is of course also possible to insert such a metafile on each page separately, but this would waste processing time and disk space.

The point *PosX*, *PosY* defines the lower left corner of the output rectangle if the coordinate system is bottom-up and the upper left corner otherwise.

Bounding box check

The function checks whether the resolution of the EMF file seems to be larger as 1800 DPI. If this is the case then the rclBounds rectangle is used to calculate the picture size since this is mostly an indication that the rclFrame rectangle was incorrectly calculated. If you need to process EMF files in a higher resoultion as 1800 DPI then disable the bounding box check with the flag mfNoBBoxCheck. See SetMetaConfFlags() for further information.

How to calculate the image size?

The calculation of the image size should be easy as possible. In most cases, metafiles must be inserted with exact proportions. Therefore, all metafile functions support two special values to make the calculation easier. The width and height can be calculated as follows:

- If *Width* or *Height* is **-1** the function uses the original width or height from the image. If both parameters are **-1** the metafile will be inserted with a resolution 72 DPI. However, this is not meaningful for metafiles because the logical size of a metafile is usually too large as if it could be used. You should not produce EMF files in a resolution of 72 DPI.

- If *Width* **or** *Height* is 0, the missing value is calculated in relation to the given value of *Height* or *Width* to preserve the metafile's aspect ratio. The resulting output is a vector graphic with exact proportions relative to its original size.

- If *Width* **and** *Height* is 0, the original size is used (same effect as -1, should not be used).

- Negative values of *Width* or *Height* are ignored; the function uses always the absolute values.

Font selection in EMF files

When selecting a font with the GDI function CreateFont(), the GDI selects a font that is a good match in comparison to the used parameters. The highest search priority has the Charset, followed by PitchAndFamily. If one of these parameters are not supported by the requested font, then the GDI selects an arbitrary other font that represents a better match.

To get a close match to what the GDI would select, the flag mfGDIFontSelection can be set (see SetMetaConvFlags() for further information). DynaPDF uses then a device context to select fonts. This produces often better results since many EMF files use invalid font names or other invalid parameters.

In general, DynaPDF checks whether the requested glyphs are available in the font. If one or more glyphs are missing then the selected font will be loaded in Unicode mode if it was loaded with a code page. If still glyphs are missing then the font will be changed to Arial Unicode MS. This fallback was mainly added for Asian EMF files which use often wrong charsets or fonts which do not support Asian characters.

Character sets

The charset parameter of a CreateFont() function should be set correctly so that DynaPDF can directly load a font with the required code page or Unicode:

- Ansi_CHARSET: The font will be loaded with the code page 1252. This saves disk space because Ansi strings require only one byte per character. However, if the same font must also be used with other encodings then use DEFAULT_CHARSET instead.

- DEFAULT_CHARSET: The font is directly loaded in Unicode mode. This charset must be used for any non-Latin charset.

- SYMBOL_CHARSET: This charset must be used for symbol fonts. Symbol codes start at 0xF000. The mapping in EMF file is often wrong. DynaPDF corrects this automatically.

- All other character sets are treated in Unicode mode. EMF files do not contain strings in other encodings.

- For Asian EMF files it is recommended but not required to set the flag mfUseUnicode. This flag makes sure that a font will not be loaded with the code page 1252 which is mostly not meaningful in Asian EMF files. It slightly improves the processing speed if a file contains many create font records with invalid charset information.

Non-portable WMF files

Two WMF file formats are available, the old non-portable WMF format and the newer portable WMF format. Both formats are converted to EMF before they can be converted to PDF. DynaPDF uses the Windows API function SetWinMetaFileBits() to convert WMF files. Because of this, WMF files are not supported on Mac OS X, Linux or UNIX.

However, the old non-portable WMF format is device depended; it contains no size information in the file's header so that this file type must be handled separately. The conversion function SetWinMetaFileBits() requires a parameter of the type METAFILEPICT, this structure is only initialized to default values by DynaPDF for non-portable WMF files:

```
METAFILEPICT pict;
pict.hMF  = NULL;
pict.mm   = MM_ANISOTROPIC;
pict.xExt = 0;
pict.yExt = 0;
```

The picture size is set to zero so that the GDI must calculate the size. WMF files converted in this way are often stretched. To get correct output results you must set the width and height of the WMF file manually with the function SetWMFDefExtent(). The size must be calculated in 0.01 millimetre units. A widely used size is 210000 x 280000 units for high resolution metafiles.

Notice: Missing lines or other objects indicates that the output size is too small. The GDI function SetWinMetaFileBits() removes records which would be invisible or too small to appear in the requested size. Set the output size to a larger value in this case.

Portable WMF files

The usage of portable WMF files is the same as of normal EMF files. However, many people want or must to personalize WMF files with additional contents and many of them have problems to calculate the correct WMF picture size which is required to get a correct EMF file. Let's see how DynaPDF calculates the size of a WMF file:

First, we need the header of the WMF file because the structure contains the size of the WMF file. The WMF header is defined as follows:

```
struct TRectS
{
   SI16 Left;
   SI16 Top;
   SI16 Right;
   SI16 Bottom;
};
#include <pshpack2.h>   // packed structure
struct TPORT_METAHEADER // 16 bit portable WMF file
{
   UI32   Key;       // WMF identifier (must be 0x9AC6CDD7)
   SI16   Handle;    // Number of handles in file
   TRectS BBox;      // Bounding rectangle
   UI16   Inch;      // Pixels per inch
   UI32   Reserved;
   UI16   CheckSum;  // Aldus checksum
};
#include <poppack.h>
```

The header files pshpack2.h and poppack.h are available in Visual Studio and Embarcadeo's C++ Builder; they are used to declare a packed structure. Fill the structure TPORT_METAHEADER now with values:

```
TPORT_METAHEADER wmf;
fread(&wmf, 1, sizeof(wmf), f);
if (wmf.Key != 0x9AC6CDD7) // Is this a portable WMF file?
{
   fclose(f);
   return -2;
}
```

Now we can calculate the size of the metafile picture:

```
METAFILEPICT pict;
if (!wmf.Inch) wmf.Inch = 96;
pict.hMF  = NULL;
pict.mm   = MM_ANISOTROPIC;
pict.xExt = (wmf.BBox.Right - wmf.BBox.Left) * 2540 / wmf.Inch;
pict.yExt = (wmf.BBox.Bottom - wmf.BBox.Top) * 2400 / wmf.Inch;
```

As you can see above, the y-axis is not calculated with 2540 pixels per inch. DynaPDF uses 2400 pixels instead. I don't know why it must be 2400 pixels in most cases, but the y-axis is often stretched otherwise. This value can be adjusted in the range 2000 to 3000 with the property *WMFPixelPerInch.* This property changes the value of the y-axis only; the value of the x-axis is always 2540.

Notice: The adjustment above is maybe not required. The value 2400 pixels per inch was determined via trial and error.

ROP Codes (Raster Operation Codes)

EMF files are mostly created for a raster device like a monitor or printer. Because of this, metafiles use often ROP codes which combine fore- and background colors to achieve transparency or other color effects. Such ROP codes are meaningful on a raster device but not on a vector device like PDF. Because no raster image is created during conversion, it is not possible to use ROP codes which combine fore- and background colors.

The function checks for unsupported ROP codes during conversion. If unsupported ROP codes will be used then the EMF file will be rendered to an image by default. The problem with ROP codes is that the result depends mainly on the background. Many unsupported ROP codes have no effect if the background is empty. In most cases it is better to disable the rasterizer with the flag mfDisableRasterEMF and to enable it only if the output is wrong. See SetMetaConvFlags() for further information.

Notice:

When an EMF file is rendered to an image then it will be rendered in the resolution that was set with SetResolution(). The default resolution is 150 DPI which is usually too low to achieve good results. For good results the resolution should be set to 300 or 600 DPI.

How to convert spool EMF files?

EMF spool files contain EMF files for every page of a GDI print job. The spool file can also contain embedded fonts which are required by the EMF files. These fonts must be loaded in a pre-conversion step before the first EMF file can be converted. Embedded font subsets and corresponding delta font records must also be converted to regular TrueType fonts.

The function ConvertEMFSpool() can be used to convert a complete spool file to PDF, or if you want to use your own spool file parser, it can be used to load the embedded fonts from the spool file. Note that you must set the flags mfUseUnicode and mfIgnoreEmbFonts with SetMetaConvFlags() if only the spool fonts should be loaded. These flags are required in this case!

If you want to do anything manually with your own code then make sure that embedded font subsets and corresponding delta fonts will be converted back to regular TrueType fonts with Microsoft's function MergeFontPackage() of the fontsub.dll.

The resulting fonts as well as all other embedded fonts must be loaded with LoadFont() with the code page cpGlyphIndexes or cpUnicode (cpGlyphIndexes is preferred). In addition, the flags mfUseUnicode and mfIgnoreEmbFonts must be set with SetMetaConvFlags().

The code page cpGlyphIndexes or cpUnicode as well as the flag mfUseUnicode must be set to avoid collusions with the internal font selection during EMF conversion. The flag mfIgnoreEmbFonts makes sure that DynaPDF does not load the same embedded fonts again which are available in GDIComment records.

Compatibility Note:

DynaPDF versions prior 2.5.0.516 were not able to load spool fonts automatically. It was required to load the fonts in the user's temp directory manually with AddFontSearchPath() and the font cache had to be cleared before the next spool file could be converted. This technique was rather inefficient and is no longer supported. Existing applications should be changed to avoid the unnecessary calls of AddFontSearchPath() and ClearHostFonts(). AddFontSearchPath() does no longer load fonts with the extension tmp.

Remarks:

DynaPDF supports several flags to convert metafiles files to PDF, see SetMetaConvFlags() for further information. Because the GDI function SetWinMeatFileBits() is used to convert WMF files to EMF, WMF files are not supported on Mac OS X, Linux or UNIX.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InsertMetafileEx

Syntax:

```
LBOOL pdfInsertMetafileEx(
    const PPDF* IPDF,    // Instance pointer
    const void* Buffer,  // Metafile buffer
    UI32 BufSize,        // Buffer length
    double PosX,         // X-Coordinate of output rectangle
    double PosY,         // Y-Coordinate of output rectangle
    double Width,        // Width of output rectangle
    double Height)       // Height of output rectangle
```

The function converts an Enhanced Meta File (EMF) or Windows Meta File (WMF) to a native PDF vector graphic in the same way as InsertMetafile(). However, this function requires a file buffer that was returned by the API function GetEnhMetaFileBits(), GetWinMetaFileBits() or GetMetaFileBitsEx(). *Buffer* can also be a file buffer of an EMF or WMF file.

This function is mostly used to convert EMF files which were already loaded into memory or displayed on screen. Those memory based EMF or WMF files can directly be converted to PDF as follows:

In this example we have already a handle of an enhanced metafile. This handle can be used to get the file buffer of the EMF file:

Example 1 (C++):

```
// First, we need the size of the file buffer
SI32 bufSize = GetEnhMetaFileBits(emf, 0, NULL);
BYTE* buffer = (BYTE*)malloc(bufSize); // Allocate memory
if (!buffer) return -1; // Out of memory?
// Now we can get the file buffer
bufSize = GetEnhMetaFileBits(emf, bufSize, buffer);
// We convert the buffer directly to PDF
pdfSetPageCoords(pdf, pcTopDown);
pdfInsertMetafileEx(pdf, buffer, bufSize, 0, 0, pdfGetPageWidth(pdf), 0);
// Free the buffer
free(buffer);
```

Example 2 (Delphi):

```
// emf is a TMetafile; we store the buffer in a TMemoryStream
emf.SaveToStream(stream);
pdf.InsertMetafileEx(stream.Memory,stream.Size,0,0,pdf.GetPageWidth,0);
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InsertMetafileExt

Syntax:

```
LBOOL pdfInsertMetafileExt(
    const PPDF* IPDF,      // Instance pointer
    const char* FileName,  // File path to EMF or WMF file
    struct TRectL* View,   // Optional view rectangle (can be NULL)
    double PosX,           // X-Coordinate of output rectangle
    double PosY,           // Y-Coordinate of output rectangle
    double Width,          // Width of output rectangle
    double Height)         // Height of output rectangle

struct TRectL
{
    SI32 Left;
    SI32 Top;
    SI32 Right;
    SI32 Bottom;
};
```

The function converts an Enhanced Meta File (EMF) or Windows Meta File (WMF) to a native PDF vector graphic in the same way as InsertMetafile(). However, the function supports an additional parameter *View* which can be used to zoom into an EMF or WMF file.

How the *View* rectangle must be calculated is described in detail under the function GetLogMetafileSize(), this function is always required for the calculation.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InsertMetafileExtEx

Syntax:

```
LBOOL pdfInsertMetafileExtEx(
    const PPDF* IPDF,      // Instance pointer
    const void* Buffer,    // Metafile buffer
    UI32 BufSize,          // Buffer length
    struct TRectL* View,   // Optional view rectangle (can be NULL)
    double PosX,           // X-Coordinate of output rectangle
    double PosY,           // Y-Coordinate of output rectangle
    double Width,          // Width of output rectangle
    double Height)         // Height of output rectangle

struct TRectL
{
    SI32 Left;
    SI32 Top;
    SI32 Right;
    SI32 Bottom;
};
```

The function converts an Enhanced Meta File (EMF) or Windows Meta File (WMF) to a native PDF vector graphic in the same way as InsertMetafileExt(). However, this function requires a file buffer

that was returned by the API function GetEnhMetaFileBis(), GetWinMetaFileBits() or GetMetaFileBitsEx(). *Buffer* can also be a file buffer of an EMF or WMF file.

The usage of the function is described at InsertMetafile(). How the *View* rectangle must be calculated is described in detail at GetLogMetafileSize(), this function is always required for the calculation (use GetLogMetafileSizeEx() instead, it supports a file buffer such as this function).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InsertMetafileFromHandle

Syntax:

```
LBOOL pdfInsertMetafileFromHandle(
    const PPDF* IPDF,          // Instance pointer
    const void* hEnhMetafile,  // Enhanced Metafile Handle
    double PosX,               // X-Coordinate of output rectangle
    double PosY,               // Y-Coordinate of output rectangle
    double Width,              // Width of output rectangle
    double Height)             // Height of output rectangle
```

This function inserts an Enhanced Metafile exactly in the same way as InsertMetafile() but accepts a HENHMETAFILE handle as input. See also InsertMetafile().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InsertMetafileFromHandleEx

Syntax:

```
LBOOL pdfInsertMetafileFromHandleEx(
    const PPDF* IPDF,          // Instance pointer
    const void* hEnhMetafile,  // Enhanced Metafile Handle
    struct TRectL* View,       // Optional view rectangle (can be NULL)
    double PosX,               // X-Coordinate of output rectangle
    double PosY,               // Y-Coordinate of output rectangle
    double Width,              // Width of output rectangle
    double Height)             // Height of output rectangle
```

This function inserts an Enhanced Metafile exactly in the same way as InsertMetafileExt() but accepts a HENHMETAFILE handle as input. See also InsertMetafileExt().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

InsertRawImage

Syntax:

```
SI32 pdfInsertRawImage(
    const PPDF* IPDF,    // Instance pointer
    const void* Buffer,  // Image buffer
    UI32 BitsPerPixel,   // Color depth in bits per pixel
    UI32 ColorCount,     // Number of colors in color table if any
    SI32 ImgWidth,       // Image width
    SI32 ImgHeight,      // Image height (negative values mirror the image)
    double PosX,         // X-Coordinate of destination rectangle
    double PosY,         // Y-Coordinate of destination rectangle
    double ScaleWidth,   // Scaled width of destination rectangle
    double ScaleHeight)  // Scaled height of destination rectangle
```

```
typedef enum
{
    rifByteAligned  = 0x1000, // Scanlines are byte aligned
    rifRGBData      = 0x2000, // RGB data format, instead of BGR
    rifCMYKData     = 0x4000  // CMYK image data or color table
}TRawImageFlags;
```

The function inserts an image from a raw image buffer that contains no image header. The parameter *Buffer* must be a valid pointer to the image data.

The scanlines of the image must be double word aligned by default. This is the default format for Device Independent Bitmaps (DIBs).

The scanline length can be calculated as follows:

```
scanLineLen = ((Width * BitsPerPixel + 31) / 32) * 4;
```

Or more efficient in C notation:

```
scanLineLen = ((Width * BitsPerPixel + 31) & ~31) >> 3;
```

DynaPDF 2.5 introduced several extensions which affect the data alignment, supported pixel formats, and color spaces.

Flag	Description
rifByteAligned	Scanlines are byte aligned. If this flag is set, the scanline length is calculated as follows: `scanLineLen = (Width * BitsPerPixel + 7) / 8;`
rifRGBData	The image data is defined in RGB format instead of BGR. If *BitsPerPixel* is less than 16 a color table in RGB format must precede the image data.
rifCMYKData	The image data is defined in CMYK. *BitsPerPixel* must be 1, 2, 4, 8, or 32 in this case. If *BitsPerPixel* is less than 32 a color table in CMYK format must precede the image data.

The flags must be added to the parameter *ColorCount* with a binary or operator:

```
UI32  colorCount  = 256 | rifRGBData | rifByteAligned;    // C/C++, C#
long  colorCount  = 256 or rifRGBData or rifByteAligned   // Visual Basic
DWORD colorCount := 256 or rifRGBData or rifByteAligned;  // Delphi
```

The above example specifies that the image data starts with a color table with 256 RGB colors and the following scanlines are byte aligned.

If the image has a color depth of 24 or 32 bits per pixel the pixel data must be defined in BGR or BGRA (BGR Alpha) or RGB or RGBA format depending on whether the flag rifRGBData was set. The alpha component will be converted to a soft mask to preserve the alpha channel. However, alpha components in a color table will be ignored at this time.

If a 32 bit image should be interpreted as CMYK image then set the flag rifCMYKData.

A color table in BGRA or RGB format must precede the image data if *BitsPerPixel* is 4 or 8 and depending on whether the flag rifRGBData was set. The parameter *ColorCount* specifies the number of colors in the color table. It is also possible to insert a gray scale image. *BitsPerPixel* must be 8 in this case, and *ColorCount* must be zero. No color table must precede the image data in this case.

If the color depth is 1, 16, 24, or 32 bits per pixel no color table must be present. The parameter *ColorCount* will be ignored for these images, it should be set to 0.

If the image height is a positive value the image is treated from bottom to top. Top-down images can be mirrored by setting the image height to a negative value.

Color spaces

If the active color space is a device color space the image will be converted into that color space if necessary (with exception of 1 bit images).

For all other color spaces the image data must be defined in the active color space. 1 bit images can also be used with Separation color spaces. See also Color Space and Images. However, all other color spaces require 8 bits per color component.

A color table cannot be prepended if the image is defined in a non device color space but it is possible to create an indexed color space for the image (see CreateIndexedColorSpace()) and to activate it with SetExtColorSpace() before the image will be inserted. The image data must contain 8 bit indexes into the color table in this case.

If the active color space is DeviceN, the image can contain up to 32 color components (256 bits per pixel). Images with more than 4 color components are always Flate compressed.

How to calculate the image size?

The calculation of the image size should be easy as possible. In most cases, images must be inserted with exact proportions. Therefore, all image functions support two special values to make the calculation easier. The width and height can be calculated as follows:

- If *ScaleWidth* or *ScaleHeight* is **-1** the function uses the original width or height from the image. If both parameters are **-1** the image will be inserted with a resolution of 72 DPI.

- If *ScaleWidth* **or** *ScaleHeight* is **0**, the missing value is calculated in relation to the given value of *ScaleHeight* or *ScaleWidth* to preserve the image's aspect ratio. The resulting output is an image with exact proportions relative to its original size.

- If *ScaleWidth* **and** *ScaleHeight* are **0**, the original size is used (same effect as -1).

- A negative value of *Width* or *Height* mirrors the image on the x- or y-axis.

The function contains no parameter to define the buffer length. However, the required length is calculated from the parameters of the function.

The pixel data must be uncompressed. Raw images are handled in the same way as other image formats; they are downscaled if necessary, converted to the current color space and compressed with the defined compression filter. Note that 1 bit images are treated as image mask if transparency is enabled (see SetUseTransparency()), see InsertImage() for further information.

Remarks:

InsertRawImageEx() is more flexible regarding the number of color components, specific scanline lengths and alpha channels in multi-channel images.

It is possible to use one image on several pages or positions. To insert an image multiple times use the function PlaceImage(). However, once an image was inserted, the physical dimension cannot be changed; it can only be scaled to different sizes. If an image should be used with different dimensions, insert the largest version first.

Return values:

If the function succeeds the return value is the image handle, a value greater or equal zero. If the function fails the return value is a negative error code.

InsertRawImageEx

Syntax:

```
SI32 pdfInsertRawImageEx(
   const PPDF* IPDF,            // Instance pointer
   double PosX,                 // X-Coordinate of destination rectangle
   double PosY,                 // Y-Coordinate of destination rectangle
   double ScaleWidth,           // Scaled width of destination rectangle
   double ScaleHeight,          // Scaled height of destination rectangle
   struct TPDFRawImage* Image)  // see below

struct TPDFRawImage
{
   UI32         StructSize;       // Must be set to sizeof(TPDFRawImage)
   const void*  Buffer;           // Image buffer
   UI32         BufSize;          // Buffer size
   UI32         BitsPerComponent; // Bits per component
   UI32         NumComponents;    // Number of components (max 32)
   TExtColorSpace CS;             // Image color space
```

```
    SI32         CSHandle;         // Color space handle or -1
    SI32         Stride;           // A negative value mirrors the image
    LBOOL        HasAlpha;         // The image has an alpha channel?
    LBOOL        IsBGR;            // Windows bitmap format?
    LBOOL        MinIsWhite;       // 1 bit images only
    UI32         Width;            // Width in pixels
    UI32         Height;           // Height in pixels
};
```

The function inserts an image from a raw image buffer that contains no image header. Unlike InsertRawImage() the function supports also multi-channel images with less than 8 bits per component as well as arbitrary aligned scanlines.

Images with 1, 2, 4, 8, and 16 bits per component are supported. 2 and 4 bits per component images can be used with Indexed and DeviceN color spaces. The corresponding color space must be created beforehand and the color space handle must be set to *CSHandle*. No color table must precede the image data.

1 bit images with only one component are treated as an image mask if SetUseTransparency() is set to true (default) and if the color space of the image is set to esDeviceGray. An image mask is drawn with the current fill color. Zero pixel values produce no output.

Images with 8 bits per component can contain an alpha channel. The parameter *HasAlpha* must be set to true in this case and the alpha channel must be assigned to the last component. The image can contain up to 32 components including the alpha channel.

If the alpha channel is stored in a separate image buffer then use AddMaskImage() to add the alpha channel. The function supports also the creation of a 1 bit image mask.

The scanline length (*Stride*) can be a positve or negative value. If negative, the image is read from bottom to top.

If *IsBGR* is true and if the color space is set to esDeviceRGB, the color components are interpreted in BGR order. This component order is mainly used in Windows Bitmaps.

If *MinIsWhite* is set to true, zero pixel values are interpreted as white. This parameter applies only to images with 1 bit per component.

The function does not use the active color space and it performs no color conversion.

Color Key Masking

Images which use a device color space are treated as usual. That means, if SetUseTransparency() is set to true (default), then the image is masked with the current transparent color (see SetTransparentColor()).

Images of other color spaces can be masked too with SetColorMask(). Color key masking depends on exact color values. To avoid issues due to color interpolation the image should be compressed with Flate encoding. See SetCompressionFilter() for further information.

How to calculate the image size?

The calculation of the image size should be easy as possible. In most cases, images must be inserted with exact proportions. Therefore, all image functions support two special values to make the calculation easier. The width and height can be calculated as follows:

- If *ScaleWidth* or *ScaleHeight* is **-1** the function uses the original width or height from the image. If both parameters are **-1** the image will be inserted with a resolution of 72 DPI.

- If *ScaleWidth* **or** *ScaleHeight* is **0**, the missing value is calculated in relation to the given value of *ScaleHeight* or *ScaleWidth* to preserve the image's aspect ratio. The resulting output is an image with exact proportions relative to its original size.

- If *ScaleWidth* **and** *ScaleHeight* are **0**, the original size is used (same effect as -1).

- A negative value of *Width* or *Height* mirrors the image on the x- or y-axis.

Remarks:

It is possible to use one image on several pages or positions. To insert an image multiple times use the function PlaceImage(). However, once an image was inserted, the physical dimension cannot be changed; it can only be scaled to different sizes. If an image should be used with different dimensions, insert the largest version first.

Return values:

If the function succeeds the return value is the image handle, a value greater or equal zero. If the function fails the return value is a negative error code.

IsBidiText

Syntax:

```
SI32 pdfIsBidiText(
   const PPDF* IPDF,   // Instance pointer
   const UI16* AText)  // Unicode string
```

The function returns the position of the first bidirectional character that can be found in the string or **-1** if no such character can be found.

IsColorPage

Syntax:

```
SI32 pdfIsColorPage(
   const PPDF* IPDF,    // Instance pointer
   LBOOL GrayIsColor)   // Treat gray as color
```

This function checks whether a page is a color page or if all graphic elements of the page use black & white only. If the parameter *GrayIsColor* is true, gray shades are treated as color. The page which should be checked must be opened with the function EditPage() beforehand.

Complex color spaces such as Lab, Separation, DeviceN, Colored Tiling Patterns, and Shadings are always treated as color. The function does not return when a color is set, the color must be used by an object.

Return values:

If an object of the page uses a color, the return value is 1. If no object uses a color the return value is 0. A negative return value indicates that an error occurred.

IsEmptyPage

Syntax:

```
SI32 pdfIsEmptyPage(
   const PPDF* IPDF) // Instance pointer
```

The function checks whether a page is empty. The page which should be checked must be opened with the function EditPage() beforehand. The function does not check whether the buffer size of a page is zero to determine whether a page is empty. Instead, the content stream will be parsed until a visible object can be found.

Return values:

If no visible object can be found the return value is 1. If a visible object can be found the return value is 0. A negative return value indicates that an error occurred.

IsWrongPwd

Syntax:

```
#define IsWrongPwd(ErrCode)                              // C/C++
Function IsWrongPwd(ByVal ErrCode As Integer) As Boolean // Visual Basic
function IsWrongPwd(ErrCode: Integer): Boolean;          // Delphi
```

The function returns true is the supplied error code indicates that a password is required to decrypt the PDF file. The error code must be a return value of the function OpenImportFile() or OpenImportBuffer().

This function is implemented as macro in C/C++. In prior DynaPDF versions the macro name was PDF_WRONG_PWD() which is still defined. However, for consistency among different programming languages the new macro name IsWrongPwd() should be used.

LineAnnot

Syntax:

```
SI32 pdfLineAnnot(
    const PPDF* IPDF,      // Instance pointer
    double x1,             // X-Coordinate of the start point
    double y1,             // Y-Coordinate of the start point
    double x2,             // X-Coordinate of the end point
    double y2,             // Y-Coordinate of the end point
    double LineWidth,      // Line width
    TLineEndStyle Start,   // End line style of the start point
    TLineEndStyle End,     // End line style of the end point
    UI32 FillColor,        // Text and interiour color -> see description
    UI32 StrokeColor,      // This is the line color
    TPDFColorSpace CS,     // The color space in which colors are defined
    const char* Author,    // Optional author
    const char* Subject,   // Optional subject
    const char* Content)   // Optional content or caption of a measure line

typedef enum
{
    leNone,
    leButt,
    leCircle,
    leClosedArrow,
    leDiamond,
    leOpenArrow,
    leRClosedArrow,
    leROpenArrow,
    leSlash,
    leSquare
}TLineEndStyle;
```

The function creates a line annotation. The simplest form of a line annotation represents a simple straight line that has an associated PopUp annotation to display the string *Content* in a floating window.

The initial window state of the associated PopUp annotation is closed by default but the state can be changed with SetAnnotOpenState() if necessary.

The coordinates are interpreted in current user space. Any transformation that was applied on the coordinate system will be taken into account.

The parameter *FillColor* is only used if the line end style of the start or end point has an interior that can be filled. The special constant NO_COLOR represents a transparent interior.

The stroke color is required and must not be set to NO_COLOR.

Measure lines

A measure line is an extended line annotation that has additional properties. To create a measure line create first the line annotation and set then the measure line specific properties with SetLineAnnotParms().

The parameter *LineWidth* must be in the range 0 through 12 units. Values outside the valid range will be adjusted to the nearest allowed value. A zero line width produces a 1 pixel wide line.

The line end styles can be changed if necessary with SetAnnotLineEndStyle().

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

LineTo

Syntax:

```
LBOOL pdfLineTo(
   const PPDF* IPDF, // Instance pointer
   double PosX,      // X-Coordinate of ending point
   double PosY)      // Y-Coordinate of ending point
```

The function draws a path from the current position up to the specified point. The start point must be set with another vector graphic function beforehand, such as MoveTo() or other functions which draw an open path segment.

In PDF all vector graphics are defined as paths, a path is invisible as long it was not stroked, filled or both. See also Path construction and Painting.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

LoadCMap

Syntax:

```
SI32 pdfLoadCMap(
  const PPDF* IPDF,      // Instance pointer
  const char* CMapName, // CMap name
  LBOOL Embed)           // See description below
```

The function loads an external CMap file into memory so that it can be used with OpenType or TrueType fonts. The returned CMap handle is a required parameter of the function SetCIDFont().

Embedding CMap Files

The CMap should be embedded if it is not predefined or if the supplement number is higher than supported in the viewer version to which compatibility is required.

However, dependencies to external CMaps are always removed if the font, that uses the external CMap, is embedded. The parameter *Embed* is ignored in this case. This handling reduces the amount of data that must be stored in a PDF file and the file can be viewed with Adobe's Acrobat or Reader without installed Asian language pack that would otherwise be required.

Predefines CMaps

The following table lists all predefined CMaps and the corresponding character collections which are supported in a specific PDF version. The corresponding Acrobat versions are Acrobat 3 for PDF 1.2, Acrobat 4 for PDF 1.3 and so on.

CMap	PDF 1.2	PDF 1.3	PDF 1.4	PDF 1.5
Chinese (Simplified)				
GB-EUC-H/V	Adobe-GB1-0	Adobe-GB1-0	Adobe-GB1-0	Adobe-GB1-0
GBpc-EUC-H	Adobe-GB1-0	Adobe-GB1-0	Adobe-GB1-0	Adobe-GB1-0
GBpc-EUC-V	–	Adobe-GB1-0	Adobe-GB1-0	Adobe-GB1-0
GBK-EUC-H/V	–	Adobe-GB1-2	Adobe-GB1-2	Adobe-GB1-2
GBKp-EUC-H/V	–	–	Adobe-GB1-2	Adobe-GB1-2
GBK2K-H/V	–	–	Adobe-GB1-4	Adobe-GB1-4
UniGB-UCS2-H/V	–	Adobe-GB1-2	Adobe-GB1-4	Adobe-GB1-4
UniGB-UTF16-H/V	–	–	–	Adobe-GB1-4
Chinese (Traditional)				
B5pc-H/V	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0
HKscs-B5-H/V	–	–	Adobe-CNS1-3	Adobe-CNS1-3
ETen-B5-H/V	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0
ETenms-B5-H/V	–	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0
CNS-EUC-H/V	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0
UniCNS-UCS2-H/V	–	Adobe-CNS1-0	Adobe-CNS1-3	Adobe-CNS1-3
UniCNS-UTF16-H/V	–	–	–	Adobe-CNS1-4
Japanese				
83pv-RKSJ-H	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1
90ms-RKSJ-H/V	Adobe-Japan1-2	Adobe-Japan1-2	Adobe-Japan1-2	Adobe-Japan1-2
90msp-RKSJ-H/V	–	Adobe-Japan1-2	Adobe-Japan1-2	Adobe-Japan1-2
90pv-RKSJ-H	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1
Add-RKSJ-H/V	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1
EUC-H/V	–	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1
Ext-RKSJ-H/V	Adobe-Japan1-2	Adobe-Japan1-2	Adobe-Japan1-2	Adobe-Japan1-2
H/V	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1
UniJIS-UCS2-H/V	–	Adobe-Japan1-2	Adobe-Japan1-4	Adobe-Japan1-4
UniJIS-UCS2-HW-H/V	–	Adobe-Japan1-2	Adobe-Japan1-4	Adobe-Japan1-4
UniJIS-UTF16-H/V	–	–	–	Adobe-Japan1-5
Korean				
KSC-EUC-H/V	Adobe-Korea1-0	Adobe-Korea1-0	Adobe-Korea1-0	Adobe-Korea1-0
KSCms-UHC-H/V	Adobe-Korea1-1	Adobe-Korea1-1	Adobe-Korea1-1	Adobe-Korea1-1
KSCms-UHC-HW-H/V	–	Adobe-Korea1-1	Adobe-Korea1-1	Adobe-Korea1-1
KSCpc-EUC-H	Adobe-Korea1-0	Adobe-Korea1-0	Adobe-Korea1-0	Adobe-Korea1-0
UniKS-UCS2-H/V	–	Adobe-Korea1-1	Adobe-Korea1-1	Adobe-Korea1-1
UniKS-UTF16-H/V	–	–	–	Adobe-Korea1-2

Adobe's Acrobat 9 and Reader 9 support also the character collections Adobe-GB1-5, Adobe-CNS1-5, and Adobe-Japan1-6. Adobe offers also many more CMaps which support other encodings like UTF-8 or UTF-32. However, CMaps which are not predefined must either be embedded or the font that uses the CMap must be embedded.

CMaps and the CID-Keyed Font Architecture are described in detail in the Adobe Technical Notes #5092 and #5099. Detailed information about Adobe Character Collections can be found in the Adobe Technical Notes #5094, #5093, #5078, #5079, #5080, and #5097. These documents can be downloaded from the Adobe website at www.adobe.com.

External CMaps cannot be used with Type1 fonts and DynaPDF does not support glyph names to access CIDs because this format is reserved for use with Type1 fonts.

Working with External CMaps

External CMaps can be used to specify a rich mixture of encodings with fixed and variable code lengths of up to four bytes per character. Many CJK encodings use a mixture of one- and two-byte encodings while Unicode encodings use one through four-byte representations.

When working with external CMaps it is important to understand how the mapping works in conjunction with the string functions in DynaPDF. Practically all string functions in DynaPDF are available in an Ansi and Wide string version.

The wide string version uses 16 bits or two-bytes per character. So, these functions can be used with CMaps which support a code length of two, four, or a mixture of two/four byte codes but it is not possible to access CIDs which are encoded with code lengths of one or three bytes because less than two bytes cannot be consumed from the source string.

No such restriction is given if the Ansi version is used instead because all code lengths from one through four bytes can be created from the source string.

The mapping of codes to CIDs works essentially as follows:

The CMap parser initializes a 32 bit variable with the first char code of the string. Depending on the string format the consumed code length is incremented by one or two bytes. Then it checks whether a CID is defined for this code. If this is not the case, the next char code is consumed and combined with the existing code to from a 16, 24, or 32 bit code. The search run continues until the maximum code length or string end was reached. If no CID is defined for that code the notdef character is returned.

DynaPDF does not know how a string is encoded when using a CID font with an external CMap. So, functions which output formatted text like WriteFText() can only be used if the CID font was loaded with a CMap that maps UTF-16 (Unicode) to CIDs (the wide string version must be used in this case).

Because TrueType fonts have no native notation of CIDs, the usage with CID-keyed OpenType fonts is preferred. CID-keyed OpenType fonts contain an embedded Type1 font in Compact Font Format (CFF). This font contains also information about the supported character collection. The supported character collection of the font must match the one of the CMap file. It is not possible to load an OpenType font with a CMap that contains a mapping into another character collection.

Remarks:

One CMap file can be used with an arbitrary number of CID fonts. The function checks whether a CMap was already loaded before it loads the same CMap again. A CMap specifies also the writing mode. The style flag fsVerticalMode will be ignored when loading a font with an external CMap.

Return values:

If the function succeeds the return value is a CMap handle, a value greater or equal zero. If the function fails the return value is a negative error code. The CMap handle is required to load a CID font with this CMap.

LoadFont

Syntax:

```
SI32 pdfLoadFont(
   const PPDF* IPDF,    // Instance pointer
   const void* Buffer,  // Font file buffer
   UI32 BufSize,        // Buffer size in bytes
   TFStyle Style,       // Font style
   double Size,         // Font size
   LBOOL Embed,         // If true, the font will be embedded
   TCodepage CP)        // Code page
```

The function loads a font from a file buffer and activates it in the graphics state if the function was called within an open page or template. Supported font formats are TrueType, OpenType, and Type1 fonts in PFB or PFA format.

It is also possible to load fonts directly from a font file with LoadFontEx(). This function is preferred because it is more flexible and reduces the memory usage.

The function returns the font handle on success. The font handle is required whenever the font should be set again with ChangeFont(), e.g. when opening a new page or if the font was changed to another one.

The function does not check whether the font was already loaded in a previous call. The caller must make sure that no unnecessary duplicates will be loaded. However, if the font should be used with different code pages, a separate copy of the font is required for each code page.

The parameter *Style* sets the style flags fsUnderlined, fsStriked, or fsVerticalLayout only. It is also possible to emulate font styles with manually loaded fonts. Use the function ChangeFontStyleEx() if a font style should be emulated.

If the font name is known by the caller then it is also possible to set this font later with SetFont(). However, for proper behaviour manually loaded fonts should always be set with ChangeFont().

If SetFont() is used then there is no guarantee that the function loads exactly this font because the function tries to load the best match according to the provided parameters.

Note that a font that was loaded with this function can only be used with the code page that was used to load the font. If the code page must be changed then you must load a further copy of the font. To reduce the memory usage, it is recommended to use LoadFontEx() instead, because this function loads only required parts of the font file and it provides generally a better resource handling.

TrueType Collections

To enable access on a specific font of a TrueType Collection the 4 high bits of the parameter *BufSize* are reserved for the font index. Index 0 through 15 can be encoded in this way (larger collections do probably not exist). The wished font index must be combined with the buffer size as follows:

```
bufSize |= (index << 28);                // C/C++, C#
bufSize  = bufSize or (index << 28)      // VB .Net
bufSize := bufSize or (index shl 28);    // Delphi
If index < 8 Then                        // Visual Basic
   bufSize = bufSize Or (index * &H10000000)
Else
   bufSize = bufSize Or ((index And 7) * &H10000000) Or &H80000000
End If
```

Remarks:

Like all other fonts, manually loaded fonts are unloaded when the PDF file is closed or discarded with FreePDF().

Return values:

If the function succeeds the return value is the font handle, a value greater or equal zero. If the function fails the return value is a negative error code.

LoadFontEx

Syntax:

```
SI32 pdfLoadFontEx(
   const PPDF* IPDF,      // Instance pointer
   const char* FontFile,  // Path to a font file
   UI32 Index,            // Index of a TrueType Collection
   TFStyle Style,         // Font style
   double Size,           // Font size
   LBOOL Embed,           // If true, the font will be embedded
   TCodepage CP);         // Code page
```

The function loads a font from a font file and activates it in the graphics state if the function was called within an open page or template. Supported font formats are TrueType, OpenType, and Type1 fonts in PFB or PFA format.

The function returns the font handle on success. The font handle is required whenever the font should be set again with ChangeFont(), e.g. when opening a new page or if the font was changed to another one.

While LoadFont() loads a font always explicitly, this function performs a duplicate check before the font is loaded. This enables a more efficient resource handling especially if a font must be loaded with different code pages. The function compares the absolute file path and not the font name. Therefore, it is strongly recommended to load fonts always from the same directories.

The parameter *Style* sets the style flags fsUnderlined, fsStriked, or fsVerticalLayout only. It is also possible to emulate font styles with manually loaded fonts. Use the function ChangeFontStyleEx() if a font style should be emulated.

If the font name is known by the caller then it is also possible to set the font later with SetFont(). However, for proper behaviour manually loaded fonts should always be set with ChangeFont().

If SetFont() is used then there is no guarantee that the function loads exactly this font because the function tries to load the best match according to the provided parameters.

The parameter *Index* is used to load a specific sub font of a TrueType Collection. If a normal font is loaded or if the index is unknown set the parameter to zero. Note that the parameter is used when DynaPDF performs the duplicate check.

Remarks:

A font file that was loaded with function must not be deleted or modified before the PDF file was closed or discarded.

Like all other fonts, manually loaded fonts are unloaded when the PDF file is closed or discarded with FreePDF().

The function is implemented in an Ansi and Unicode compatible variant. The Unicode version is preferred on Windows because Unicode is the default string format that DynaPDF uses for font files on this platform.

Unicode file paths are converted to UTF-8 on non-Windows operating systems.

Return values:

If the function succeeds the return value is the font handle, a value greater or equal zero. If the function fails the return value is a negative error code.

LoadFDFData

Syntax:

```
LBOOL pdfLoadFDFData(
  const PPDF* IPDF,     // Instance pointer
  const char* FileName, // File path of the FDF file
  const char* Password, // Defined for future use, should be NULL
  UI32 Flags)           // Defined for future use, should be 0
```

The function loads form data from a FDF file (Forms Data Format). The corresponding form must be imported before the function can be used. See ImportPDFFile() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

LoadFDFDataEx

Syntax:

```
LBOOL pdfLoadFDFDataEx(
  const PPDF* IPDF,      // Instance pointer
  const void* Buffer,    // FDF file buffer
  UI32 BufSize,          // Buffer size if bytes
  const char* Password,  // Defined for future use, should be NULL
  UI32 Flags)            // Defined for future use, should be 0
```

The function loads form data from a FDF file (Forms Data Format) like LoadFDFData but it accepts a file buffer instead. The corresponding form must be imported before the function can be used. See ImportPDFFile() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

LoadHeaderFooterSettings

Syntax:

```
SI32 pdfLoadHeaderFooterSettings(
  const PPDF* IPDF,                   // Instance pointer
  struct TPDFHeaderFooter* Init,      // Pointer to TPDFHeaderFooter structure
  struct TPDFHdrFtr* HFArray)         // Array of 6 TPDFHdrFtr structures
// Only constants which are considered by this function are listed.
typedef enum TPDFHdrFtrFlags
{
    hffLoadUTF8   = 8,   // Load all strings in UTF-8 format. The PHP extension uses this flag.
    hffLoadBatesN = 32, // Force to load bates number settings.
    hffLoadHF     = 64, // Force to load regular header / footer settings.
    hffSearchRun  = 128 // Scan the file for available header / footer types.
};
```

The function loads imported header / footer settings from the PDF file currently in memory. The function is able to identify header / footer settings independent of the way how they were imported, e.g. with ImportPDFFile(), ImportPage(), or ImportPageEx().

The parameter *HFArray* must be an array of six *TPDFHdrFtr* records. The function sets header / footer settings in their natural order. That means the indexes 0..2 contain the header definitions of the alignments left, center, and right. The remaining indexes 3..5 contain the footer definitions of the very same alignments.

A record whose *TextLen* is zero must be ignored. Otherwise either the Ansi or Unicode string is set. If the flag *hffLoadUTF8* is absent, the function sets usually *TextW*, and *TextA* otherwise. However, the caller must be able to process both variants. Strings are generally null-terminated.

All Ansi or single byte strings are encoded in the Windows code page 1252 unless the flag *hffLoadUTF8* was set. In the latter case UTF-8 encoded Unicode strings are returned.

Flags can be set via the member *Flags* of the TPDFHeaderFooter structure right after it was initialized with InitHeaderFooter().

Notice:

> Header / footer settings are stored in *PieceInfo* dictionaries. If the import flag *ifPieceInfo* was absent, e.g. if the flag IfImportAll was not set, then it is not possible to identify header / footer settings, since the required data was not imported. See SetImportFlags() for further information.

Header / footer types

Two different types of headers / footers exist: headers / footers which contain arbitrary text, page numbers, or dates, and headers / footers wich contain also bates numbers. The two types can be loaded and deleted independently from each other.

Loading header / footer or bates number settings

By default, the function loads whatever it finds first. However, with the flags *hffLoadBatesN* and *hffLoadHF* it is possible to force loading of either type.

Although this approach works perfectly for most applications, it is also possible to scan the file for header / footer settings without loading the data.

How to check whether a file contains headers or footers?

A pure search run can be achieved with the flag *hffSearchRun*. If this flag is set, the function searches for both header / footer types and returns as soon as both types were found. This makes it possible to check very quickly whether a file contains headers / footers, bates numbers, or none of them.

If the flag *hffSearchRun* is set the function returns one of the following values:

- `< 0: An error occurred.`
- `  0: No header / footer settings were found.`
- ` 32: The file contains bates numbers.`
- ` 64: The file contains normal headers / footers.`
- ` 96: The file contains bates numbers and heades / footers.`

As you can see above, the values 32, 64, and 96 are just combinations of the flags *hffLoadBatesN*, *hffLoadHF*, and (*hffLoadBatesN | hffLoadHF*).

Return values:

If the flag *hffSearchRun* was not set the function returns the number of records which were filled with data. Note that the array can contain empty records.

LoadLayerConfig

Syntax:

```
LBOOL pdfLoadLayerConfig(
   const PPDF* IPDF, // Instance pointer
   SI32 Index)       // See description
```

The function loads the specified layer configuration. A PDF file that contains layers (Optional Content Groups in PDF syntax) contains usually at least a default configuration dictionary. This configuration is loaded by default when the visibility state of a layer or optional content group must be determined, e.g. when rendering a page.

Use this function to load another configuration if available. To determine the number of available configurations call GetLayerConfigCount().

The parameter *Index* can be one of the following:

- -1: Load the default configuration.
- -2: If a configuration was already loaded, leave it unchanged. Load the default configuration otherwise.
- 0..Count -1: Load the specified configuration.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

LockLayer

Syntax:

```
LBOOL pdfLockLayer(
   const PPDF* IPDF, // Instance pointer
   UI32 Layer)       // Handle of an OCG
```

The functions adds a layer to the list of locked layers. The state of a locked layer cannot be changed through the user interface of a PDF viewer. However, the state of locked layers can still be changed via Javascript or SetOCGState actions.

Returns values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

MarkTemplateAsWatermark

Syntax:

```
LBOOL pdfMarkTemplateAsWatermark(
   const PPDF* IPDF, // Instance pointer
   UI32 Handle)      // Template handle
```

The function adds additional metadata to a template (XObject type Form in PDF terms, see BeginTemplate() or BeginTransparencyGroup() for further information) so that PDF editors like Adobe Acrobat are able to identify the template as watermark.

Watermarks can be deleted with Acrobat and many other PDF editors. DynaPDF can delete such watermarks too with DeleteWatermark() or Optimize().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

MovePage

Syntax:

```
LBOOL pdfMovePage(
  const PPDF* IPDF, // Instance pointer
  UI32 Source,      // Source page number
  UI32 Dest)        // Target page number
```

The function moves a page to another position in the document. Moving a page requires some processing time if the document contains many bookmarks or page links because the destinations of all links and bookmarks must be modified so that they refer still to the correct page. For example, if the first page of a document should be moved to the last page, all pages of the document must be reordered. If the document is large and if it contains a few thousands page links as well as bookmarks, this action can take a while.

The parameter *Source* must be a valid page number in the document. Page numbering starts at one. If the destination page is larger than the source page, the page number can be greater than the number of pages currently available in the document. The document is then filled with empty pages until the destination page becomes valid.

If the destination page is smaller than the source page number, both page numbers must already exist in the document.

It is also possible to exchange two pages with ExchangePage().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

MoveTo

Syntax:

```
LBOOL pdfMoveTo(
   const PPDF* IPDF, // Instance pointer
   double PosX,      // X-Coordinate of new position
   double PosY)      // Y-Coordinate of new position
```

The function moves the current position to the point specified by *PosX*, *PosY*. This function must be called before a line or curved path segment can be drawn. See also LineTo(), Bezier_1_2_3().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

MultiplyMatrix

Syntax:

```
LBOOL pdfMultiplyMatrix(
   struct TCTM* M1,        // First transformation matrix
   struct TCTM* M2,        // Second transformation matrix
   struct TCTM* NewMatrix) // Resulting transformation matrix
```

The function multiplies two transformation matrices and stores the result in the parameter *NewMatrix*. This function is required to calculate the resulting matrix in user coordinate space when using the functions GetPageText() in combination with WriteTextMatrix().

The two matrices *M1* and *M2* are multiplied as follows:

```
NewMatrix->a = M2->a * M1->a + M2->b * M1->c;
NewMatrix->b = M2->a * M1->b + M2->b * M1->d;
NewMatrix->c = M2->c * M1->a + M2->d * M1->c;
NewMatrix->d = M2->c * M1->b + M2->d * M1->d;
NewMatrix->x = M2->x * M1->a + M2->y * M1->c + M1->x;
NewMatrix->y = M2->x * M1->b + M2->y * M1->d + M1->y;
```

Matrix multiplication is not commutative - the order in which matrices are multiplied is significant.

This function is implemented as native procedure in the interfaces for Visual Basic, Visual Basic .Net and Delphi.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

NewPDF

Syntax:

```
void* pdfNewPDF(void)
```

The function creates a new PDF instance and returns the pointer of it. If there is no sufficient memory to create the PDF instance the return value is NULL.

To improve processing speed one PDF instance should be used as long as possible. A PDF instance holds a few important data structures such as the font cache and CMap cache. Rebuilding these caches is an expensive operation and should be avoided if not necessary. Using the same PDF instance to create as many PDF files as possible can significantly improve processing speed.

It is possible to use mutliple PDF instances at the same time. Each instance is fully isolated from each other and thread-safe.

Thread-safety

If multiple instances are used in different threads, make sure that code executed in callback functions, e.g. in the error callback function, is thread safe too. If unsafe code must be executed, acquire a mutex and lock the thread before calling non-thread safe code.

Thread-safety is guaranteed for every PDF instance but it is not allowed to call functions of one instance in different threads.

If a PDF instance is no longer needed then it must be deleted with DeletePDF().

Remarks:

This function is automatically called in the wrapper classes for Visual Basic, Visual Basic .Net, C#, and Delphi. There is no need to create PDF instances manually when using these programming languages.

C/C++ developers must create a PDF instance before a DynaPDF function can be executed.

OpenImportBuffer

Syntax:

```
SI32 pdfOpenImportBuffer(
   const PPDF* IPDF,      // Instance pointer
   const void* Buffer,    // Pointer to PDF file
   UI32 BufSize,          // Buffer length in bytes
   TPwdType PwdType,      // Kind of password and flags (see below)
   const char* Password)  // Password to decrypt the file

typedef enum
{
    ptOpen        = 0, // Open password
    ptOwner       = 1, // Owner password
    ptForceRepair = 2, // See description below
    ptDontCopyBuf = 4  // If set, the buffer is not copied
}TPwdType;
```

The function opens an external PDF file like OpenImportFile() but accepts a file buffer as input.

The function supports the additional flag ptDontCopyBuf to specify whether the import buffer should be copied to an internal buffer or not.

If the flag *ptDontCopyBuf* is absent then DynaPDF creates a copy of the input buffer before it will be accessed. The original buffer should be released immediately after the function returns in this case. The internal copy of the file buffer will be released when CloseImportFile() is called, when loading another PDF file, or when the PDF file in memory is closed.

If the flag *ptDontCopyBuf* is set, then the function works directly with the source buffer without creating a copy of it. This reduces the memory usage and improves the processing speed due to the unnecessary copy action. However, the caller must make sure that the buffer exist until CloseImportFile() is called. When finish, call first CloseImportFile() and then release the input buffer.

How to keep multiple memory based PDF files open?

It is also possible to hold more than one memory based PDF file simultaneously in memory. This is the case if the flag **if2UseProxy** is set and if CloseImportFile() was not called before opening the next PDF file (the flag can be set with SetImportFlags2()). The flag reduces also the memory usage and should normally be set by default.

Be careful with the flag *ptDontCopyBuf* when loading more than one PDF file into memory. Do not delete the buffer before all required parts of the PDF file were imported!

Since memory based PDF files have no file name, an existing instance can only be re-opened with ReOpenImportFile(). The function requires the file handle that OpenImportBuffer() returned. To reduce the memory usage, memory based PDF files should be closed as soon as possible. A specific file instance can be closed with CloseImportFileEx().

Remarks:

The flag ptForceRepair can be used to explicitly load a PDF file in repair mode. The usage of is described at OpenImportFile().

Return values:

If the function succeeds the return value is zero or the file handle if the flag if2UseProxy is set (a value greater or equal zero). If the function fails the return value is a negative error code. See also OpenImportFile().

OpenImportFile

```
SI32 pdfOpenImportFile(
    const PPDF* IPDF,      // Instance pointer
    const char* FileName,  // File path to PDF file
    TPwdType PwdType,      // Kind of password see below
    const char* Password)  // Password to decrypt the file

typedef enum
{
    ptOpen        = 0, // Open password
    ptOwner       = 1, // Owner password
    ptForceRepair = 2, // See description below
    ptDontCopyBuf = 4  // Not meaningful with OpenImportFile()
}TPwdType;
```

The function opens an external PDF file so that it can be imported entirely or parts of it. After a PDF file was opened, properties of it can be derived by several functions such as GetInDocInfo(), GetInPageCount() and so on. To import specific pages from the file use the functions ImportPage() or ImportPageEx(). The entire PDF file can be imported with the function ImportPDFFile() (see example below).

The parameter *PwdType* specifies the kind of password that is supplied to the function to decrypt a PDF file if necessary. If the file is not encrypted the parameter *Password* is ignored.

It is not required to close a PDF file explicitly before another one can be opened. However, if the file is no longer needed then it should be closed with CloseImportFile().

Recommended settings to split PDF files

To improve processing speed when splitting large PDF files into smaller pieces, one or more PDF files can be loaded permanent into memory. In this case the PDF files are not automatically closed when CloseFile() or FreePDF() is called. In order to load a PDF file permanent into memory set the property SetUseGlobalImpFiles() to true.

After the file or files were fully processed, set the property back to false before closing the last PDF file in memory or call FreePDF() so that all open import files can be released. For an example take a look into the description of SetUseGlobalImpFiles().

How to keep multiple PDF files open?

Since DynaPDF 3.0.26.69 it is possible to hold more than one PDF file open to improve the resource handling when pages from different PDF files cannot be imported in one pass.

A parser instance contains a duplicate array to determine whether a PDF object must be imported or not. When the parser will be deleted, e.g. when CloseImportFile() was called, then there is no more a way to determine whether an object was already imported if the same PDF file will be opened again to import additional pages or objects of it. This yields often to double resources since already imported objects can no longer be shared.

To avoid this issue, it is possible to hold more than one parser instance in memory. This is the case if the flag **if2UseProxy** is set and if CloseImportFile() was not called before opening the next PDF file (the flag can be set with SetImportFlags2()). The flag reduces also the memory usage and should normally be set by default.

The function returns the file or parser handle in this case, a value greater or equal zero. This handle can be used to close or re-open a specific parser instance with CloseImportFileEx() or ReOpenImportFile().

CloseImportFile() closes the last open import file as usual and deletes the corresponding parser instance.

A parser instance requires about 280 KB memory depending on the number of objects in it. The memory usage grows a little bit when pages will be imported. Therefore, it is recommended to close instances when no longer needed to reduce the memory usage.

PDF files with only one page should be closed immediatly after the contents was imported.

OpenImportFile() checks whether an existing instance can be used before it creates a new one. Therefore, it is safe to open the same PDF file arbitrary often. More efficient to re-open existing parser instances is to call ReOpenImportFile() with the file handle that OpenImportFile() returned.

DynaPDF makes sure that no more than 6 file handles are opened simultaneously during import. Maximal 12 files are kept opened simultaneously if SetUseGlobalImpFiles() was set to true. This ensures that mutliple PDF files can be efficiently imported and written to disk without using too many system resources. The overhead to re-open additional files is minimal.

Editing encrypted PDF files

If your application should allow the modification of encrypted PDF files, you may check the access permissions to grant user rights, if the file was opened with the *open password* instead of the *owner password* (see also GetUserRights()).

Due to the license agreement of Adobe, all manufacturers of applications which make the treatment of encrypted PDF files possible, must respect the access permissions of a PDF file, if the file was opened with the *open password*.

Only if the file was opened with the *owner password*, all rights should be granted. See *PDF Reference 1.7* for further information. This document is available at http://www.adobe.com.

If the property *UseExactPwd* is set to false (see SetUseExactPwd()), the function checks whether the open or owner password in the file is an empty string. If one password is not set, then the file is decrypted no matter whether the supplied password was wrong.

However, the property *UseExactPwd* should be **true**, if the application is a commercial product (default).

Use the function IsWrongPwd() to determine whether the function failed due to a wrong password. The function requires the return value of this function to determine whether the password was wrong. The function can be executed in a loop so that the user is able to enter another password.

Damaged PDF files

The function reads the file header, the cross-reference tables and the required global objects when opening a PDF file. It checks also if the first page object is available. When it is possible to load these objects without errors then the file is loaded in normal mode by using the cross-reference table to fetch objects. If an error occurs during loading the global objects then the function tries to repair the file by scanning the all the objects in the file to rebuild the cross-reference table.

However, PDF files can contain damages in the cross-reference table which cause not necessarily an error when opening the file, e.g. when the damages affect only specific pages in the file. In such cases it is possible to load a file explicitly in repair mode. This can be achieved with the flag ptForceRepair. The flag must be combined with a binary or operator with the parameter PwdType, e.g. (TPwdType)(ptOpen | ptForceRepair).

The ptForceRepair flag should only be set if it was not possible to load specific pages due to errors in the file. Check first whether the file was not already loaded in repair mode with GetInRepairMode(). If the function returns false then load the file in repair mode and try to import it again. If the file contains no fatal errors then it is often possible to repair the damages.

The repair mode is supported for PDF files with an uncompressed file structure only because PDF files with a compressed object structure are organized in a manner that does not allow further repair actions.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Unicode paths are converted to UTF-8 on non-Windows operating systems.

Return values:

If the function succeeds the return value is zero or the file handle if the flag if2UseProxy is set (a value greater or equal zero). If the function fails the return value is a negative error code. This code can be used to check with IsWrongPwd() whether the supplied password was wrong.

Example (C):

This example loads a PDF file and encrypts it. After a PDF file was loaded it is also possible to edit specific pages of it. Existing pages can be opened with EditPage()…

```
// First, we declare an error callback function so that we can see all
// errors or warnings.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
```

```cpp
   return 0;
}
int main(int argc, char* argv[])
{
   char outFile[] = "cout.pdf";
   void* pdf = pdfNewPDF(); // Create a new PDF instance
   if (!pdf) return -2; // Out of memory?
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   // The output file is opened later...
   pdfCreateNewPDF(pdf, NULL);
   // No need to convert pages to templates. See SetImportFlags() for
   // further information.
   pdfSetImportFlags(pdf, ifImportAll | ifImportAsPage);
   pdfSetImportFlags2(pdf, if2UseProxy); // Reduce the memory usage
   pdfOpenImportFile(pdf, "test.pdf", ptOwner, NULL);
   pdfImportPDFFile(pdf, 1, 1.0, 1.0);
   pdfCloseImportFile(pdf);
   // No fatal error occurred?
   if (pdfHaveOpenDoc(pdf))
   {
      // OK, now we can open the output file. The function can be called
      // in a while statement, e.g. to display a file open dialog.
      if (!pdfOpenOutputFile(pdf, outFile))
      {
         pdfDeletePDF(pdf);
         _getch();
         return -1;
      }
      // We encrypt the file, low-resolution printing should be allowed
      if (pdfCloseFileEx(pdf,NULL,"%3Fc&",kl128bit, rsDenyAll & ~rsPrint))
      {
         printf("PDF file \"%s\" successfully created!\n", outFile);
         ShellExecute(0, "open", outFile, NULL, NULL, SW_SHOWMAXIMIZED);
      }
   }
   pdfDeletePDF(pdf); // Do not forget to delete the PDF instance
   return 0;
}
```

OpenOutputFile

Syntax:

```
LBOOL pdfOpenOutputFile(
    const PPDF* IPDF,   // Instance pointer
    const char* OutPDF) // Output file name
```

The function opens the output file into which the PDF file should be written. The PDF file must be created in memory if this function should be used. This is the case if the output file name is set to NULL or to an empty string in CreateNewPDF().

After the output file was opened the PDF file is no longer created in memory. So, there is no need to call FreePDF() after the file was finished with CloseFile() or CloseFileEx().

OpenOutputFile() can be called in a while statement, e.g. to display a open file dialog if the file could not be opened. Once the function succeeds the PDF file can be finished with CloseFile().

It is strongly recommended to check with HaveOpenDoc() whether a PDF file is still in memory before calling this function.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Unicode paths are converted to UTF-8 on non-Windows operating systems.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

OpenOutputFileEncrypted

Syntax:

```
LBOOL pdfOpenOutputFileEncrypted(
    const PPDF* IPDF,         // Instance pointer
    const char* OutPDF,       // The output file that should be opened
    const char* OpenPwd,      // Optional open password
    const char* OwnerPwd,     // Optional owner password (should be set)
    TKeyLen KeyLen,           // Encryption key length
    TRestrictions Restrict)   // Restriction flags (see below)

typedef enum
{
    kl40bit     = 0, // RC4 Encryption -> PDF 1.2, Acrobat 3 or higher
    kl128bit    = 1, // RC4 Encryption -> PDF 1.4, Acrobat 5 or higher
    kl128bitEx  = 2, // RC4 Encryption -> PDF 1.5, Acrobat 6 or higher
    klAES128    = 3, // AES Encryption -> PDF 1.6, Acrobat 7 or higher
    klAES256    = 4, // AES Encryption -> PDF 1.7, Acrobat 9 or higher
    klAESRev6   = 5  // AES Encryption -> PDF 2.0, Acrobat X or higher
}TKeyLen;

typedef SI32 TRestrictions;
#define rsDenyNothing      0x00000000 // Encrypt the file only
#define rsDenyAll          0x00000F3C // Deny anything
#define rsPrint            0x00000004 // Deny printing
#define rsModify           0x00000008 // Deny modification of contents
#define rsCopyObj          0x00000010 // Deny copying of contents
#define rsAddObj           0x00000020 // No commenting
/* 128/256 bit encryption only -> ignored if 40 bit encryption is used */
#define rsFillInFormFields 0x00000100 // requires rsModify + rsAddObj
#define rsExtractObj       0x00000200 // requires rsModify
#define rsAssemble         0x00000400 // requires rsModify
#define rsPrintHighRes     0x00000800 // Disable high res. printing
#define rsExlMetadata      0x00001000 // PDF 1.5 Exclude metadata streams
#define rsEmbFilesOnly     0x00002000 // PDF 1.6 AES Encryption only
```

The function opens the output file and sets the encryption parameters. This function enables in combination with FlushPages() the creation of very large encrypted PDF files with minimal memory usage.

The function can be called in a while statement, e.g. to display a open file dialog if the file could not be opened. Once the function succeeds the PDF file can be finished with CloseFile().

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Unicode paths are converted to UTF-8 on non-Windows operating systems.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

OpenTag

Syntax:

```
LBOOL pdfOpenTag(
  const PPDF* IPDF,        // Instance pointer
  TPDFBaseTag Tag,         // The tag that should be created
  const char* Lang,        // Optional RFC 3066 laguage identifier
  const char* AltText,     // Optional alternate text
  const char* Expansion)   // Optional expansion of abbrevations or
                           // acronyms

typedef enum
{
    btArt,
    btArtifact,       // Artifact (contents that should be ignored)
    btAnnot,          // An annotation must be inserted to finish the tag!
    btBibEntry,       // Bibliography entry
    btBlockQuote,
    btCaption,
    btCode,
    btDiv,
    btDocument,
    btFigure,
    btForm,           // A form field must be inserted to finish the tag!
    btFormula,
    btH,
    btH1,
    btH2,
    btH3,
    btH4,
    btH5,
    btH6,
    btIndex,
    btLink,           // A link annotation must be inserted to finish the tag (FileLink(), PageLink(),
                      // or WebLink()!
    btList,           // L
    btListElem,       // LI
    btListText,       // LBody
    btNote,
    btP,
    btPart,
    btQuote,
    btReference,
    btSection,        // Sect
    btSpan,
    btTable,
    btTableDataCell,  // TD
    btTableHeader,    // TH
    btTableRow,       // TR
    btTOC,
    btTOCEntry,       // TOCI
    btCL,             // CL (Continued List)
    btLabel,          // Lbl
    btRuby,
    btWarichu,
    // PDF 2.0 tags:
    btAbbr,
    btCallout,
    btDek,
    btEm,
    btLineNum,
    btReason,
    btRedaction,
    btSidebar,
```

```
    btStrong,
    btSub,
    btSup
}TPDFBaseTag;
```

The function opens the specified tag so that contents can be written into it. When the corresponding contents was drawn or output the tag must be closed with CloseTag(). The supported tags have the same meaning as the corresponding HTML tags.

Many tags support additional attributes which cannot be set by this function. If additional attributes should be set, use OpenTagEx() instead.

Notice:

Tagged PDF files require a global structure tree that must be created with CreateStructureTree() before this function can be called.

DynaPDF is able to extend an existing structure tree that was imported from an external PDF file. However, when editing external PDF files the following rules must be considered:

- The structure information of a PDF file will only be imported when the entire PDF file is imported with ImportPDFFile(). When importing single pages with ImportPage() or ImportPageEx() of a PDF file that contains structure information then tagging will be disabled because DynaPDF is not able to import the structure information of a PDF file on a per page basis.

- Import **first** the PDF file and call then CreateStructureTree(). The reverse order causes that the structure tree will not be imported and tagging will be disabled!

- Only one PDF file with structure information can be imported without invalidating the structure information.

- Keep in mind that the structure tree of a PDF file is a complex global structure that is difficult to edit. Because of this, probably millions of PDF files exist that contain damages in the structure tree, mostly due to editing actions in certain viewer applications. When opening an existing page with EditPage() the function tries to find the corresonding StructParents array of the page in the ParentTree of the document's Structure Tree. When this action fails then tagging will be disabled for this page. OpenTag() and CloseTag() do not produce further warnings in this case.

- Note that some tags require a bounding box, e.g. *btFigure* or *btTable*. Use OpenTagBBox() to open such a tag.

The parameters *Lang*, *AltText*, and *Expansion* are all optional. The *Lang* parameter, if set, must be a valid language identifier as defined in RFC 3066, *Tags for the Identification of Languages*. It overrides the global language identifer that can be set with SetLanguage(). The main language of the document should always be set with SetLanguage() when creating Tagged PDF files.

Note that several structure elements like *btListElem* or *btTableHeader* cannot be used stand alone. That means the corresponding parent tag must be opened before such a tag can be used. For example, the tag *btListElem* is only valid as child of a btList element and *btListElemText* is in turn only valid when used as a child of a *btListElem* tag!

Example:

```
...
pdfOpenTag(PDF, btList, NULL, NULL, NULL);                    // OK
    pdfOpenTag(PDF, btListElem, NULL, NULL, NULL);           // OK
        pdfOpenTag(PDF, btListElemText, NULL, NULL, NULL); // OK
            pdfWriteText(PDF, 50.0, 50.0, "Some list text!");
        pdfCloseTag(PDF);
    pdfCloseTag(PDF);
pdfCloseTag(PDF);

pdfOpenTag(PDF, btList, NULL, NULL, NULL);                    // OK
    pdfOpenTag(PDF, btListElemText, NULL, NULL, NULL);      // Wrong
        pdfOpenTag(PDF, btListElem, NULL, NULL, NULL);      // Wrong
            pdfWriteText(PDF, 50.0, 50.0, " Some list text!");
        pdfCloseTag(PDF);
    pdfCloseTag(PDF);
pdfCloseTag(PDF);

pdfOpenTag(PDF, btListElem, NULL, NULL, NULL);               // Wrong
    pdfWriteText(PDF, 50.0, 50.0, "List Element!");
pdfCloseTag(PDF);
```

The tags *btAnnot*, *btForm*, and *btLink* can be used to add annotations and form fields to the structure. Note that the corresponding annotation or form field is required! When opening such a tag without inserting the corresponding annotation or field before the tag is closed then the structure will be damaged!

Remarks:

The maximum allowed nesting level of marked content operators is 64, including optional content operators (Layers). At time of publication DynaPDF does not check whether tags are used in the correct hierarchy.

However, when creating tagged PDF files you should always validate the tagging information with the accessibility check of Adobe's Acrobat (Menu Advanced/Accessibility/Full Check...).

Tagged PDF files should be created in the logical reader.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is zero.

OpenTagBBox

Syntax:

```
LBOOL pdfOpenTagBBox(
    const PPDF* IPDF,        // Instance pointer
    TPDFBaseTag Tag,         // The tag that should be created
    const char* Lang,        // Optional RFC 3066 laguage identifier
    const char* AltText,     // Optional alternate text
    const char* Expansion,   // Optional expansion of abbrevations or
                             // acronyms
    struct TBBox* BBox)      // Optional bounding box
```

The function opens a tag in the very same way as OpenTag() but supports the additional parameter *BBox* that is required by the tags *btFigure* and *btTable*. Since *BBox* is optional the function can be used to create all tags. Set *BBox* to NULL if it is not required. See OpenTag() for further information.

BBox must be defined in the current coordinate system that is used to output the corresponding contents. All transformations will be taken into account.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is zero.

OpenTagEx

Syntax:

```
LBOOL pdfOpenTagEx(
    const PPDF* IPDF,
    TPDFBaseTag Tag,            // The tag that should be created
    const char* Lang,           // Optional RFC 3066 laguage identifier
    const char* AltText,        // Optional alternate text
    const char* Expansion,      // Optional expansion of abbrevations or
                                // acronyms
    const char* Attributes)     // Optional. See description below.
```

The function opens a tag in the very same way as OpenTag() but supports additional parameters which can be set in a JSON (Javascript Object Notation) like format.

The optional parameter *Attributes* is a JSON like string. The JSON parser in DynaPDF supports a few extensions to make the definition of additional attributes as easy as possible. See JSON Parser for further information.

Examples (C/C++):

```
const char attr[] = "{/BBox[50, 450, 200, 500]/Placement/Inline}";   // Ok
const char attr[] = "/BBox :[50 450 200 500]/Placement/Inline";      // Ok
const char attr[] = "{{{{/BBox :[50 450 200 500]/Placement/Inline";  // Ok
const char attr[] = "/BBox :[50 450 200 500]/Placement/Inline}}}}";  // Ok
const char attr[] = "\"BBox\":[50 450 200 500]\"Placement\"/Inline"; // Ok
const char attr[] = "/BBox,[50 450 200 500]\"Placement\"/Inline";    // Ok
const char attr[] = "/BBox:[50 450 200 500]/Placement/Inline";       // Ok
```

The above examples produce all exactly the same output:

```
/A<</BBox[50 450 200 500]/Placement/Inline/O/Layout>>
```

Note that the /O key is added by DynaPDF if not already present. The missing right or left braces in the third and fourth example cause no error or warning. As described earlier, outer braces are ignored.

This example produces incorrect output:

```
const char attr[] = "\"BBox\":[50 450 200 500]\"Placement\"\"Inline\""; // Wrong!
```

Result:

```
/A<</BBox[50 450 200 500]/Placement(Inline)/O/Layout>>
```

The problem is that the value of the */Placement* key must be a name object. The JSON parser knows that a PDF key is always a name object. Therefore, the function converts a key defined as string automatically to a name object. However, the value of a key can be a name, string, boolean, array, dictionary, null, or a number. The PDF Standard specifies which data type a key accepts. In this particular case, the */Placement* key requires a name object as value. The function does not know whether the value of an arbitrary key is valid.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is zero.

Optimize

Syntax:

```
LBOOL pdfOptimize(
    const PPDF* IPDF,              // Instance pointer
    TOptimizeFlags Flags,         // See below
    struct TOptimizeParams* Parms) // Can be NULL, see desciption below
typedef enum
{
    ofDefault            = 0x00000000,// Just rebuild the content streams.
    ofInMemory           = 0x00000001,// Optimize the file fully in memory.
    ofConvertAllColors   = 0x00000002,// If set, Separation, DeviceN, and NChannel color spaces
                                      // will be converted to the device space.
    ofIgnoreICCBased     = 0x00000004,// If set, ICCBased color spaces will be left unchanged.
    ofScaleImages        = 0x00000008,// Scale images.
    ofSkipMaskedImages   = 0x00000010,// Meaningful only if ofScaleImages is set. If set, don't
                                      // scale  images with a color mask.
    ofNewLinkNames       = 0x00000020,// If set, rename all object links to short names.
    ofDeleteInvPaths     = 0x00000040,// Delete invisible paths. An invisible path is a path that
                                      // was finished with the no-op operator "n".
    ofFlattenLayers      = 0x00000080,// Flatten layers if any.
    ofDeletePrivateData  = 0x00000100,// Delete private data from pages, templates, and images.
    ofDeleteThumbnails   = 0x00000200,// No need to embed thumbnails (can be created on demand).
    ofDeleteAlternateImages = 0x00000400,// If set, alternate images will be deleted if any.
    ofNoImageSizeCheck   = 0x00000800,// Meaningful only if ofScaleImages is set. If set, do not
                                      // check whether the scaled image is smaller.
    ofIgnoreZeroLineWidth = 0x00001000,// Meaningful only if the parameter MinLineWidth of the
                                      // TOptimizeParams structure is greater zero.
                                      // If set, ignore line width operators with a value of zero
                                      // (zero means one device unit).
```

```
    ofAdjZeroLineWidthOnly  = 0x00002000,// Meaningful only if the parameter MinLineWidth of the
                                         // TOptimizeParams structure is greater zero. If set, change
                                         // the line width of real hairlines only (a hairline is a
                                         // one pixel width line -> LineWidth == 0).
    ofCompressWithJBIG2     = 0x00004000,// If set, 1 bit images are compressed with JBIG2 if not
                                         // already compressed with this filter.
    ofNoFilterCheck         = 0x00008000,// Meaningful only, if the flag ofCompressWithJBIG2 is set.
                                         // If set, re-compress all 1 bit images, also if already
                                         // compressed with JBIG2.
    ofConvertGrayTo1Bit     = 0x00010000 // Useful for scanned faxes since many scanners create gray
                                         // images for black & white input.
   /* ------------------------- Notice: ----------------------------------------------
    * Special color spaces like Separation, DeviceN, and NChannel are ignored unless the flag
      ofConvertAllColors is set too.
    */
    ofConvertToGray         = 0x00020000, // Convert images, text, and vector graphics to DeviceGray.
    ofConvertToRGB          = 0x00040000, // Convert images, text, and vector graphics to DeviceRGB.
    ofConvertToCMYK         = 0x00080000, // Convert images, text, and vector graphics to DeviceCMYK.
    ocReplaceJP2KWithJPEG   = 0x00100000  // Meaningful only, if the flag ofConvertToGray,
                                          // ofConvertToRGB, or ofConvertToCMYK is set. If set, JPEG
                                          // 2000 compressed images (which are converted to another
                                          // color space) will be compressed with JPEG instead of
                                          // JPEG 2000. JPEG 2000 compression is very slow and
                                          // requires much memory. JPEG compression is around 10
                                          // times faster and produces almost identical results.
    ofUseOtsuFilter         = 0x00200000  // Meaningful only, if the flag ofConvertGrayTo1Bit is set.
                                          // The Otsu filter is useful for gray scanned faxes.
    ofConvTextToOutlines    = 0x00400000, // If set, texts are converted to outlines or vector
                                          // graphics. The resulting file contains no fonts with
                                          // exception of Type3 fonts, if any.
    ofConvNonEmbFontsOnly   = 0x00800000  // Meaningful only, if the flag ofConvTextToOutlines is
                                          // set. If set, texts of non-embedded fonts are converted
                                          // to outlines. Embedded fonts remain embedded.
    ofIgnoreDeviceN         = 0x01000000, // Meaningful only if a color conversion flag is set. If
                                          // set, DeviceN color spaces are left unchanged.
    ofIgnoreNChannel        = 0x02000000, // If set, NChannel color spaces are left unchanged.
    ofIgnoreSeparation      = 0x04000000, // If set, Separation color spaces are left unchanged.
    ofFailOnOverprinting    = 0x08000000, // Meaningful only if a color conversion flag is set. If
                                          // set, the function returns with a fatal error if an
                                          // object with enabledoverprinting was found on the page.
                                          // The appearance of overprinted objects would change and
                                          // make the page maybe unusable.
                                          // Overprinted objects can be left unchanged (set the flag
                                          // ofSkipOverprintedObj in this case) or Optimize() can
                                          // return with a fatal error to avoid the creation of an
                                          // invalid document.
    ofSkipOverprintedObj    = 0x10000000  // Meaningful only if a color conversion flag is set. If
                                          // set, objects that would be rendered with enabled
                                          // overprinting are left unchanged to avoid color errors on
                                          // the page.
    ofRemoveBatesNumbers    = 0x20000000, // Remove bates numbers if any. The type BatesN can contain
                                          // headers / footers too.
    ofRemoveHeaderFooter    = 0x40000000, // Remove headers / footers if any. This type is set if no
                                          // bates numbering was added. To determine whether a file
                                          // contains headers or footers call
                                          // LoadHeaderFooterSettings().
    ofIgnoreSeparationAll   = 0x80000000  // Meaningful only if a color conversion flag is set. If
                                          // set, Separation color spaces with the special colorant
                                          // "All" will be ignored.
}TOptimizeFlags;

typedef enum
{
    of2Default              = 0x00000000, // Nothing to do
    of2DeleteWatermarks     = 0x00000001, // Delete watermark templates
    of2InclWatermarkAnnots  = 0x00000002  // If set, watermark annotations will be deleted.
}TOptimizeFlags2;

// Placeholder defined for future use
```

```
typedef enum
{
    of3Default             = 0x00000000  // Nothing to do
}TOptimizeFlags3;

typedef enum
{
    of4Default             = 0x00000000  // Nothing to do
}TOptimizeFlags4;

struct TOptimizeParams
{
    UI32               StructSize;     // Must be set to sizeof(TOptimizeParams).

    UI32               Min1BitRes;     // Minimum resolution before scaling.
    UI32               MinGrayRes;     // Minimum resolution before scaling.
    UI32               MinColorRes;    // Minimum resolution before scaling.

    UI32               Res1BitImages;  // 1 bit black & white images.
    UI32               ResGrayImages;  // Gray images.
    UI32               ResColorImages; // Multi-channel images.

    TCompressionFilter Filter1Bit;     // Filter for black & white images.
    TCompressionFilter FilterGray;     // Filter for gray images.
    TCompressionFilter FilterColor;    // Filter for multi-channel images.
    SI32               JPEGQuality;    // JPEG quality.
    SI32               JP2KQuality;    // JPEG 2000 quality.
    float              MinLineWidth;   // Zero means no hair line removal.
    const UI32*        ExcludeCS;      // Array of color space handles which should be excluded from
                                       // color conversion.
    SI32               ExcludeCSCount; // Number of handles in the array. This can be a negative
                                       // value to reverse the meaning. That means convert these
                                       // color spaces and ignore all others.
    TOptimizeFlags2    Flags2;         // Additional flags
    TOptimizeFlags3    Flags3;         // Additional flags
    TOptimizeFlags4    Flags4;         // Additional flags
};
```

The function rebuilds the content streams of all pages, templates, patterns, annotations, and form fields. Useless operators as well as errors in content streams will be fully removed. The resulting content streams are error free and usually smaller. How much the optimization takes effect depends on the quality of the original content streams.

Besides content optimization it is also possible to flatten layers, scale or just re-compress images, delete watermarks, or to correct hairlines.

The parameter *Flags* is a bit mask. Multiple flags can be combined with a binary or operator, e.g. ofInMemory | ofScaleImage.

The flag ofScaleImages will be ignored if the parameter *Parms* is set to NULL.

Optimize() does **not** rebuild font subsets like CheckConformance(). The function is part of DynaPDF Professional.

Usage

It is assumed that this function is called right before closing the PDF file. It is not allowed to call the function twice for the same PDF file in memory.

After Optimize() was executed, close the file with CloseFile(), CloseFileEx(), or CloseAndSignFile() for example.

If the flag *ofInMemory* is absent, then pages are directly written to the output file after optimization to reduce the memory usage. If the PDF file was created in memory (this is the case if no ouput file name was passed to CreateNewPDF()) then the output file must be opened with OpenOutputFile() or OpenOutputFileEncrypted() beforehand. Note that it is not possible to sign the PDF file if the flag *ofInMemory* is absent.

If the file should be signed too, then import the resulting PDF file again and sign it with CloseAndSignFile() or CloseAndSignFileEx() for example.

Re-compressing 1 bit images with JBIG2

If the flag *ofCompressWithJBIG2* is set, 1 bit images will be re-compressed with JBIG2 (if the image was not already compressed with this filter). This can drastically reduce the file size since JBIG2 compression achieves much higher compression rates than any other 1 bit image filter that PDF supports. The JBIG2 compression filter in DynaPDF is lossless, that means the original image quality will be preserved.

Text to outline conversion

The function converts text to outlines if the flag *ofConvTextToOutlines* is set. The flattening algorithm loads exactly the same fonts as the rendering engine would load and it scales fonts in the very same way. What you see is what you get.

Outlines are scaled to a master size of 1000 units to achieve high quality output. It stores hinted outlines with preserved curves. The resulting PDF files are much smaller in comparison to fully flattened outlines and offer higher quality since curves stay smooth regardless of the zoom factor.

It is also possible to restrict the conversion to non-embedded fonts with the flag *ofConvNonEmbFontsOnly* since these fonts are usually the problematic ones when printing on Postscript devices.

Annotations and form fields are not affected by the conversion. In order to consider these objects too it is possible to flatten all annotations and form fields before calling Optimize(). See FlattenAnnots() for further information.

Color conversion

The function converts images, text, and vector graphics to a device color space if one of the flags *ofConvertToGray*, *ofConvertToRGB*, or *ofConvertToCMYK* is set.

If color management is enabled (see InitColorManagement() for further information), make sure that the destination color space is set to the same output color space as specified in the Optimize() function call.

The optional array *ExcludeCS* can be used to define an array of color space handles which should be excluded from color conversion.

ExcludeCSCount can be set to a negative number to reverse the meaning. That means convert these color spaces and ignore all others.

Color conversion rules

The function does never convert gray objects into a higher color space. Image data is left unchanged if the image uses an indexed color space and if scaling is not required. Only the indexed color space is converted in this case.

Interactive objects like annotations and form fields are not converted to the destination color space unless these objects would be flattened beforehand. Annotations and form fields can be flattened with FlattenAnnots().

Overprinting

Objects that use overprinting can mostly not be converted to another color space without changing the appearance of a page. To avoid issues with overprinted objects, the function can ignore such objects, return with a fatal error, or convert them like any other object.

If overprinted objects should be excluded from color conversion, set the flag *ofSkipOverprintedObj*. If the function should return with an error instead, set the flag *ofFailOnOverprinting*. If both flags are absent then overprinted objects are converted to the specified color space.

Recommended compression filters

If PDF files are converterted to CMYK, e.g. for printing purposes, it is usually best to use loss-less compression filters for all image types, e.g. JBIG2 for 1 bit images, and Flate for anything else. The compression filters can be specified in the *TOptimizeParams* structure.

If no *TOptimizeParams* structure is provided, then images will be compressed with the same filter that the original image used. The quality of JPEG and JPEG 2000 compressed images is taken from the global property (see SetJPEGQuality()) in this case. Note that recompressing JPEG or JPEG 2000 images causes loss of quality.

Special compression filter flags:

The compression filter members of the structure *TOptimizeParms* can be combined with the flags *cfPresLosslessFilter* and *cfPresLossyFilter* to preserve the original compression filter. If both flags are set, the image is compressed with the original compression filter.

The flags can be combined with a compression filter constant with a binary **or** operator:

Example (C/C++):

```
p.FilterGray = (TCompressionFilter)(cfFlate | cfPresLossyFilter);
```

If no compression filter constant is set, cfFlate is used as alternate filter.

Image size check

The flag *ofNoImageSizeCheck* is implicitely set if a color conversion flag is set. This makes sure that images can be converted to another color space also if the new image becomes larger. This is usually always the case when converting images to CMYK from a lower color space.

Converting gray images to 1 bit (black & white)

Gray images will be converted to 1 bit b&w if the flag *ofConvertGrayTo1Bit* is set. The flag is considered only if the flag *ofCompressWithJBIG2* is set or if the parameter *Parms* and the flag *ofScaleImages* is set. The conversion of gray images to b&w is mainly useful for scanned faxes since many scanners create gray images for b&w input.

Image scaling

If the flag *ofScaleImages* is set, the function scales imported images according to the settings in the *TOptimizeParams* structure. The function distuingishes between 1 bit, gray, and color images. The minimum resolution of the original image that must be reached before scaling is applied can be set for these three image types independently.

The minimum resolution of 1 bit images should be around 50% higher than the destination resolution because scaling of 1 bit images can cause significant loss of quality if the original resolution is near the destination resolution.

The minimum resolution of gray and color images can be freely chosen. 1 means scale all images. The function scales images only if the minimum resolution is greater zero.

Spot color spaces

Images wich use a spot color space like Separation, DeviceN and NChannel are scaled only if the flag *ofConvertAllColors* ist set.

Image size check

The function checks whether the new image is smaller as the original one (except if a color conversion flag is set). If no size reduction can be achieved then the image is stored unchanged. The size check can be disabled with the flag *ofNoImageSizeCheck*.

Images which were inserted with DynaPDF functions like InsertImage() will be ignored.

Masked images (color key masking)

Color key masking is a technique to make parts of an image transparent. This type of masking depends on exact color values because a specific color is used to mask the transparent areas. If such an image will be scaled, it is possible that the color of certain pixels will be changed due to color interpolation and this can lead to visible differences which are maybe not acceptable.

To avoid issues with masked images it is recommended to set the flag *ofSkipMaskedImages*.

Hairline correction

If the member *MinLineWidth* of the structure TOptimizeParams is greater zero the line width of stroked paths will be set to *MinLineWidth* if the scaled value was smaller. The function considers all transformations of the coordinate system.

The line width is measured in PDF units. One PDF unit represents 1/72 inch.

The minimum line width should be smaller or equal 1.0. The maximum value is 10.0 units.

Depending on the used flags it is possible to correct real hairlines only (zero width lines), all lines which are thinner than *MinLineWidth*, or all lines with exception of real hairlines (line width = 0).

Possible issues

Hairline correction does not always produce the expected result, e.g. lines which look like a stroked path are in fact filled rectangles. It is also possible that a surrounding clipping path prevents a visible change of the line width.

Special flags

The flag *ofDeletePrivateData* can be set to delete private data objects from pages, templates, and images. Private data objects are created by several applications. For example, Adobe's Photoshop or InDesign write the original images or PDF pages into private data structures (depending on the settings in these applications).

A normal PDF viewer does not understand this data and ignores it. Because such objects are often very large it is recommended to delete useless private data.

Recommendations

Optimize can do a lot but not anything. For example, the function does not perform a duplicate check for images, templates, and so on. Such a duplicate check is available when importing a PDF

file but not after the file was already imported. In addition, the memory usage must be restricted. Optimize() can flush pages away after optimization but it is also important to make sure that not too much memory will be consumed before the function is called.

Therefore, the right flags must be used when importing a PDF file. Here are the default import flags that should be used when optimizing PDF files:

```
pdfSetImportFlags(pdf, ifImportAll | ifImportAsPage);
pdfSetImportFlags2(pdf, if2UseProxy | if2DuplicateCheck);
```

The above flags avoid the conversion of pages to templates, the memory usage will be reduced, and a duplicate check is applied.

Many other flags are available but the above flags can be used by default whenever a PDF file should be optimized.

Error handling

This function adds warnings to the error log of DynaPDF. The error log can be accessed with GetErrLogMessageCount() / GetErrLogMessage().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0. No PDF file is in memory if the function fails. Therefore, it is not required to call FreePDF() on failure.

PageLink

Syntax:

```
SI32 pdfPageLink(
   const PPDF* IPDF, // Instance pointer
   double PosX,      // X-Coordinate of bounding rectangle
   double PosY,      // Y-Coordinate of bounding rectangle
   double Width,     // Width of bounding rectangle
   double Height,    // Height of bounding rectangle
   UI32 DestPage)    // Destination page
```

The function adds a page link to the current open page. The parameter *DestPage* specifies the destination page which should be opened (the page number). The function does not check whether the destination page exists; it can be created later. If the destination page does not exist when the document is closed it will be set to the first page.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the bounding rectangle. If the coordinate system is top-down it defines the upper left corner.

The border of the link annotation is drawn by using the current line width, stroke color and line dash pattern. If the link should appear without a border set the line width to zero beforehand.

When clicking on a link annotation the rectangle is highlighted, that is a simple visual effect. Several highlight modes are supported, see SetLinkHighlightMode() for further information.

Remarks:

The destination page is always opened at the top corner. More precise destinations can be created with the function PageLinkEx(). Page links can also be used to execute an action. The destination page is ignored in the latter case, the annotation works then like a button.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

PageLink2

Syntax:

```
SI32 pdfPageLink2(
   const PPDF* IPDF, // Instance pointer
   double PosX,      // X-Coordinate of bounding rectangle
   double PosY,      // Y-Coordinate of bounding rectangle
   double Width,     // Width of bounding rectangle
   double Height,    // Height of bounding rectangle
   UI32 NamedDest)   // Handle of a named destination
```

The function adds a page link to the current open page by using a named destination as target. Named destinations can be used if a destination should be accessible from another PDF file. The parameter *NamedDest* must be a valid handle of named destination. See CreateNamedDest() for further information.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the bounding rectangle. If the coordinate system is top-down it defines the upper left corner.

The border of the link annotation is drawn by using the current line width, stroke color and line dash pattern. If the link should appear without a border set the line width to zero beforehand.

When clicking on a link annotation the rectangle is highlighted, that is a simple visual effect. Several highlight modes are supported, see SetLinkHighlightMode() for further information.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

PageLink3

Syntax:

```
SI32 pdfPageLink3(
   const PPDF* IPDF,        // Instance pointer
   double PosX,             // X-Coordinate of bounding rectangle
   double PosY,             // Y-Coordinate of bounding rectangle
   double Width,            // Width of bounding rectangle
   double Height,           // Height of bounding rectangle
   const char* NamedDest);  // Name of a named destination
```

The function adds a page link to the current open page by using a named destination as target. The difference in comparison to PageLink2() is that the named destination can be defined as string.

This makes it possible to create the link, also if you don't have the required information to create the named destination at this point. If the named destination does not exist when the file is closed then the link does nothing. See also, CreateNamedDest().

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

PageLinkEx

Syntax:

```
SI32 pdfPageLinkEx(
   const PPDF* IPDF,     // Instance pointer
   double PosX,          // X-Coordinate of bounding rectangle
   double PosY,          // Y-Coordinate of bounding rectangle
   double Width,         // Width of bounding rectangle
   double Height,        // Height of bounding rectangle
   TDestType DestType,   // see below
   UI32 DestPage,        // Destination page
   double a,             // Various, depends on destination type
   double b,             // Various, depends on destination type
   double c,             // Various, depends on destination type
   double d)             // Various, depends on destination type

typedef enum
{
    dtXY_Zoom,    // Three parameters (a, b, c) -> (X, Y, Zoom)
    dtFit,        // No parameters
    dtFitH_Top,   // One parameter    (a)
    dtFitV_Left,  // One parameter    (a)
    dtFit_Rect,   // Four parameters  (left, bottom, right, top)
    dtFitB,       // No parameters
    dtFitBH_Top,  // One parameter    (a)
    dtFitBV_Left  // One parameter    (a)
}TDestType;
```

The function adds a page link to the current open page. The parameter *DestPage* specifies the destination page which should be opened (the page number). The function does not check whether the destination page exists; it can be created later. If the destination page is not created before the document is closed, it will be set to the first page of the document.

If the coordinate system is bottom-up the point *PosX, PosY* defines the lower left corner of the bounding rectangle. If the coordinate system is top-down it defines the upper left corner.

The border of the link annotation is drawn by using the current line width, stroke color and line dash pattern. If the link should appear without a border set the line width to zero beforehand.

When clicking on a link annotation the rectangle is highlighted, that is a simple visual effect. Several highlight modes are supported, see SetLinkHighlightMode() for further information.

The destination created by this annotation allows jumping to a specific position in a document instead of simply opening the page such as PageLink() does.

Destination type	Description

dtXY_Zoom	Display the page designated by page with the coordinates (left top) positioned at the top-left corner of the window and the contents of the page magnified by the factor zoom. A zero value for any of the parameters left top or zoom specifies that the current value of that parameter is to be retained unchanged.
	
	Example:
	`// The zoom factor is left unchanged`
	`pdfPageLinkEx(pdf,150,50,150,20,2,dtXY_Zoom,50,750,0,0);`

dtFit	Display the page designated by page with its contents magnified just enough to fit the entire page within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the page within the window in the other dimension. This destination type has no parameters.

dtFitH_Top	Display the page designated by page with the vertical coordinate top positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of the page within the window.
	
	Example:
	`// The parameter a specifies the top coordinate`
	`pdfPageLinkEx(pdf,150,50,150,20,2,dtFitH_Top,750,0,0,0);`

dtFitV_Left	Display the page designated by page with the horizontal coordinate left positioned at the left edge of the window and the contents of the page magnified just enough to fit the entire height of the page within the window.
	
	Example:
	`// The parameter a specifies the left edge`
	`pdfPageLinkEx(pdf,150,50,150,20,2,dtFitV_Left,50,0,0,0);`

dtFitRect	Display the page designated by page with its contents magnified just enough to fit the rectangle specified by the coordinates left bottom right and top entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the rectangle within the window in the other dimension.
	
	Example:
	
	`pdfPageLinkEx(pdf,150,50,150,20,2,dtFit_Rect,150,550,450,`
	`700);`

The destination types *dtFitB, dtFitBH_Top and dtFitBV_Left* use the media box of the page to fit the page into the window. All other destination types use the crop box if any.

The destination types are the same as for a go-to actions and the function creates in fact a go-to action which is executed by the link annotation. However, the action is stored in a more compact format and cannot be shared with other objects.

If a destination should be used with multiple objects such as bookmarks, create a go-to action instead and add it to a normal page link (see PageLink()). The same action can then be added to other objects.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

ParseContent

Syntax:

```
LBOOL pdfParseContent(
    const PPDF* IPDF,                  // Instance pointer
    const void* Data,                  // User defined pointer
    struct TPDFParseInterface* Stack,  // Parser interface
    TParseFlags Flags)                 // See below

struct TPDFParseInterface
{
    TApplyPattern*       ApplyPattern;
    TBeginPattern*       BeginPattern;
    TBeginTemplate*      BeginTemplate;
    TBezierTo1*          BezierTo1;
    TBezierTo2*          BezierTo2;
    TBezierTo3*          BezierTo3;
    TClipPath*           ClipPath;
    TClosePath*          ClosePath;
    TDrawShading*        DrawShading;
    TEndPattern*         EndPattern;
    TEndTemplate*        EndTemplate;
    TLineTo*             LineTo;
    TMoveTo*             MoveTo;
    TMulMatrix*          MulMatrix;
    TRectangle*          Rectangle;
    TRestoreGraphicState* RestoreGraphicState;
    TSaveGraphicState*   SaveGraphicState;
    TSetCharSpacing*     SetCharSpacing;
    TSetExtGState*       SetExtGState;
    TSetFillColor*       SetFillColor;
    TSetFont*            SetFont;
    TSetLeading*         SetLeading;
    TSetLineCapStyle*    SetLineCapStyle;
    TSetLineDashPattern* SetLineDashPattern; // Obsolete -> Use SetLineDashPattern2 instead
    TSetLineDashPattern2* SetLineDashPattern2;
    TSetLineJoinStyle*   SetLineJoinStyle;
    TSetLineWidth*       SetLineWidth;
    TSetMiterLimit*      SetMiterLimit;
    TSetStrokeColor*     SetStrokeColor;
    TSetTextDrawMode*    SetTextDrawMode;
    TSetTextScale*       SetTextScale;
    TSetWordSpacing*     SetWordSpacing;
    void*                Reserved001;    // See comment below
    void*                Reserved002;    // See comment below
    TShowTextArrayW*     ShowTextArrayW; // Preferred for text extraction
    TInsertImage*        InsertImage;
    TShowTextArrayA*     ShowTextArrayA; // Preferred for text searching
    // Additional reserved members follow (must be set to NULL).
};

typedef UI32 TParseFlags;
#define pfNone             0x00000000 // Default
#define pfDecomprAllImages 0x00000002 // See description
#define pfNoJPXDecode      0x00000004 // See description
#define pfDitherImagesToBW 0x00000008 // Floyd-Steinberg dithering.
#define pfConvImagesToGray 0x00000010 // See description
#define pfConvImagesToRGB  0x00000020 // See description
#define pfConvImagesToCMYK 0x00000040 // See description
#define pfImageInfoOnly    0x00000080 // See description
// The following two flags are used internally by CheckConformance().
#define pfErrorCheckOnly   0x00000100 // If set, images are checked for decompression errors only.
#define pfFailOnError      0x80000000 // Fail on decompression errors.
```

The function parses the content stream of the current open page. The content parser can be used to extract text, images, and vector graphics from a PDF file. The parameter *Stack* holds a set of callback functions which are executed if corresponding operators were found in the content stream. The parameter *Data* is a user defined pointer that is passed unchanged to the callback functions.

All callback functions are optional. Which callback functions must be set depends on the kind of information that should be extracted. For example, an application that extracts images must at least provide the callback function TInsertImage.

All callback functions which return an integer value can break processing if necessary. A return value of zero indicates success and processing continues. A return value of 1 of the TBeginTemplate or TBeginPattern callback functions indicates that the object should be skipped. The corresponding content streams are not executed in this case. This can be useful when extracting images. Any other return value breaks processing.

Notice:

It is allowed to write arbitrary objects into the page while the content parser is executed but it is strongly required to check whether a fatal error occurred when writing something to the page. The callback function must return a negative value in such a case to break processing. This is required because the parser doesn't notice when a fatal error occurs. New objects will be ignored when parsing a page.

ParseContent() is already part of DynaPDF since version 2.0.30 but it was never documented. Because the function was undocumented, a few important changes were made in DynaPDF 2.5 which do not break backward compatibility. However, an application that uses the following features must be slightly changed when it is recompiled:

- The flag pfTranslateStrings is no longer defined because the used callback function specifies already whether strings should be converted to Unicode. The old constant 1 is ignored.

- Two additional callback functions TShowTextA and TShowTextW were defined at the reserved fields Reserved0001 and Reserved0002. These callback functions are no longer defined and no longer executed. These functions could be used in combination with TShowTextArrayA or TShowTextArrayW only. DynaPDF processes now the entire text with the array versions. So, existing applications will still work as expected.

The function supports several flags which are useful when extracting images from a PDF file. These flags are meaningful only, if the TInsertImage callback function is set.

Flag	Description
pfDecomprAllImages	If set, all images are decompressed. Decompressing all images can be useful if the images should be stored in a specific file format like TIFF, PNG, or BMP.
	If the flag is absent, images which are already stored in a valid file format are returned as is. This can be useful if no conversion to a specific image format is required. Images which can be stored unchanged are Gray and RGB JPEG images, as well as JPEG 2000 compressed images.
pfNoJPXDecode	Meaningful only if the flag pfDecomprAllImages is set. If set, JPEG 2000 images are not decompressed.
pfDitherImagesToBW	This flag is considered only if an image is decompressed. If set, decompressed images are converted to 1 bit black and white with the Floyd-Steinberg dithering algorithm. 1 bit images are returned unchanged.
pfConvImagesToGray	This flag is considered only if an image is decompressed. If set, decompressed images are converted to gray.
pfConvImagesToRGB	This flag is considered only if an image is decompressed. If set, decompressed images are converted RGB.
pfConvImagesToCMYK	This flag is considered only if an image is decompressed. If set, decompressed images are converted CMYK.
pfImageInfoOnly	If set, images are not decompressed. This flag can be useful if no further processing is required, e.g. if images should be enumerated.

Although the TDrawShading callback function is defined there is no way to process shadings at this time. Shadings can only be rastered to an image to get a meaningful result. However, such a feature is not available yet.

The content parser is not part of the Visual Basic interface due to the bad support for callback functions. Preferred programming languages are C/C++, C#, Delphi, or VB .Net. These programming languages offer optimal execution speed and the usage is relatively simple. Although .Net languages support no pointers the execution speed is almost identically to C/C++ or Delphi.

The Graphics State

An application that uses the content parser must maintain an internal graphics state stack that stores the current graphics state in a LIFO (last in first out) data structure. Depending on what kind of information should be extracted only a few members of the graphics state maybe required. For example, an application that extracts text from a PDF file can ignore path painting parameters like the current line cap style, line width and so on.

A complete graphics state contains the following variables:

Parameter	Type	Initial Value
CharSpacing	float	0.0f
Clipping Path	Vector array	Crop box or Media box
DashPattern	float*	NULL (Solid line)
DashPhase	UI32	0
FillColor	double[32]	Black
FillColorSpace	TExtColorSpace, IColorSpace*	esDeviceGray, NULL
FillPattern	IPattern*	NULL
Font	IFont*	NULL
FontSize	double	0.0
FontType	TFontType	-
Leading	float	0.0f (Info only, see below)
LineCapStyle	TLineCapStyle	csButtCap
LineJoinStyle	TLineJoinStyle	jsMiterJoin
LineWidth	float	1.0f
Matrix	TCTM	{1, 0, 0, 1, 0, 0}
MiterLimit	float	10.0f
StrokeColor	double[32]	Black
StrokeColorSpace	TExtColorSpace, IColorSpace*	esDeviceGray, NULL
StrokePattern	IPattern*	NULL
TextDrawMode	TDrawMode	dmNormal
TextScale	float	100.0f
WordSpacing	float	0.0f

The current graphics state must be initialized with the above default values before parsing a new page. Templates (Form XObjects in PDF terms) inherit the graphics state of the parent object that paints the template.

The parameter *Leading* is provided for information only because it is already considered in the text matrix of all text callback functions (see Coordinate Spaces for further information).

The variables which make up the graphics state are normally stored in a structure or class so that the graphics state can be saved and restored if corresponding operators are executed.

If the TSaveGraphicState callback function is executed, the application must create a copy of the current graphics state and push it onto the graphics state stack. In the corresponding TRestoreGraphicState callback function, the last element of the graphics state stack must be popped and copied to the current graphics state.

A correct handling of the graphics state is always required if the application needs to compute coordinates of graphic objects.

Note also that the graphics state can be saved and restored very often. The maximum allowed nesting level of save/restore graphics state operators in PDF is 28. The application should not depend on this limit but it can be used for optimizations, e.g. to create a graphics state cache.

Coordinate Spaces

The coordinates of a page extend from left to right and from bottom to top (subject to alteration by the orientation of the page (see GetOrientation())). Note that SetPageCoords() does not affect the coordinates returned by the content parser.

The length of a unit in default user space is **1/72** inch. The visible area of a page is defined by the crop box of the page, or if absent, by the media box (see GetBBox() for further information). The media box represents the coordinate system in which the page is defined.

In addition to user space, PDF uses a variety of other coordinate spaces for specialized purposes:

- Text coordinates are defined in text space. The transformation from text space to user space is defined by a text matrix which is provided in the text callback functions. PDF defines also several text related positioning parameters in the graphics state. However, to make the usage easier DynaPDF pre-computes the text transformation matrix so that no further calculations are required to transform the text space to user space. Note that the current leading, which specifies the distance between two text lines, is already considered in the text matrix. The text matrix is always provided. To transform text space to user space the text matrix must be multiplied with the one of the current graphics state.

- Images are defined in image space. An image in PDF is always one unit wide and one unit high in user space, regardless of the number of samples in the image. The mapping to the visible region in user space is achieved by temporarily altering the current transformation matrix. The image position, as well as the width and height must be computed from the current transformation matrix (take a look at the description of the TInsertImage callback function for further information).

- Templates (Form XObjects in PDF terms) are defined in form space. The transformation from form space to user space is defined by a form matrix that is provided in the TBeginTemplate callback function. However, the matrix parameter of the callback function is optional and can be set to NULL. Null means the matrix is set to the identity matrix. To transform form space to user space the form matrix must be multiplied with the one of the current graphics state.

- Patterns are defined in pattern space. The transformation from pattern space to the **initial** default user space of the object in which it is used is provided in the TBeginPattern callback function. However, the matrix parameter of the callback function is optional and can be set to NULL. Null means the matrix is set to the identity matrix. Note that patterns are treated like a color. The TBeginPattern callback function starts the definition of the pattern and activates it in the graphics state when the corresponding TEndPattern callback function is executed. However, a pattern is applied when the next fill or stroke path operation occurs and if the pattern was defined as fill or stroke color.

    o   If a pattern is used on a page, the pattern matrix maps pattern space to the default (initial) coordinate space of the page. Changes to the page's transformation matrix that occur within the page's content stream, such as rotation and scaling, have no

effect on the pattern; it maintains its original relationship to the page no matter where on the page it is used. Similarly, if a pattern is used within a template, the pattern matrix maps pattern space to the template's default user space (that is, the template coordinate space at the time the template is painted). A pattern may be used within another pattern; the inner pattern's matrix defines its relationship to the pattern space of the outer pattern.

Working with Transformation Matrices

Coordinate transformations in PDF are achieved with so called Affine Transformations. Affine transformations can represent any linear mapping from one coordinate system to another.

Affine transformations work with transformation matrices which consist of 6 real numbers. A transformation matrix describes the coordinate origin, scaling factors of the x- and y-axis, as well as the rotation angle of the coordinate system. A transformation matrix represents essentially a coordinate space in which graphical objects can be drawn. Two coordinate spaces can easily be concatenated by multiplying the corresponding transformation matrices.

Coordinates of objects which use their own coordinate space, such a text, templates, or patterns, must be transformed to user space before the coordinates can be used. How this must be done is described later.

Transformation matrices in DynaPDF are stored in the structure TCTM that is defined as follows:

```
struct TCTM
{
    double a;
    double b;
    double c;
    double d;
    double x;
    double y;
};
```

The identity matrix is [1, 0, 0, 1, 0, 0] (a, b, c, d, x, y).

The following overview lists the most important matrix manipulations:

- **Translations** are specified as [1, 0, 0, 1, tx, ty] where tx and ty represent the distances to translate the origin of the coordinate system in horizontal and vertical dimensions, respectively.

- **Scaling** is achieved by [sx, 0, 0, sy, 0, 0]. This scales the coordinates so that 1 unit in the horizontal and vertical dimensions of the new coordinate system is the same size as sx and sy units, respectively, in the previous coordinate system.

- **Rotations** are achieved by [$\cos \alpha$, $\sin \alpha$, -$\sin \alpha$, $\cos \alpha$, 0, 0], which has the effect of rotating the coordinate system axes by an angle α (measured in radians) counter clockwise.

- **Skew** is specified by [1, $\tan \alpha$, $\tan \beta$, 1, 0, 0], which skews the x-axis by an angle α, and the y-axis by an angle β (measured in radians).

All transformations above can occur in any combination in one transformation matrix.

Helper Functions

The following functions are helpful and some of them are required when working with transformation matrices. To achieve maximum execution speed the source codes of the most important functions are provided:

```cpp
// Distance between two points
double CalcDistance(double x1, double y1, double x2, double y2)
{
   double dx = x2-x1;
   double dy = y2-y1;
   return sqrt(dx*dx + dy*dy);
}
// Rotation angle in radians
double GetRotationAngle(TCTM &M)
{
   double x1 = 0.0;
   double y1 = 0.0;
   double x2 = 1.0;
   double y2 = 0.0;
   Transform(M, x1, y1);
   Transform(M, x2, y2);
   return atan2(y2-y1, x2-x1);
}
// Overall scaling factor
double GetScaleFactor(TCTM &M)
{
   #define sin45 0.70710678118654752440084436210485
   double x = sin45 * M.a + sin45 * M.c;
   double y = sin45 * M.b + sin45 * M.d;
   return sqrt(x*x + y*y);
}
// Scaling factor of the x-axis
double GetScaleX(TCTM &M)
{
   double x1 = 0.0;
   double y1 = 0.0;
   double x2 = 1.0;
   double y2 = 0.0;
   Transform(M, x1, y1);
   Transform(M, x2, y2);
   if (x1 > x2)
      return -CalcDistance(x1, y1, x2, y2);
   else
      return CalcDistance(x1, y1, x2, y2);
}
// Scaling factor of the y-axis
double GetScaleY(TCTM &M)
```

```cpp
{
   double x1 = 0.0;
   double y1 = 0.0;
   double x2 = 0.0;
   double y2 = 1.0;
   Transform(M, x1, y1);
   Transform(M, x2, y2);
   if (y1 > y2)
      return -CalcDistance(x1, y1, x2, y2);
   else
      return CalcDistance(x1, y1, x2, y2);
}
// Multiply two matrices. Note that matrix multiplications are not
// commutative. It is a difference whether the matrix M1 is multiplied
// with M2 or vice versa!
TCTM MulMatrix(TCTM &M1, TCTM &M2)
{
   TCTM retval;
   retval.a = M2.a * M1.a + M2.b * M1.c;
   retval.b = M2.a * M1.b + M2.b * M1.d;
   retval.c = M2.c * M1.a + M2.d * M1.c;
   retval.d = M2.c * M1.b + M2.d * M1.d;
   retval.x = M2.x * M1.a + M2.y * M1.c + M1.x;
   retval.y = M2.x * M1.b + M2.y * M1.d + M1.y;
   return retval;
}
// Transform a point with a matrix
void Transform(TCTM &M, double &x, double &y)
{
   double tx = x;
   x = tx * M.a + y * M.c + M.x;
   y = tx * M.b + y * M.d + M.y;
}
```

Text Coordinates and Metrics

As mentioned earlier text coordinates are defined in text space. The transformation from text space to user space is achieved by multiplying the text matrix with the one of the current graphics state. However, the graphics state contains several text related parameters which require some further explanation. The following sub-clauses describe in detail in which coordinate space these parameters are defined, how they can be transformed to user space, and how text metrics like the text width must be interpreted.

Font Size

The current font size, which is a parameter of the graphics state, is defined in text space. The visible font size is measured in user space, so, the value of the graphics state is not directly usable.

The transformation to user space can only be done in the used text callback function because the mapping from text space to user space depends on the text matrix which is only available in a text record.

The font size can be transformed to user space as follows:

```
// We must first transform the text matrix to user space. The text
// matrix is a parameter of the used text callback function. m_GState
// represents the current graphics state. The function GetScaleY() is
// described at Helper Functions.
TCTM m = MulMatrix(m_GState.Matrix, *Matrix);
double realFontSize = fabs(m_GS.FontSize * GetScaleY(m));
```

The above code is very simple but can be fairly optimized because the function GetScaleY() causes some unnecessary overhead. When extracting text from a PDF file, the start point of the text must always be calculated and the font size is always a positive value. The optimized code looks as follows:

```
double x1 = 0.0;
double y1 = 0.0;
double x2 = 0.0;
double y2 = m_GState.FontSize;
// Transform the text matrix to user space
TCTM m = MulMatrix(m_GState.Matrix, *Matrix);
Transform(m, x1, y1); // Start point of the text record
Transform(m, x2, y2); // Second point to compute the font size
double realFontSize = CalcDistance(x1, y1, x2, y2);
```

The main difference to the first code is that the start point of the text record can be used for further calculations and the real font size is already a positive value. Such optimizations are very important especially when using more complex algorithms to construct text lines or logically sorted text output.

Text Width

The text width, which is provided in all text callback functions, is calculated in text space. The provided text width includes already the graphics state parameters Character Spacing, Word Spacing, and Text Scaling. However, these parameters are still required if the width of a sub string must be computed. The parameters must be available in the graphics state including the corresponding callback functions which set the parameters.

The real text width measured in user space can be calculated as follows:

```
double x1 = 0.0;
double y1 = 0.0;
double x2 = Width; // Width is a parameter of the callback function
double y2 = 0.0;
// Transform the text matrix to user space
TCTM m = MulMatrix(m_GState.Matrix, *Matrix);
Transform(m, x1, y1); // Start point of the text record
```

```
Transform(m, x2, y2); // End point of the text record
double realTextWidth = CalcDistance(x1, y1, x2, y2);
```

The end point of a text record is usually required to determine whether the next record lies on the same line. An algorithm that is able to construct text lines in arbitrary rotated coordinate systems is provided in the example Text Extraction which is delivered with all DynaPDF versions.

Character Spacing

As described above the current character spacing is already considered in the text width that is provided in all text callback functions. However, the value must be stored in the graphics state if the width of a sub string must be computed. Character spacing is measured in unscaled font units. The required transformation to text space is done in functions like GetTextWidth().

Word Spacing

Like character spacing, the current word spacing is already considered in the text width that is provided in all text callback functions. However, word spacing applies to the space character of simple fonts only.

An application that extracts text from PDF files maybe wants to preserve the original formatting of the text. In this case, the distance between two words in the same text record must be known, e.g. to insert a number of spaces to emulate the word spacing.

However, note that the current word spacing must be ignored if the font type is ftType0 (the font type is a parameter of the graphics state and is set with the TSetFont callback function).

Another thing that must be considered is that word and character spacing are measured in unscaled font units. The width of a space character including word spacing can be calculated with the function GetTextWidth() that is part of the font API (the name is fntGetTextWidth() in C/C++).

An algorithm that considers word spacing must check whether the source string contains space characters. If a space was found, the width of the sub string that occurs before must be calculated so that the start and end point of the word can be calculated. Additional spaces can be skipped and the cursor position is updated to the position behind the spaces. Processing continues until the entire text of the record was processed.

An algorithm that processes text in this way calculates essentially the start and end coordinates of every text part that is either separated by spaces or kerning space.

The required source code looks as follows (C++):

```
// The following code fragment uses the TShowTextArrayW callback function.
SI32 parseShowTextArrayW(const void* Data, const struct TTextRecordA*
Source, struct TCTM* Matrix, const struct TTextRecordW* Kerning, UI32
Count, double Width, LBOOL Decoded)
{
   if (!Decoded) return 0;
   double x1 = 0.0;
   double y1 = 0.0;
```

```
// Transform the text matrix to user space
TCTM m = MulMatrix(m_GState.Matrix, *Matrix);
Transform(m, x1, y1); // Start point of the text record
// Word spacing applies to simple fonts only.
if (m_GState.FontType != ftType0)
{
   UI32 i, j, last;
   double x2 = 0.0;
   double y2 = 0.0;
   double textWidth = 0.0;
   for (i = 0; i < Count; i++)
   {
      j    = 0;
      last = 0;
      // Consider kerning space
      if (Source[i].Advance != 0.0f)
      {
         textWidth -= Source[i].Advance;
         x1 = textWidth;
         y1 = 0.0;
         // Update the cursor position
         Transform(m, x1, y1);
      }
      // Scan for spaces
      while (j < Source[i].Length)
      {
         if (Source[i].Text[j] != 32)
            ++j;
         else
         {
            // A space was found
            if (j > last)
            {
               // The text must be taken from the Source array!
               textWidth += fntGetTextWidth( m_GState.ActiveFont,
                                             Source[i].Text + last,
                                             j - last,
                                             m_GState.CharSpacing,
                                             m_GState.WordSpacing,
                                             m_GState.TextScale);
               x2 = textWidth;
               y2 = 0.0;
               // Calculate the end point.
               Transform(m, x2, y2);
               // We have now the exact start and end coordinate of the
               // text's baseline. The text could now be written to a
               // file or added to a list of text records depending on
               // how the algorithm processes text. Note that the
               // string length in the source and Unicode (Kerning)
```

```
              // record can differ, e.g. if a source character is
              // decomposed to two or more Unicode characters. In this
              // case, the function fntTranslateString2() can be used
              // to translate the sub string to Unicode because we
              // don't know where we are in the already existing
              // Unicode string.
              /*AddText(x1, y1,      // Start point
                        x2, y2,      // End point
                        Source[i],   // Source record
                        Kerning[i], // Translated Unicode record
                        last,        // String offset
                        j - last);   // String length
              */
          }
          last = j++;
          // Skip additional spaces
          while (j < Source[i].Length && Source[i].Text[j] == 32)
          {
              ++j;
          }
          textWidth += fntGetTextWidth( m_GState.ActiveFont,
                                        Source[i].Text + last,
                                        j - last,
                                        m_GState.CharSpacing,
                                        m_GState.WordSpacing,
                                        m_GState.TextScale);
          last = j;
          x1 = textWidth;
          y1 = 0.0;
          // Update the cursor position.
          // We are now behind the space(s).
          Transform(m_Matrix1, x1, y1);
      }
  }
  // All spaces in the string were processed. Do we have some
  // remaining text?
  if (j > last)
  {
      textWidth += fntGetTextWidth( m_GState.ActiveFont,
                                    Source[i].Text + last,
                                    j - last,
                                    m_GState.CharSpacing,
                                    m_GState.WordSpacing,
                                    m_GState.TextScale);
      x2 = textWidth;
      y2 = 0.0;
      Transform(m, x2, y2);
      /*AddText(x1, y1,
                x2, y2,
```

```
                        Source[i],
                        Kerning[i],
                        last,
                        j - last);*/

            // Update the cursor position. More text can follow.
            x1 = x2;
            y1 = y2;
        }
    }
    }
}
```

DynaPDF is delivered with several example projects which demonstrate how text coordinates must be computed and how text can be extracted. The above code is a fragment of the example project Text Coordinates which is delivered with all DynaPDF versions.

Text Scaling

Like character and word spacing the current text scaling is already considered in the text width that is provided in all text callback functions. However, the value must be stored in the graphics state if the width of a sub string must be computed. Text scaling is measured in percent of the original unscaled text width.

Sub string coordinates

Sub string processing is somewhat more complicated because the width of a sub string cannot be calculated from the Unicode string and one source character does necessarily correspond to one Unicode character.

Simple fonts use one byte encodings where one source byte can be decoded to one or more Unicode characters. CID fonts support also multi-byte encodings with fixed and variable code lengths. A sequence of n source bytes can be decoded to m Unicode characters. So, there is no logical relationship between the source and converted Unicode string.

If a text search algorithm should provide the coordinates of a found string, then it must be able to find the position of the search text in the source string because it is not possible to calculate the string width from the Unicode string. DynaPDF provides several helper functions to calculate the width of a sub string or to convert an arbitrary source string manually to Unicode. It is always possible to calculate the exact position of a string but the recommended strategy depends on the used text callback function and on the kind of algorithm that should be developed:

- Text extraction algorithms require usually not the exact position of every character or word in a string. Coordinates of sub strings are only required if word spacing must be considered but word spacing refers to simple fonts only. Because the code length of a simple font is always one byte the string width can be easily computed with fntGetTextWidth() and in cases where the source string is shorter than the Unicode string the source string can be manually converted to Unicode with TranslateRawString2() (the name is

fntTranslateRawString2() in C/C++). For this kind of algorithm the TShowTextArrayW callback function should be used because it provides anything required to develop fast text extraction algorithms. The example projects text_extraction and text_coordinates demonstrate how text extraction algorithms can be developed.

- Text search algorithms could use the TShowTextArrayW callback function too but the usage is much more complicated if strings of CID fonts must be processed. CID fonts support encodings with arbitrary code lengths from one through four bytes per character. Because the string width cannot be computed from the translated Unicode string the function must be able to find the position in the source string. This is not easy especially if the search text was stored in multiple text records.

  To simplify the development of text search algorithms the content parser provides the TShowTextArrayA callback function which returns the raw source strings. The conversion to Unicode can be done with TranslateRawCode() (the name is fntTranslateRawCode() in C/C++). The function converts a sequence of source bytes to Unicode and calculates the width of that character. The advantage is that the exact position of every character in a string can be easily calculated independent of the current font type. The overhead due to the call on a per character basis is not large because the function is strongly optimized to improve processing speed. The example text_search demonstrates how a text search algorithm can be developed.

Using the Content Parser

The content parser can be used to extract text, vector graphics, and images from a PDF file. The following sections describe which callback functions must set, what must be stored in the graphics state, as well as other important aspects.

Note that DynaPDF is delivered with several example projects which demonstrate how the content parser can be used. Before developing your own code take a look into the examples text_extraction, text_search, or image_extraction.

Text Extraction or Text Search Algorithms

The following callback functions should be set to process PDF text:

```
TBeginTemplate
TEndTemplate          // Optional
TMulMatrix
TSetCharSpacing
TSetFont
TSetLeading           // Optional
TRestoreGraphicState
TSaveGraphicState
TSetFillColor         // Optional
TSetStrokeColor       // Optional
TSetTextDrawMode
```

```
    TSetTextScale
    TSetWordSpacing
    TShowTextArrayA or    // Preferred for text search algorithms
    TShowTextArrayW       // Preferred for text extraction algorithms
```

Which text callback function should be used depends on the requirements of the algorithm. If the coordinates of arbirary sub strings must be computed the usage of the TShowTextArrayA callback function in combination with TranslateRawCode() is recommended. If sub string processing not required then TShowTextArrayW should be used.

Patterns, shadings, images, and vector graphics can be ignored when extracting text. So, it is not required to set related callback functions.

The graphics state should contain these variables:

Parameter	Type	Initial Value
CharSpacing	float	0.0f
FillColor	UI32 (see comment below)	Black (optional)
Font	IFont*	NULL
FontSize	double	0.0
FontType	TFontType	–
Leading	float	0.0f (optional)
Matrix	TCTM	{1, 0, 0, 1, 0, 0}
SpaceWidth	float	0.0f
StrokeColor	UI32 (see comment below)	Black (optional)
TextDrawMode	TDrawMode	dmNormal
TextScale	float	100.0f
WordSpacing	float	0.0f

The text color is normally not required when extracting text. If it should be considered the current fill and stroke color must be available in the graphics state and the corresponding callback functions must be set. Whether the current fill or stroke color must be used as text color depends on the text draw mode (see also SetTextDrawMode()).

Note that colors in PDF are represented by an array of double where each component ranges from 0.0 through 1.0 of the corresponding color space. However, colors are normally processed in a unique device color space such as DevicedGray, DeviceRGB, or DeviceCMYK because applications which use the content parser have normally no build-in support for PDF color spaces. Colors which are set by the TSetFillColor and TSetStrokeColor callback functions should be converted to a device color space with ConvColor(). The resulting device color must then be stored in the graphics state.

The TEndTemplate callback function is optional because no specific operation is required to be executed when a template is leaved.

Unicode conversion

The extraction of human readable text requires a conversion to a well known encoding like Unicode because PDF strings are not necessarily human readable.

Whether it is possible to convert a PDF string to Unicode depends on whether the required encoding information is available. This is always the case if a font uses a predefined encoding like WinAnsi or MacRoman or if the glyph names of Type1 fonts are available in the Adobe Glyph List or ZapfDingbats encoding.

Fonts which use a symbol encoding can provide a ToUnicode CMap which offers the required mapping to Unicode. However, this CMap is optional and is not necessarily available. If a symbol font does not contain a ToUnicode CMap the strings are converted to the code page 1252.

External CMaps

A widely used technique to reduce the amount of data that must stored in a PDF file is the usage of non embedded CID fonts. CID fonts, whether embedded or not, can depend on external CMap files which offer the required mapping to Unicode.

To process strings of such fonts correctly, DynaPDF must be able to load required CMap files if necessary. Therefore, DynaPDF is delivered with the most important CMap files which are provided by Adobe Systems. These CMaps can be found in the DynaPDF installation directory at /Resource/CMap/. Applications which extract text from PDF files should include these CMaps so that they can be loaded at runtime.

The search path to external CMaps must be set with SetCMapDir() before executing ParseContent() the first time. The function creates a CMap cache that is hold in memory until the PDF instance will be deleted. The search path(s) to external CMap files should be set only one time per PDF instance and one PDF instance should be used to process so many PDF files as possible. This can significantly improve processing speed.

If a required CMap is absent the *Decoded* parameter of the TShowTextArrayW callback function is set to false and the string should be ignored in this case because no meaningful values can be returned.

Inside the Callback Functions

The following sub-clauses describe important operations which must be executed in the callback functions to achieve correct results.

TBeginTemplate

This callback function is executed when a Form XObject in PDF terms is painted (we call this object type template in DynaPDF). The parameter *BBox* specifies the template's bounding box or visible area measured in form space. The required mapping from form space to user space is specified by the *Matrix* parameter (see also Coordinate Spaces). The *Matrix* parameter is optional and can be set to NULL. In the latter case, the form matrix is set to the identity matrix.

If the form matrix is present then it must be multiplied with the current transformation matrix of the graphics state as follows:

```
// m_GState represents the current graphics state
if (Matrix)
{
   m_GState.Matrix = MulMatrix(m_GState.Matrix, *Matrix);
}
```

The corresponding TEndTemplate callback function is executed if the contents of a template were fully processed. The definition of this callback function is optional because no specific code is required to be executed.

Return values:

- 0 indicates success and processing continues.
- 1 means that the template should be skipped.
- Any other return value breaks processing.

TMulMatrix

The TMulMatrix callback function is executed if the current transformation matrix must be multiplied with a new one. When using the function MulMatrix() the matrix must be multiplied with the one of the graphics state as follows:

```
// m_GState represents the current graphics state
m_GState.Matrix = MulMatrix(m_GState.Matrix, *Matrix);
```

The parameter *Matrix* is always set; it is not required to check whether the parameter is set to NULL.

TSetFont

The TSetFont callback function activates a font in the graphics state. The following variables of the graphics state must be set when the callback function is executed:

```
m_GState.ActiveFont = Font;
m_GState.FontSize   = FontSize;
m_GState.FontType   = Type;
m_GState.SpaceWidth = (float)(fntGetSpaceWidth(Font, FontSize) * 0.5);
```

The space width is required to construct text lines. It is usually best to use the half space width to determine whether a space must be inserted at a specific position because many PDF fonts do not

contain a space character and the function fntGetSpaceWidth() returns just a default width in this case (the function name is GetSpaceWidth() in C#, Delphi, and VB .Net).

The space width is measured in text space like the member *Advance* of a kerning record. However, the distance between two text records is measured in user space! So, the space width must be transformed to user space before it can be compared with the distance between two text records. See the description of the TShowTextArrayW callback function for further information.

TRestoreGraphicState

As the name of the function suggests the graphics state must be restored when the callback function is executed. The graphics state stack is a LIFO (last in first out) data structure that stores copies of the graphics state. The last element from the graphics state stack must be popped and the values must be copied to the current graphics state.

It is guaranteed that this function is not executed without a preceeding TSaveGraphicState call. Invalid restore graphics state operators are trated as an error. The parser breaks processing when it encounters such an error.

The return value of the function must be zero. Any other return value breaks processing.

TSaveGraphicState

As the name suggests the current graphics state must be saved when the callback function is executed. As already described at TRestoreGraphiState the graphics state stack is a LIFO (last in first out) data structure that stores copies of the graphics state. The application must create a copy of the current graphics state and push it onto the graphics state stack.

It is **not** guaranteed that all saved graphics states will be restored until the page was fully parsed. The application must make sure that remaining copies of the graphics state will be released before parsing the next page or terminating the application.

The return value of the function must be zero. Any other return value breaks processing.

TShowTextArrayA

This is the preferred callback function to develop text search algorithms. See also Sub string coordinates for further information. The function returns the source strings which can be converted to Unicode with TranslateRawCode(). TranslateRawCode() converts the source string on a per character basis and calculates the width of that character in one pass.

The parameter *Width* represents the width of the entire text record. The source array provides also the displacement vector *Advance*. Advance is a vector also if only coordinate is given; the y-coordinate is always zero. Positive values of *Advance* move the cursor to the **left**; negative values move it to the **right** in a non-rotated coordinate system. The string widths and the displacement vector are measured in text space.

The displacement vector is often used to apply kerning between two characters but it can also be used to emulate spaces or to move the cursor to an arbitrary position on the x-axis of the text line. Because CID fonts do not support word spacing, spaces are very often emulated with the displacment vector.

The source strings are **not** null-terminated. The array can also contain strings with zero length. In this case only the displacement vector *Advance* must be considered.

DynaPDF is delivered with the example text_search which demonstrates how a text search algorithm can be developed. This project should be used as basis to develop your own code.

TShowTextArrayW

This is the preferred callback function to develop text extraction algorithms. See also Sub string coordinates for further information.

The arrays *Source* and *Kerning* contain the source and translated Unicode strings of a text record. Both arrays contain always the same number of elements (parameter *Count*).

The parameter *Width* represents the width of the enitre text record. The kerning array provides also the width of each sub record and the displacement vector *Advance*. Advance is a vector also if only coordinate is given; the y-coordinate is always zero. Positive values of *Advance* move the cursor to the left; negative values move it to the right in a non-rotated coordinate system. The string widths and the displacement vector are measured in text space.

The displacement vector is often used to apply kerning between two characters but it can also be used to emulate spaces or to move the cursor to an arbitrary position on the x-axis of the text line. Because CID fonts do not support word spacing, spaces are very often emulated with the displacment vector.

The source strings are required if the width of a sub string must be calculated. Note that it is not possible to calculate the the width of a sub string from the Unicode string.

It is possible that one or more sub records contain strings with a zero length. In this case, only the displacment vector Advance must be considered.

DynaPDF is delivered with the examples text_extraction and text_coordinates which demonstrate how text extraction algorithms can be developed and how text coordinates must be calculated. One of these projects should be used as basis to develop your own code.

Image Extraction

The following callback functions should be set to extract images:

```
TBeginTemplate
TEndTemplate          // Optional
TInsertImage
TMulMatrix            // Optional
TRestoreGraphicState  // Optional
TSaveGraphicState     // Optional
```

```
TSetFillColor         // Optional
```

It is possible to extract images without a graphics state. The destination width and height, the output resolution, and the parallelogram into which the image would be drawn are already pre-computed and can be taken from the sructure TPDFImage.

1 bit images require a special processing if the image is drawn as an image mask. This is the case if no color table is present and if the member *Transparent* of the TPDFImage structure is set to true. Non-zero pixels are are rendered in the current fill color while zero pixel values produce no output. To simply processing the current fill can be taken from the structure TPDFImage.

The provided fill color is already converted to the specified device color space if one of the color conversion flags is set (pfConvImagesToGray, pfConvImagesToRGB, or pfConvImagesToCMYK), or it was converted to the nearest device color space.

If a graphics state is used then it should contain at least these variables:

Parameter	Type	Initial Value
FillColor	double[32] or UI32	Black (0)
FillColorSpace	TExtColorSpace, IColorSpace*	esDeviceGray, NULL (optional)
Matrix	TCTM	{1, 0, 0, 1, 0, 0}

The fill color can be stored as array of double if the original values are of interest or as unsigned integer if colors are converted to a device space with ConvColor().

The visible size of an image, measured in user space, can be calculated from the current transformation matrix as follows:

```
double x1 = 0.0;
double y1 = 0.0;
double x2 = 1.0;
double y2 = 0.0;
double x3 = 0.0;
double y3 = 1.0;
// m_GState represents the current graphics state
Transform(m_GState.Matrix, x1, y1); // lower left corner
Transform(m_GState.Matrix, x2, y2); // lower right corner
Transform(m_GState.Matrix, x3, y3); // upper left corner
// CalcDistance() is described at Helper Functions.
double w = CalcDistance(x1, y1, x2, y2); // Width in user space
double h = CalcDistance(x1, y1, x3, y3); // Height in user space
```

The above values are already pre-computed and can be taken from the structure TPDFImage.

An image can be used several times in a document, e.g. if it represents a logo or other repeating contents. To avoid the extraction of unnecessay duplicates a duplicate check should be performed before extracting an image. If the image is **not** an inline image, the member *ObjectPtr* of the structure TPDFImage represents a unique pointer to the image object.

Inline images are fully defined in the content stream and cannot be used repeatly. A duplicate check is not required for these images. The only way to compare two inline images is to compare the image data.

A duplicate check should also be performed for templates. Templates are normally used for repeating contents such as fixed page backgrounds, logos and so on. To skip a template the TBeginTemplate calback function must return 1.

Physical organization of images

The visible appearance and the physical structure how an image is stored in a PDF file is sometimes somewhat confusing. Images can be split into bands or tiles. There are various reasons why this can be done but if an image was split into smaller pieces then it is very difficult to restore the original image. DynaPDF does not provide algorithms which try to identify pieces of a larger image.

It is usually best to ignore images which are less than two units high. For example, applications like Microsoft Word split images often into separate scan lines or smaller pieces if the image contained transparent areas. The resulting PDF file contains then hundreds or thousands of very small images. Because such small pieces are not really meaningful when viewed alone, the application can either try to reconstruct the original image, or if this is not possible, such images should be skipped.

If the content parser returns lots of very small images then it is usually best to provide a fallback that renders the entire page with RenderPage() or RenderPageToImage().

Image coordinate space

An image occupies a rectangle in image space w units wide and h units high, where w and h are the width and height of the image in samples or pixels. Each sample occupies one square unit. The coordinate origin (0, 0) is at the upper-left corner of the image, with coordinates ranging from 0 to w horizontally and 0 to h vertically.

The image's sample data is ordered by row, with the horizontal coordinate varying most rapidly. The upper-left corner of the first sample is at coordinates (0, 0), the second at (1, 0), and so on through the last sample of the first row, whose upper-left corner is at (0, 0) and whose upper-right corner is at (w - 1, 0). The next samples after that are at coordinates (0, 1), (1, 1), and so on to the final sample of the image, whose upper-left corner is at (0, 0) and whose lower-right corner is at (w-1, h-1).

The correspondence between image space and user space is constant: the unit square of user space, bounded by user coordinates (0, 0) and (1, 1), corresponds to the boundary of the image in image space. Following the normal convention for user space, the coordinate (0, 0) is at the lower-left corner of this square, corresponding to coordinates (0, h) in image space. The implicit transformation from image space to user space, if specified explicitly, would be described by the matrix [1/w, 0, 0, -1/h, 0, 1].

An image can be placed on the output page in any position, orientation, and size by modifying the current transformation matrix to map the unit square of user space to the rectangle or parallelogram

in which the image shall be painted. Typically, this is done within a pair of save/restore graphics state operators to isolate the effect of the transformation, which can include translation, rotation, reflection, and skew.

Please note that the content parser returns images as is. It is maybe required to mirror an image on the x- or y-axis or to rotate it according to the current transformation matrix. DynaPDF does not provide functions to manipulate images.

Helper functions

When extracting images it is usually required to create some kind of image file from the raw image data that is returned by the content parser. The following helper functions can be used to create image files:

- `CreateImage()`
- `AddImage()`
- `CloseImage()`

CreateImage() supports the creation of single and multi-page images (TIFF only). To create a multi-page image the function must be called before ParseContent() or before parsing the first page. If all images should be stored in the image file the flag pfDecomprAllImages must be set.

The function converts images that use a complex color space to the nearest device color space by default. It is also possible to convert images to a specific device color space by setting one of the flags pfConvImagesToGray, pfConvImagesToRGB, or pfConvImagesToCMYK.

Also if a color conversion flag is set, 1 bit images and gray images are never converted into a higher color space.

JPEG and JPEG 2000 images are already stored in valid file formats in PDF. If the creation of a multi-page TIFF image is not required then it is usually best to extract these image formats as is. In this case the flag pfDecomprAllImages must be absent.

16 bit images

Images which use 16 bits per component are supported by these encoders:

- JPEG 2000
- PNG
- TIFF

All other encoders do not yet support 16 bit images. Note also that 16 bit CMYK images are not supported by most image viewers.

DynaPDF is delivered with the demo project image_extraction that demonstrates how images can be extracted.

Vector Graphics

To extract vector graphics the following callback functions should be set:

```
TBeginPattern
TEndPattern            // Optional
TBeginTemplate
TBezierTo1
TBezierTo2
TBezierTo3
TClipPath
TClosePath
TEndTemplate           // Optional
TLineTo
TMoveTo
TMulMatrix
TRectangle
TRestoreGraphicState
TSaveGraphicState
TSetExtGState          // Optional
TSetFillColor
TSetLineCapStyle
TSetLineDashPattern
TSetLineJoinStyle
TSetLineWidth
TSetMiterLimit
TSetStrokeColor
```

The graphics state should contain these variables:

Parameter	Type	Initial Value
Clipping Path	Vector array	Crop box or Media box
DashPattern	double*	NULL (Solid line)
DashPhase	UI32	0
FillColor	double[32]	Black
FillColorSpace	TExtColorSpace, IColorSpace*	esDeviceGray, NULL
FillPattern	IPattern*	NULL
LineCapStyle	TLineCapStyle	csButtCap
LineJoinStyle	TLineJoinStyle	jsMiterJoin
LineWidth	float	0.0f
Matrix	TCTM	{1, 0, 0, 1, 0, 0}
MiterLimit	float	10.0f
StrokeColor	double[32]	Black
StrokeColorSpace	TExtColorSpace, IColorSpace*	esDeviceGray, NULL
StrokePattern	IPattern*	NULL

If vector graphics should be processed in a unique device color space then it is usually best to convert PDF colors directly to the device color space with ConvColor() and to store the device color then in the graphics state. The data type of the members FillColor and StrokeColor must then be changed to unsigned int (UI32). There is no need to store the fill and stroke color space in the graphics state in this case.

Vector graphics are defined in page or form space depending on the object type in which the graphics are stored. The mapping to user space is achieved by tranforming the coordinate pairs with the current transformation matrix of the graphics state.

For example, if the TMoveTo callback function would be executed the application must transform the provided coordinates to user space as follows:

```
// x and y are the parameters of the TMoveTo callback function
Transform(m_GState.Matrix, x, y);
```

Vector graphics in PDF are always defined as paths. A path is invisible until it will be filled or stroked. PDF supports primitive path painting operators like bezier curves, lines, and rectangles only. Circles and ellipses are always drawn with bezier curves in PDF.

Once a path was fully defined it is normally consumed by a TClosePath() or TClipPath() callback function call. If this is not the case the path must be deleted when the page was fully processed.

When a path must be filled, stroked, or discarded the TClosePath callback function is executed. The parameter *Mode* of the callback function specifies what must be done with the path:

```
fmFillNoClose            // Fill the path (Winding)
fmFillEvOddNoClose       // Fill the path (Even Odd)
fmFillStroke             // Close, fill and stroke the path (Winding)
fmFillStrokeEvOdd        // Close, fill and stroke the path (Even Odd)
fmFillStrokeNoClose      // Fill and stroke the path (Winding)
fmFillStrokeEvOddNoClose // Fill and stroke the path (Even Odd)
fmStrokeNoClose          // Stroke the path
fmStroke                 // Close and stroke the path
fmNoFill                 // Discard the path
fmClose                  // Close the path with a straight line segment
```

A normal path is **not** part of the graphics state. Once a path was drawn it must be deleted.

The TClipPath callback function supports the same modes than TClosePath but the required actions that must be executed are different. A clipping path is part of the graphics state. If the graphics state contains already a clipping path, the existing path must be intersected with new one. The resulting clipping path must then be stored in the graphics state.

If the path must also be filled or stroked, it is important to hold an unmodified version of the path in memory so that the required path painting operation can be executed. The previously created clipping region must of course be considered.

The mode fmNoFill has the meaning create the clipping region, but do not fill or stroke the path. So, after the clipping region was created the path can be deleted from memory. This is the default behavior. The combination of a path painting operator with a clipping path occurs seldom.

PlaceImage

Syntax:

```
LBOOL pdfPlaceImage(
   const PPDF* IPDF,    // Instance pointer
   SI32 ImgHandle,      // Image handle
   double PosX,         // X-Coordinate of destination rectangle
   double PosY,         // Y-Coordinate of destination rectangle
   double ScaleWidth,   // Scaled width of destination rectangle
   double ScaleHeight)  // Scaled height of destination rectangle
```

Images can be used multiple times on different positions and with different sizes. This function places an image onto a page or template that was already inserted beforehand by an image function.

The image can be placed in the same way as by every other image function. However, if the size of the image will be changed, the image is not rescaled physically to the current resolution. Insert the largest version of an image first to avoid up-scaling.

The width and height can be calculated as follows:

- If *ScaleWidth* or *ScaleHeight* is **-1** the function uses the original width or height from the image. If both parameters are **-1** the image will be inserted with a resolution 72 DPI.

- If *ScaleWidth* **or** *ScaleHeight* is **0**, the missing value is calculated in relation to the given value of *ScaleHeight* or *ScaleWidth* to preserve the image's aspect ratio. The resulting output is an image with exact proportions relative to its original size.

- If *ScaleWidth* **and** *ScaleHeight* are **0**, the original size is used (same effect as -1).

- A negative value of *Width* or *Height* mirrors the image on the x- and or y-axis.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

PlaceSigFieldValidateIcon

Syntax:

```
LBOOL pdfPlaceSigFieldValidateIcon(
   const PPDF* IPDF, // Instance pointer
   UI32 SigField,    // Handle of a signature field
   double PosX,      // X-Coordinate of the icon
   double PosY,      // Y-Coordinate of the icon
   double Width,     // Icon width
   double Height)    // Icon height
```

The function places the validation icon within a signature field to the wished position. By default, the validation icon is scaled to the width or height of the signature field and it is horizontally and vertically centered. When creating a user defined appearance for a signature field it is sometimes

required to place the validation icon on another position within the signature field so that no important contents will be overdrawn.

The unscaled size of the validation icon is 100 x 100 units. It can be scaled to every size you want but it is usually best to preserve the aspect ratio. The icon should normally be placed fully inside the signature field. If the validation icon should not be shown then place it outside the signature field.

The demo package of DynaPDF contains the project signature_ap that demonstrates the usage of the function as well as the creation of a user defined signature appearance. See also CreateSigFieldAP().

Remarks:

Please note that the position of the validation icon is no property that can be stored in the PDF file. The position and size that you set with this function can only be considered if the file will be signed later with DynaPDF (see CloseAndSignFile(), CloseAndSignFileEx(), or CloseAndSignFileExt() for further information).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

PlaceTemplate

Syntax:

```
LBOOL pdfPlaceTemplate(
   const PPDF* IPDF,    // Instance pointer
   SI32 TmplHandle,     // Template handle
   double PosX,         // X-Coordinate
   double PosY,         // Y-Coordinate
   double ScaleWidth,   // Scaled width
   double ScaleHeight)  // Scaled height
```

The function places a template on a page, another open template, or pattern. The parameter *TmplHandle* must be a valid template handle that was returned by BeginTemplate(), ImportPage() or ImportPageEx().

Templates can be used multiple times on different pages or positions and with different sizes. Unlike images, a template can be scaled without losing quality as far as the template contains vector graphics and text objects only.

The calculation of the width and height is the same as for images:

- If *Width* or *Height* is **-1** the function uses the mirrored original width or height from the template.

- If *Width* **or** *Height* is **0**, the missing value is calculated in relation to the given value of *Height* or *Width* to preserve the template's aspect ratio. The resulting output is a template with exact proportions relative to its original size.

- If *Width* **and** *Height* are **0**, the original size is used (same effect as -1 but the template is not mirrored).

- A negative value of *Width* or *Height* mirrors the template on the x- or y-axis.

Notice:

Imported pages can contain non-normalized bounding boxes and the page can be rotated. The coordinate origin, crop box, and rotation angle of a page must be considered when placing a template on a page because templates do not support features like a crop box or individual orientation. The required calculations to find the correct coordinate origin to de-rotate a template are relatively complex. Therefore, DynaPDF provides the function PlaceTemplateEx() which considers the coordinate origin, orientation, and crop box automatically. PlaceTemplateEx() greatly simplifies the handling of templates. If possible, this function should be used when working with imported pages.

Remarks:

A template is invisible as long it was not placed on a page, template, or pattern.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

PlaceTemplateEx

Syntax:

```
LBOOL pdfPlaceTemplateEx(
  const PPDF* IPDF,    // Instance pointer
  SI32 TmplHandle,     // Template handle
  double PosX,         // X-Coordinate
  double PosY,         // Y-Coordinate
  double ScaleWidth,   // Scaled width
  double ScaleHeight)  // Scaled height
```

The function places a template on a page, another open template, or pattern. The parameter *TmplHandle* must be a valid template handle that was returned by BeginTemplate(), ImportPage() or ImportPageEx().

The function was specifically designed to work with templates which were created from imported pages. A PDF page is converted to a template if it was imported with ImportPage() or ImportPageEx() outside of an open page.

The calculation of the width and height is the same as for images:

- If *Width* or *Height* is **-1** the function uses the mirrored original width or height from the template.

- If *Width* **or** *Height* is **0**, the missing value is calculated in relation to the given value of *Height* or *Width* to preserve the template's aspect ratio. The resulting output is a template with exact proportions relative to its original size.

- If *Width* **and** *Height* are **0**, the original size is used (same effect as -1 but the template is not mirrored).

- A negative value of *Width* or *Height* mirrors the template on the x- or y-axis.

A template does not support a crop box or orientation angle. When placing a template on a page that was created from an imported PDF page, the coordinate origin must be adjusted according to the original media or crop box. In addition, the template must maybe be drawn into a clipping path (if the original page contained a crop box), and the template must maybe also be rotated so that the original orientation can be preserved.

All these considerations complicate the handling of imported templates. To make the usage easier the function considers the coordinate origin, it draws the template automatically into a clipping rectangle if required, and it rotates the template according to the original page orientation.

The only thing that must be considered is that the original size of the template must be calculated from the crop box if present. In addition, the so computed width and height must be exchanged if the original page orientation was 90, -90, 270, or -270 degrees. See example below.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (C++)

The following code collects 6 imported pages on one destination page. The imported pages are placed horizontally and vertically centered into a fixed destination rectangle.

```cpp
// This is our error callback function
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode,
const char* ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}

// This function places a template horizontally and vertically centered
// into a destination rectangle
SI32 PlaceCenteredEx(const void* PDF, // Instance pointer
                  SI32 Tmpl,          // Template handle
                  double SrcWidth,    // Logical page width
                  double SrcHeight,   // Logical page height
                  double DestX,       // Destination x-coordinate
                  double DestY,       // Destination y-coordinate
                  double DestWidth,   // Destination width
                  double DestHeight)  // Destination height
{
    double x, y, w, h, sx;
    sx = DestWidth / SrcWidth; // Compute the scaling factor

    // Which side must be centered?
```

```cpp
    if (SrcHeight * sx <= DestHeight)
    {
        w = SrcWidth  * sx;
        h = SrcHeight * sx;
        x = DestX + (DestWidth  - w) * 0.5;
        y = DestY + (DestHeight - h) * 0.5;
        pdfPlaceTemplateEx(PDF, Tmpl, x, y, DestWidth, 0.0);
    }else
    {
        sx = DestHeight / SrcHeight;
        w = SrcWidth  * sx;
        h = SrcHeight * sx;
        x = DestX + (DestWidth  - w) * 0.5;
        y = DestY + (DestHeight - h) * 0.5;
        pdfPlaceTemplateEx(PDF, Tmpl, x, y, 0.0, DestHeight);
    }
    // We draw the destination rectangle on the page so that we can check
    // whether the position was correctly calculated.
    pdfSetLineWidth(PDF, 0.5);
    return pdfRectangle(PDF, DestX, DestY, DestWidth, DestHeight, fmStroke);
}

// The PDF instance must be created outside the function
LBOOL TestTemplates(const void* PDF)
{
    bool newPage = true;
    SI32 i, tmpl, orientation = 0;
    TPDFRect mediaBox, cropBox;
    double w, h;
    // Create a new PDF file
    pdfCreateNewPDF(PDF, "cppout.pdf");
    pdfSetPageCoords(PDF, pcTopDown);
    // Open an existing PDF file
    pdfOpenImportFile(PDF, "c:/PDFs/test.pdf", ptOpen, NULL);

    double x = 50.0;
    double y = 50.0;

    SI32 count = pdfGetInPageCount(PDF);
    for (i = 1; i <= count; i++)
    {
        // Get the page format and orientation angle of the source page.
        // We need the original page format to center the template into the
        // destination rectangle.
        pdfGetInBBox(PDF, i, pbMediaBox, mediaBox);
        pdfGetInBBox(PDF, i, pbCropBox, cropBox);
        orientation = pdfGetInOrientation(PDF, i);
        // Import the page as a template
        tmpl = pdfImportPage(PDF, i);
        // Calculate the logical width and height of the page.
        w = cropBox.Right - cropBox.Left;
        h = cropBox.Top - cropBox.Bottom;
        if (w < 1.0)
        {
            w = mediaBox.Right - mediaBox.Left;
            h = mediaBox.Top - mediaBox.Bottom;
        }
        // Exchange the widht and height if necessary
        switch(orientation)
        {
            case 90:
```

```
            case -90:
            case 270:
            case -270:
            {
                double t = w;
                w = h;
                h = t;
            }
        }
        if (newPage)
        {
            x = 50.0;
            y = 50.0;
            pdfAppend(PDF);
            PlaceCenteredEx(PDF, tmpl, w, h, 50.0, 50.0, 227.5, 227.3);
            x += 247.5;
            newPage = false;
        }else
        {
            PlaceCenteredEx(PDF, tmpl, w, h, x, y, 227.5, 227.3);
            x += 247.5;
            if (x > 500.0)
            {
                x   = 50.0;
                y += 247.3;
            }
            if (y > 700.0)
            {
                pdfEndPage(PDF);
                newPage = true;
            }
        }
    }
    // Close the last open page
    if (!newPage) pdfEndPage(PDF);
    return pdfCloseFile(PDF);
}
```

PlaceTemplByMatrix

Syntax:

```
LBOOL pdfPlaceTemplByMatrix(
  const PPDF* IPDF, // Instance pointer
  SI32 TmplHandle)  // Template handle
```

This is an internal debug function which is used to test the content parser of DynaPDF. The function uses the current transformation matrix to place a template on the page.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

PolygonAnnot

Syntax:

```
SI32 pdfPolygonAnnot(
    const PPDF* IPDF,           // Instance pointer
    struct TFltPoint* Vertices, // At least two vertices must be provided
    UI32 NumVertices,           // Number of vertices in the array
    double LineWidth,           // Line width
    UI32 FillColor,             // Fill color or NO_COLOR = transparent
    UI32 StrokeColor,           // Stroke color or NO_COLOR = transparent
    TPDFColorSpace CS,          // Color space of the color values
    const char* Author,         // Optional author
    const char* Subject,        // Optional subject
    const char* Content)        // Optional content or comment
```

The function creates a Polygon Annotation. The vertices are connected by straight lines. The path is always closed before it will be drawn. It is not required to close the path explicitly. At least two vertices must be provided.

The coordinates of the vertices are interpreted in current user space. Any transformation that was applied on the coordinate system will be taken into account.

The stroke or fill color can be set to the special constant NO_COLOR to fill or stroke the polygon. It is not allowed to set both colors to NO_COLOR since this would result in an invisible annotation.

This annotation type has an associated PopUp annotation that displays the string *Content* in a floating window. The initial window state of the associated PopUp annotation is closed by default but the state can be changed with SetAnnotOpenState() if necessary.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

PolyLineAnnot

Syntax:

```
SI32 pdfPolyLineAnnot(
    const PPDF* IPDF,          // Instance pointer
    struct TFltPoint* Vertices, // At least two vertices must be provided
    UI32 NumVertices,          // Number of vertices in the array
    double LineWidth,          // Line width
    TLineEndStyle Start,       // Line end style of the start point
    TLineEndStyle End,         // Line end style of the end point
    UI32 FillColor,            // Fill color or NO_COLOR = transparent
    UI32 StrokeColor,          // Stroke color or NO_COLOR = transparent
    TPDFColorSpace CS,         // Color space of the color values
    const char* Author,        // Optional author
    const char* Subject,       // Optional subject
    const char* Content)       // Optional content or comment

typedef enum
{
    leNone,
    leButt,
    leCircle,
    leClosedArrow,
    leDiamond,
    leOpenArrow,
    leRClosedArrow,
    leROpenArrow,
    leSlash,
    leSquare
}TLineEndStyle;
```

The function creates a PolyLine Annotation. The vertices are connected by straight lines. At least two vertices must be provided.

The coordinates of the vertices are interpreted in current user space. Any transformation that was applied on the coordinate system will be taken into account.

The parameter *FillColor* is only used if the line end style of the start or end point has an interior that can be filled. The special constant NO_COLOR represents a transparent interior.

The stroke color is required and must not be set to NO_COLOR.

This annotation type has an associated PopUp annotation that displays the string *Content* in a floating window. The initial window state of the associated PopUp annotation is closed by default but the state can be changed with SetAnnotOpenState() if necessary.

The parameter *LineWidth* must be in the range 0 through 12 units. Values outside the valid range will be adjusted to the nearest allowed value. A zero line width produces a 1 pixel wide line.

The line end styles can be changed if necessary with SetAnnotLineEndStyle().

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

PrintPage

Syntax:

```
LBOOL pdfPrintPage(
    const PPDF* IPDF,      // Instance pointer
    UI32 PageNum,          // The page that should be printed
    const char* DocName,   // Optional document name (default value "out")
    const void* DC,        // Required, a printer device context
    TPDFPrintFlags Flags,  // Various flags (see PrintPDFFile() below)
    struct TRectL* Margin, // Optional additional margins (in Pixels)
    struct TPDFPrintParams* Parms); // See description in PrintPDFFile()
```

The function prints the page *PageNum* to the specified device context. The first page is denoted by one. At time of publication the function is implemented on Windows only and pages are printed as image.

The function parameters are described in detail at PrintPDFFile().

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

PrintPDFFile

Syntax:

```
LBOOL pdfPrintPDFFile(
    const PPDF* IPDF,      // Instance pointer
    const char* TmpDir,    // Ignored, defined for future use
    const char* DocName,   // Optional document name (default value "out")
    const void* DC,        // Required, a printer device context
    TPDFPrintFlags Flags,  // See below
    struct TRectL* Margin, // Optional additional margins (in Pixels)
    struct TPDFPrintParams* Parms) // Optional print parameters, see below
typedef enum
{
    pffDefault          = 0,   // Gray printing
    pff1Bit             = 1,   // Recommended for b&w printers
    pffColor            = 2,   // Color printing
    pffAutoRotateAndCenter = 4,   // Rotate and center pages if necessary
    pffPrintAsImage     = 8,   // Default on Windows
    pffShrinkToPrintArea = 16,  // Useful if a page contains no margins
    pffNoStartDoc       = 32,  // Do not call StartDoc on Windows
    pffNoStartPage      = 64,  // Do not call StartPage on Windows
```

```
    pffNoEndDoc             = 128, // Do not call EndDoc on Windows
    pffPrintPageAsIs        = 256, // Do not scale or rotate a page to fit into the printable area
    pffAutoRotate           = 512, // Rotate the page if necessary
    pffAutoCenter           = 1024 // Center the page
}TPDFPrintFlags;

struct TPDFPrintParams
{
    UI32                StructSize;    // Must be set to sizeof(TPDFPrintParams).
    LBOOL               Compress;      // Meaningful only for image output. If true, compressed images
                                       // are send to the printer.
    TCompressionFilter FilterGray;     // Meaningful only for image output. Supported filters on
                                       // Windows: cfFlate and cfJPEG.
    TCompressionFilter FilterColor;    // Meaningful only for image output. Supported filters on
                                       // Windows: cfFlate and cfJPEG.
    UI32                JPEGQuality;   // JPEG Quality in percent (1..100). Zero == Default == 60%.
    UI32                MaxRes;        // Maximum print resolution. Zero == Default == 600 DPI.
    LBOOL               IgnoreDCSize;  // If true, PageSize is used to calculate the output format.
    TIntRect            PageSize;      // Considered only, if IgnoreDCSize is set to true.
};
```

The function prints the PDF file that is currently in memory. At time of publication the function is implemented on Windows only and pages are printed as image.

The parameter *DocName* is optional but should be set if possible. The document name is set to "out" if no name is provided.

Output color format

It is possible to determine which color formats a printer supports as input, but it is not possible to determine whether a printer outputs black & white, gray shades, or color. The default should be set to gray or 1 bit but the user should be able to configure the color format.

Progress callback functions

The function calls the progress callback functions if set (see SetProgressProc() for further information). TInitProgressProc is called with *ProgType* set to ptPrintPage and *MaxCount* is set to the number of pages.

The progress callback function is called right before the next page is send to the printer (right before StartPage() is called). The progress callback function must return zero to continue processing. Any other return value breaks processing.

Print flag pffNoStartPage

If the flag pffNoStartPage is set, the function behaves as follows:

- The Windows function StartPage() must be called inside the progress callback function. The progress callback function is required if the flag pffNoStartPage ist set! The progress callback functions can be set with SetProgressProc().

- If *ActivePage* is **greater** 1, the Windows function EndPage() must be called.

- If the return value of the progress callback function should be non-zero to break processing, the function must still call EndPage() if *ActivePage* is greater 1.

- If the flag pffNoEndDoc is absent, the function calls EndPage() for the last page. If the flag pffNoEndDoc is set too, EndPage() is **not** called for the last page.

If you want to print a watermark or something similar on top of all pages, the flag pffNoEndDoc should be set too, otherwise it is not possible to print additional contents on the last page.

Maximum output resolution

The default maximum output resolution is 600 DPI to restrict the memory usage and to reduce the amount of data that must be send to the printer. The output resolution can be adjusted if necessary with printing parameters, see blow.

Print parameters

The optional parameter *Parms* can be used to adjust certain default print settings. The member *StructSize* must be set to sizeof(TPDFPrintParams).

If *Compress* is set to true, the function tries to send compressed images to the printer. The function checks first whether the printer supports compressed images. If compressed images are not supported then the function restarts the print job and sends uncompressed images to the printer (a warning is added to the error log in this case (see GetErrLogMessage() for further information)).

JPEG compression is recommended for color print jobs and PNG compression for gray or black & white output.

Note that many printer support JPEG compression but Flate compression is often not supported (PNG images). It is usually best to check with ExtEscape() whether PNG images are supported before this compression filter is set. If JPEG compression is used then the JPEG quality should not be lower than 60% for text documents to avoid visible compression artifacts.

Visual Basic 6 interface

Because VB6 has no direct support of optional structure parameters, the parameter Parms is defined as optional parameter of type long.

The structure TPDFPrintParams can be passed to the function as follows:

```
Dim r As TRectL, p As TPDFPrintParams
p.StructSize  = LenB(p)
p.Compress    = True
p.FilterGray  = cfFlate
p.FilterColor = cfFlate
p.MaxRes      = 150
Call pdf.PrintPDFFile(vbNullString, FileName, Printer.hDC, pffDefault, r, VarPtr(p))
```

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ReadImageFormat (obsolete)

Syntax:

```
LBOOL pdfReadImageFormat(
    const PPDF* IPDF,        // Instance pointer
    const char* FileName,    // File path to image
    UI32 ADDR Width,         // Width of the image in pixel
    UI32 ADDR Height,        // Height of the image in pixel
    SI32 ADDR BitsPerPixel,// Color depth
    SI32 ADDR UseZip)        // If true, Flate compression should be used
```

The function retrieves the most important properties of an image file. All parameters of the function must not be NULL. The function reads the image header only to improve processing speed. The parameter *UseZip* determines whether Flate compression produces better compression ratios for this image format. If the parameter *BitsPerPixel* is 32 the image is a CMYK image. Note that TIFF images support color depths up to 64 bits per pixel.

Remarks:

This function is marked as obsolete, please use ReadImageFormat2() whenever possible. ReadImageFormat2() will be renamed to ReadImageFormatFromFile() in DynaPDF 3.0.

Return values:

If the function succeeds the return value is 1 and the parameters are filled with values. If the function fails the return value is 0.

ReadImageFormat2

Syntax:

```
LBOOL pdfReadImageFormat2(
    const PPDF* IPDF,        // Instance pointer
    const char* FileName,    // File path to image
    UI32 Index,              // Image index of multi-page image
    UI32 ADDR Width,         // Width of the image in pixel
    UI32 ADDR Height,        // Height of the image in pixel
    SI32 ADDR BitsPerPixel,  // Color depth
    SI32 ADDR UseZip)        // If true, Flate compression should be used
```

The function retrieves the most important properties of an image file. All parameters of the function must not be NULL. The function reads only the image header to improve processing speed. The parameter *UseZip* determines whether Flate compression produces probably better compression results for this image format. If the parameter *BitsPerPixel* is 32, the image is a CMYK image. Note that TIFF images support color depths up to 64 bits per pixel. The parameter *Index* specifies the array index of a multi-page image that should be read in; numbering starts at 1. The parameter is ignored for non-multi-page image formats. Use GetImageCount() to determine the number of images in an image file.

1 bit images

1 bit images use sometimes different resolutions for the x- and y-axis. This function returns the logical size of the image taking the image resolution into account but note that this adjustment is done for 1 bit images only. All functions to insert an image consider this adjustment in the very same way. So, you can work with the image as if the resolution for the x- and y-axis would be identically.

However, sometimes we need the physical size of the image and this size can be calculated as follows:

```
SI32 bits, useZip;
UI32 resX = 0, resY = 0, physHeight, physWidth, w, h;

pdfReadImageFormat2(pdf, "test.tif", 1, w, h, bits, useZip);
pdfReadImageResolution(pdf, "test.tif", 1, resX, resY);
if (resX != resY && resX > 0 && resY > 0){
    if (resX > resY){
        physWidth  = w;
        physHeight = h / (resX / resY);
    }else{
        physWidth  = w / (resY / resX);
        physHeight = h;
    }
}else{
    physHeight = h;
    physWidth  = w;
}
```

Note that the above calculation uses integer arithmetic. It is meaningful for 1 bit images only.

Return values:

If the function succeeds the return value is 1 and the parameters are filled with values. If the function fails the return value is 0.

ReadImageFormatEx

Syntax:

```
LBOOL pdfReadImageFormatEx(
   const PPDF* IPDF,        // Instance pointer
   void* hBitmap,           // HBITMAP handle
   UI32 ADDR Width,         // Width of the image in pixel
   UI32 ADDR Height,        // Height of the image in pixel
   SI32 ADDR BitsPerPixel,  // Color depth
   SI32 ADDR UseZip)        // If true, Flate compression should be used
```

The function retrieves the most important properties of a memory bitmap. All parameters of the function must not be NULL. The function uses the GDI function GetObject() to retrieve the bitmap parameters. This function is available on Windows only. Please consider the special treatment of 1 bit images. See ReadImageFormat2() for further information.

Return values:

If the function succeeds the return value is 1 and the parameters are filled with values. If the function fails the return value is 0.

ReadImageFormatFromBuffer

Syntax:

```
LBOOL pdfReadImageFormatFromBuffer(
   const PPDF* IPDF,        // Instance pointer
   const void* Buffer,      // Pointer to image buffer
   UI32 BufSize,            // Buffer size
   UI32 Index,              // Image index of multi-page image
   UI32 ADDR Width,         // Width of the image in pixel
   UI32 ADDR Height,        // Height of the image in pixel
   SI32 ADDR BitsPerPixel,  // Color depth
   SI32 ADDR UseZip)        // If true, Flate compression should be used
```

The function retrieves the most important properties of an image in the same way as ReadImageFormat2(), but accepts an image buffer as input. See ReadImageFormat2() for further information.

Return values:

If the function succeeds the return value is 1 and the parameters are filled with values. If the function fails the return value is 0.

ReadImageResolution

Syntax:

```
LBOOL pdfReadImageResolution(
    const PPDF* IPDF,       // Instance pointer
    const char* FileName,   // File path to image
    UI32 Index,             // Image index of multi-page image
    UI32 ADDR ResX,         // Horizontal resolution
    UI32 ADDR ResY)         // Vertical resolution
```

The function retrieves the horizontal and vertical resolution of an image file. The image resolution is a user defined value that can be stored in certain image formats such as Bitmap or TIFF.

The function sets the variables *ResX* and *ResY* to zero if the image format does not support a resolution record or if no values are stored in the image file, the image resolution is then 72 DPI.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ReadImageResolutionEx

Syntax:

```
LBOOL pdfReadImageResolutionEx(
    const PPDF* IPDF,     // Instance pointer
    const void* Buffer,   // Pointer to image buffer
    UI32 BufSize,         // Buffer size
    UI32 Index,           // Image index of multi-page image
    UI32 ADDR ResX,       // Horizontal resolution
    UI32 ADDR ResY)       // Vertical resolution
```

The function retrieves the horizontal and vertical resolution of an image file in the same way as ReadImageResolution() but accepts an image buffer as input. The image resolution is a user defined value that can be stored in certain image formats such as Bitmap or TIFF.

The function sets the variables *ResX* and *ResY* to zero if the image format does not support a resolution record or if no values are stored in the image file, the image resolution is then 72 DPI.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Rectangle

```
LBOOL pdfRectangle(
   const PPDF* IPDF,        // Instance pointer
   double PosX,             // X-Coordinate of rectangle
   double PosY,             // Y-Coordinate of rectangle
   double Width,            // Width of the rectangle
   double Height,           // Height of the rectangle
   TPathFillMode FillMode)  // Fill mode
```

This function draws a rectangle.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the rectangle. If the coordinate system is top-down it defines the upper left corner.

The draw direction can be changed with the function SetDrawDirection().

A rectangle is a closed path that can be filled, stroked or both. It is also possible to draw a rectangle invisible to apply the filling rules nonzero winding number or even-odd. The filling rules are described under ClipPath(). The parameter *FillMode* is ignored if the rectangle is drawn inside a clipping path. The fill modes are described under ClosePath().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Redraw (Rendering Engine)

Syntax:

```
void rasRedraw(
   IRAS* RasPtr,     // Pointer of the rasterizer
   const void* DC,   // Handle of a Windows Device Context (HDC)
   SI32 DestX,       // Destination x-coordinate
   SI32 DestY)       // Destination y-coordinate
```

Because there is no direct access to the internal DIB Sesction when rendering into a device context, this function can be used to redraw the bitmap on the device context. If RenderPageEx() was called in a separate thread then this function can be called **after** the thread returned. While the thread is running you can change the position by directly manipulating the coordinate variables which were passed to RenderPageEx().

The function calls the system function SetDIBitsToDevice() to copy the image into the device context.

ReEncryptPDF

Syntax:

```
SI32 pdfReEncryptPDF(
    const PPDF* IPDF,          // Instance pointer
    const char* FileName,      // File to encrypt
    TPwdType PwdType,          // Kind of password see below
    const char* InPwd,         // Password to open the PDF file
    const char* NewOpenPwd,    // New open password (can be NULL)
    const char* NewOwnerPwd,   // New owner password (can be NULL)
    TKeyLen NewKeyLen,         // Key length for RC4 encryption
    TRestrictions Restrict)    // Restrictions
```

This function encrypts an already encrypted or unencrypted PDF file. The file is recompressed during import; this reduces the file size in most cases.

By using specific import flags (see SetImportFlags()), it is also possible to remove unwanted objects from the PDF file, such as annotations, form fields, bookmarks and so on.

If the input file contains a compressed object structure, it will be converted back to a normal PDF file.

The input PDF file will be replaced with the new one if no error occurred during import. If an error occurred the file is left unchanged.

It is not possible to change the file except removing specific objects by using specific import flags. However, the document info entries are changed to new values if they were set beforehand. See SetDocInfo() for further information.

The encryption flags are described in detail under CloseFileEx().

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Unicode version is supported by Windows NT systems only. The file path is converted back to Ansi if the Unicode version is used under Linux or UNIX.

Return values:

If the function succeeds the return value is 0. If the function fails the return value is a negative error code. The returned error code can be used to determine whether a password is required to open the file. Pass the return value to the macro PDF_WRONG_PWD(), if the macro returns true a password is required to open the file.

RenameSpotColor

Syntax:

```
SI32 pdfRenameSpotColor(
  const PPDF* IPDF,      // Instance pointer
  const char* Colorant, // UTF-8 colorant name to change
  const char* NewName)  // New name or NULL (see description)
```

The function renames the spot color *Colorant* to *NewName* in all Separation, DeviceN, and NChannel color spaces in which the colorant can be found. If the parameter *NewName* is set to NULL, the function renames the colorant to the special value ***None*** which produces no visible output.

The representation of the colorant in the alternate color space is left unchanged, also if the colorant is set to the special value ***None*** or ***All***. Note that the colorant names must be UTF-8 encoded Unicode strings. If the strings are preceded with a BOM (Byte Order Mark) then the BOM will be removed.

Return values:

The function returns the number of color spaces in which the colorant has been changed. If the function fails the return value is a negative error code.

RenderAnnotOrField

Syntax:

```
SI32 pdfRenderAnnotOrField(
  const PPDF* IPDF,              // Instance pointer.
  UI32 Handle,                   // Annotation or field handle.
  LBOOL IsAnnot,                 // If false, Handle must be a field handle.
  TButtonState State,            // The state that should be rendered.
  struct TCTM* Matrix,           // PageSpace of TPDFRasterImage structure.
  TRasterFlags Flags,            // As used with RenderPage().
  TPDFPixFormat PixFmt,          // The pixel format.
  TCompressionFilter Filter,     // Only used if CreateImage() was called
                                 // beforehand.
  struct TPDFBitmap* Out)        // Required, see below.

struct TPDFBitmap
{
    UI32  StructSize; // Must be set to sizeof(TPDFBitmap)
    BYTE* Buffer;     // Raw image buffer
    UI32  BufSize;    // Buffer size in bytes
    SI32  DestX;      // x-coordinate in the main image (the rendered page)
    SI32  DestY;      // y-coordinate in the main image (the rendered page)
    UI32  Height;     // Image height in pixels
    SI32  Stride;     // Scanline length in bytes
    UI32  Width;      // Image width
};
```

The function renders an annotation or field independent of the page. The page must be rendered with RenderPage() or RenderPageEx() beforehand since these functions calculate also the required mapping from user space to device space. The member *PageSpace* of the TPDFRasterImage structure represents this mapping and must be passed to the parameter *Matrix*.

The function must be called within an open page and with the same *Flags* which were used in the RenderPage() or RenderPageEx() call. The flags to exclude annotations or form fields from rendering will be ignored.

To render the annotations and fields from a page, open the wished page with EditPage() and call GetPageAnnotCount(). Now you can loop over the annotations from 0 to GetPageAnnotCount() - 1 with GetPageAnnotEx(). A form field is an annotation of the type atWidget.

The member *StructSize* of the structure TPDFBitmap must be set to sizeof(TPDFBitmap) before the function can be called.

The members *DestX*, *DestY* define the destination position on the rendered page image. The position is returned in top down coordinates and measured in pixels. Note that annotations and form fields can occur outside the visible area of a page.

If the coordinates of the page were translated, this is the case if the x or y members of the matrix were non-zero when rendering the page, then this offset must be added to the returned destination coordinates:

```
out.DestX += (SI32)img.Matrix.x;
out.DestY += (SI32)img.Matrix.y;
```

Since the matrix is already defined in device coordinates, no further math is required to calculate the correct destination position.

Note that annotations and form fields can contain transparent areas. The pixel format should contain an alpha channel so that it can be correctly blended on the page image.

The Button State (parameter State)

Only form fields support different appearance states. Annotations use always the up state. If no appearance stream is defined for a specific state then the normal appearance will be returned. Check boxes contain mostly four different appearance streams: two for the up state and two for the down state. The function selects the appearance stream for a specific state depending on the field value. Therefore, all four appearances can only be rendered, if the field value will be changed from checked to unchecked or vice versa.

Raw image output

By default, annotations or form fields are rendered into a raw memory buffer that is returned in the TPDFBitmap structure. The image buffer can be released with FreeImageBuffer() when it is no longer needed. It is not required to release the image buffer every time before the next annotation or field can be rendered but previously returned pointers become invalid once the next image was rendered. After the last annotation was rendered, the image buffer should be released with FreeImageBuffer().

Note that the return value of the function must be zero, otherwise no image will be returned. See Return values at the end of the function description.

How to create image files

It is also possible to create image files from rendered annotations or fields. To achieve this, create a new image with CreateImage() beforehand and call RenderAnnotOrField() afterwards. The rendered image is passed to AddRasImage() internally. If the output format is TIFF, more than one image can be added to the output image. The parameter *Filter* is only used if CreateImage() was called beforehand. The output image must finally be closed with CloseImage(). If the image was created in memory call GetImageBuffer() as usual.

Example (C++):

```cpp
void RenderPDFPage(const PPDF* PDF, const char* InFile, UI32 PageNum, const char* OutDir)
{
    UI32 w, h;
    TPDFBitmap out;
    char path[MAX_PATH]; TPDFRasterImage img; TPDFAnnotationEx annot;

    // We do not create a PDF file in this example
    pdfCreateNewPDF(PDF, NULL);

    pdfSetImportFlags(PDF, ifImportAll | ifImportAsPage);
    pdfSetImportFlags2(PDF, if2UseProxy);

    if ((retval = pdfOpenImportFile(PDF, InFile, ptOpen, NULL)) < 0) return retval;
    pdfAppend(PDF);
        pdfImportPageEx(PDF, PageNum);
    pdfEndPage(PDF);

    memset(&img, 0, sizeof(img));
    img.StructSize = sizeof(img);
    img.DefScale   = psFitBest;
    img.Flags      = TRasterFlags(rfInitBlack | rfExclFormFields | rfExclAnnotations);
    img.InitWhite  = true;
    img.Matrix.a   = 1.0;
    img.Matrix.d   = 1.0;

    IPGE* page = pdfGetPageObject(PDF, 1);
    rasCalcPagePixelSize(page, psFitBest, 1.0, 2500, 2500, rfDefault, &w, &h);

    BYTE* buffer = (BYTE*)malloc(w * 4 * h);

        IRAS* ras = rasCreateRasterizer(PDF, NULL, buffer, w, h, w * 4, pxfRGBA);

            pdfRenderPage(PDF, page, ras, &img);

            sprintf(path, "%s/page.png", OutDir);
            pdfCreateImage(PDF, path, ifmPNG);
                pdfAddRasImage(PDF, ras, cfFlate);
            pdfCloseImage(PDF);

        rasDeleteRasterizer(&ras);

    free(buffer);

    pdfEditPage(PDF, 1);
        SI32 i, count = pdfGetPageAnnotCount(PDF);
        out.StructSize = sizeof(out);

        for (i = 0; i < count; i++)
        {
            pdfGetPageAnnotEx(PDF, i, annot);

            sprintf(path, "%s/annot_%2d.png", OutDir, i);
            // We store the result as PNG image in this example
```

```
            pdfCreateImage(PDF, path, ifmPNG);
            pdfRenderAnnotOrField(  PDF,
                                    annot.Handle,
                                    annot.Type != atWidget,
                                    bsUp,
                                    &img.PageSpace, // RenderPage() calculated the matrix
                                    rfDefault,
                                    pxfRGBA,
                                    cfFlate,
                                    &out);
            pdfCloseImage(PDF);
        }
    pdfEndPage(PDF);

    pdfFreePDF(PDF); // Release the PDF in memory
}
```

Return values:

- 0   The annotation or field was successfully rendered.
- 1   The annotation is hidden. No image was returned. In order to render hidden annotations remove first the hidden flag with SetAnnotFlags() or SetFieldFlags().
- 2   The annotation or field contains no appearance stream. No image was returned.
- < 0   An error occurred. No image was returned.

RenderPage (Rendering Engine)

Syntax:

```
LBOOL pdfRenderPage(
    const PPDF* IPDF,           // Instance pointer
    IPGE* PagePtr,              // Page pointer
    IRAS* RasPtr,               // Pointer of the rasterizer
    struct TPDFRasterImage* Img) // See below

struct TPDFRasterImage
{
    UI32            StructSize;     // Must be set to sizeof(TPDFRasterImage)
    TRasterFlags    Flags;          // See below
    TPDFPageScale   DefScale;       // Specifies how the page should be scaled.

    LBOOL           InitWhite;      // If true, the image buffer is initialized to white before
                                    // rendering. When a clipping rectangle is set, only the area
                                    // inside the clipping rectangle is initialized to white.

    struct TIntRect ClipRect;       // Optional clipping rectangle defined in device coordinates
                                    // (Pixels), default 0,0,0,0 (no clipping)
    struct TCTM     Matrix;         // Optional transformation matrix. Initialize the variable to
                                    // the identity matrix (1,0,0,1,0,0) if you don't need it. The
                                    // matrix can be used to move and scale the page.
    struct TCTM     PageSpace;      // Out -> The matrix represents the mapping from page space to
                                    // device space. This matrix is required when further objects
                                    // should be drawn on the page, e.g. form fields or annotations
                                    // if rendered separately.

    LBOOL           DrawFrameRect;  // If true, the area outside the page's bounding box is filled
                                    // with the frame color. InitWhite can still be used, with or
                                    // without a clipping rectangle.
    UI32            FrameColor;     // Must be defined in the color space of the pixel format but in
                                    // the natural component order, e.g. RGB.

    TOnUpdateWindow* OnUpdateWindow; // Optional, UpdateOnPathCount and UpdateOnImageCoverage define
                                    // when the function should be called
```

```
    void*           OnInitDecoder;   // Not yet defined
    void*           OnDecodeLine;    // Not yet defined
    const void*     UserData;        // A pointer that should be passed to the callback function

    UI32            UpdateOnPathCount;
    // Optional -> Call OnUpdateWindow when the limit was reached. Clipping paths increment the
    // number too. Only full paths are considered, independent of the number of vertices they
    // contain. The value should be between 3000 and 10.000. Values between 3000 and 5000 produce
    // best results. DynaPDF fires an update event when either 100.000 vertices were rendered or when
    // the UpdateOnPathCount limit was reached, whatever occurs first.

    float           UpdateOnImageCoverage;
    // Optional -> UpdateOnImageCoverage specifies a factor that is multiplied with a master area
    // of 1,000,000 pixels. So, when the factor is set to 1.0 then the update event is fired when
    // 1,000,000 unscaled source pixels were processed.

    // Statistics...
    UI32            NumAnnots;        // Out
    UI32            NumBezierCurves;  // Out
    UI32            NumClipPaths;     // Out
    UI32            NumFormFields;    // Out
    UI32            NumGlyphs;        // Out
    UI32            NumImages;        // Out
    UI32            NumLineTo;        // Out
    UI32            NumPaths;         // Out
    UI32            NumPatterns;      // Out
    UI32            NumRectangles;    // Out
    UI32            NumRestoreGState; // Out
    UI32            NumSaveGState;    // Out
    UI32            NumShadings;      // Out
    UI32            NumSoftMasks;     // Out
    UI32            NumTextRecords;   // Out
};

typedef enum
{
    psFitWidth,  // Scale the page to the width of the image
    psFitHeight, // Scale the page to the height of the image
    psFitBest,   // Scale the page so that it fits fully into the image
    psFitZoom    // Should be used when zooming into a page to avoid
                 // adjustments on the transformation matrix
}TPDFPageScale;

typedef enum
{
    rfDefault         = 0x00000000,
    rfScaleToMediaBox = 0x00000001,
    rfIgnoreCropBox   = 0x00000002,
/*
    The art, bleed, or trim box is first intersected with the media box
    or crop box if present since these boxes represent the maximum extend
    of the page. If the flag rfIgnoreCropBox is set, the boxes are
    intersected with the media box. The page is scaled to the media or crop
    box depending on the above flags. By default a page is scaled to the
    crop box if present.
*/
    rtClipToArtBox    = 0x00000004,
    rtClipToBleedBox  = 0x00000008,
    rtClipToTrimBox   = 0x00000010,
    rtExclAnnotations = 0x00000020,
    rtExclFormFields  = 0x00000040,
    rfSkipUpdateBG    = 0x00000080, // Skip the update event after initializing the background
    rfRotate90        = 0x00000100, // Rotate the page by 90 degress
    rfRotate180       = 0x00000200, // Rotate the page by 180 degress
    rfRotate270       = 0x00000400, // Rotate the page by 270 degress
    rfInitBlack       = 0x00000800, // Initialize the image buffer to black
                                    // before rendering. This flag is not
                                    // meaningful if the pixel format has
```

```
                                  // no alpha channel.
      rfCompositeWhite  = 0x00001000, // Composite the image with a white
                                      // background after rendering. This
                                      // flag is meaningful only for pixel
                                      // formats with an alpha channel.
      rfExclPageContent = 0x00002000, // If set, only annotations and form fields will be rendered
      rfExclButtons     = 0x00004000, // Exclude push buttons
      rfExclCheckBoxes  = 0x00008000, // Exclude check boxes
      rfExclComboBoxes  = 0x00010000, // Exclude combo boxes
      rfExclListBoxes   = 0x00020000, // Exclude list boxes
      rfExclTextFields  = 0x00040000, // Exclude text fields
      rfExclSigFields   = 0x00080000, // Exclude signature fields
      // ------------------------------
      rfScaleToBBox     = 0x00100000, // Meaningful only, if rfClipToArtBox, rfClipToBleedBox, or
                                      // rfClipToTrimBox is set. If set, the picture size is set to the
                                      // size of the wished bounding box.

      rfDisableAAClipping    = 0x00200000, // Disable Anti-Aliasing for clipping paths. This flag is
                                           // the most important one since clipping paths cause often
                                           // visible artefacts in PDF files with flattened
                                           // transparency.
      rfDisableAAText        = 0x00400000, // Disable Anti-Aliasing for text.
      rfDisableAAVector      = 0x00800000, // Disable Anti-Aliasing for vector graphics.
      // Fully disable Anti-Aliasing.
      rfDisableAntiAliasing  = rfDisableAAClipping | rfDisableAAText | rfDisableAAVector,
      rfDisableBiLinearFilter = 0x01000000, // Disable the BiLevel filter for images. Sometimes useful
                                            // if sharp images are needed, e.g. for barcodes.
      rfClipBoxMask          = rfClipToArtBox | rfClipToBleedBox | rfClipToTrimBox // Internal flags
      rfRenderInvisibleText  = 0x02000000, // If set, treat text rendering mode Invisible as Normal.
      rfEnableBlendCS        = 0x10000000  // If set, the page is rendered in the color space of the
                                           // page group, soft proof color space, or output intent (if
                                           // set), and finally converted to the destination color
                                           // space.
      rfRenderPrintState     = 0x20000000, // If set, the print state of layers, annotations, and form
                                           // fields will be rendered.
      rfForceInterpolation   = 0x40000000  // If set, image interpolation will be applied.
}TRasterFlags;
```

```
typedef SI32 PDF_CALL TOnUpdateWindow(const void* Data, TIntRect Area);
```

Special flags (see description at "The OnUpdateWindow Event"):

```
#define SKIP_INIT_SCREEN   0x10000000 // Do not initialize the screen before rendering begins
#define SKIP_UPDATE_SCREEN 0x20000000 // Do not finally update the screen after the page was rendered
```

The function renders a PDF page into an image buffer. This function was mainly designed to render PDF pages in a viewer application but it can also be used for other purposes where fast rendering into an image buffer is required. It is also possible to render pages into a device context, see RenderPageEx() for further information.

Before rendering the first page the application should set the path from which external CMaps can be loaded (see SetCMapDir()). Set the path with the flag lcmDelayed so that the files will only be loaded if necessary.

Rendering speed can be improved by enabling the image cache. See EnableImageCache() for further information.

It is also strongly recommended to initialize color management before rendering the first page (see InitColorManagement()) and to set the screen resolution (see SetScreenRes()). The output intent (if any) is loaded automatically before the page will be rendered if color management is enabled.

Minimal initialization

A few members of the the structure *TPDFRasterImage* must be correctly initialized, otherwise nothing can be rendered:

```
TPDFRasterImage img;
memset(&img, 0, sizeof(img));
img.StructSize = sizeof(img);
img.Matrix.a   = 1.0; // Identity matrix
img.Matrix.d   = 1.0; // Identity matrix (1.0, 0.0, 0.0, 1.0, 0.0, 0.0)
```

The structure size is used to identify different versions of the structure. The a and d members of the transformation matrix represent the scaling factors. If these factors are set to zero, then nothing can be rendered! The matrix should be set to the identity matrix, unless *DefScale* is set to psFitZoom.

The members *Flags* and *DefScale* should be set to appropriate values too. Note that the structure contains also many output fields. Especially the member *PageSpace* is sometimes important since this matrix represents the mapping from user space to device space.

Basics

The function requires three pointers: the instance pointer of the PDF instance, a pointer of a rasterizer object (see CreateRasterizer()), and finally a pointer of the page object that should be rendered (returned by GetPageObject()). All three pointers must be valid and and not NULL.

Before a page can be rendered the caller must create a rasterizer with CreateRasterizer(). The rasterizer is normally created when the application is initialized and deleted when the application is terminated.

The size of the image buffer can be changed at runtime, e.g. when the window size must be changed, without creating a new rasterizer. Simply attach the new buffer with AttachImageBuffer().

The pixel format cannot be changed at runtime but it is possible to use multiple rasterizers if necessary.

The rasterizer and the rendering functions use the error log of the PDF instance to output errors and warnings. An error that occurs during rendering does normally not affect the PDF file in memory. This is also the case for fatal errors. The rendering engine can be seen as a separate modul that is not allowed to interact directly with the PDF instance from which pages were rendered.

However, it is not always possible to isolate errors during rendering from the PDF instance. For example, when a font resource must be decompressed and if an out of memory exception occurs during decompression, then this will also cause an out of memory exception when trying to finish the file, e.g. with CloseFile() or similar functions.

Keep in mind that the rasterizer contains a reference of the error log of the PDF instance. Therefore, it is not allowed to delete the PDF instance without deleting the rasterizer. The rasterizer should also be deleted before the PDF instance.

It is allowed to use one rasterizer with different PDF instances but note that errors will be reported into the error log of the PDF instance that is associated with the rasterizer. To avoid multi-threading

issues it is usually best to create a separate rasterizer for each PDF instance that is used to render PDF pages.

Pixel Formats

PDF uses a transparent imaging model, that means many objects depend on the availability of an alpha channel. To achieve optimal results the image buffer should contain an alpha channel. If the pixel format contains an alpha channel then DynaPDF produces an image with pre-multiplied alpha since this is what most operating systems expect.

The image buffer must be initialized with zero or black if the pixel format contains an alpha channel. It is also possible to set *InitWhite* to true and to set the flag rfInitBlack so that an image buffer with an alpha channel can be correctly initialized. The flag rfInitBlack is only considered if the pixel format contains an alpha channel. If the flag rfInitBlack is absent then the image buffer will be initialized to white but the alpha channel is set to zero for each pixel.

The internal blend functions consider this initialization so that blend functions produce still correct results. This kind of initialization can be useful if the image should be copied into the video buffer with a function that doesn't support alpha blending, e.g. SetDIBitsToDevice(). However, to achieve correct results the flag rfCompositeWhite must be set too. This flag makes sure that the image will be composited with a white background after the page was fully rendered. The result is a fully opaque image. The advantage is that it is not required to clear the background with white before the image can be drawn. This technique is ued by the page cache and by the viewer examples if the pixel format is set to pxfBGRA.

An opaque image buffer must be initialized with 255 or fully white. You can also set *InitWhite* to true to make sure that the image buffer will be correctly initialized before rendering.

Rendering into an opaque image buffer produces mostly the same result but not always. Differences can occur when a blend function is used. Since the background is opaque, anti-alisasing artifacts can occur since anti-aliased pixels are rendered toward white and not to fully transparent as it would be the case when the image contains an alpha channel.

The difference can be seen in the pictures below. The three circles were drawn with the blend mode bmExclusion. The RGBA version contains no anti-aliasing artifacts but the RGB version contains an ugly border where the circles overlap.

RGBA   RGB
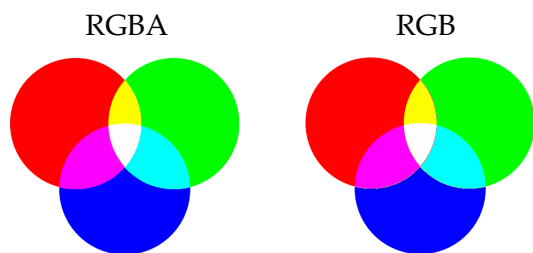
Differences can also occur when compositing a transparency group with the background. This occurs seldom but differences are possible.

Blending Color Spaces

Pages and isolated transparency groups can specify the blending color space in which the group must be rendered to achieve correct results. This is important because blend modes produce completely different results for additive and subtractive color spaces.

Blending color spaces can be enabled with the flag *rfEnableBlendCS*. They are not enabled by default because much more memory and processing time is required to render a page in this way since the page must first be rendered in the blending color space and finally converted back to the destination color space.

Rendering PDF Pages

Most members of the structure TPDFRasterImage are meaningful when pages are rendered in a viewer application. For other purposes most members can be set to their defaults.

RenderPage() renders a PDF page according to the page's orientation and bounding boxes. When a crop box is present the page is scaled and clipped into the crop box. When no crop box is present the media box is used instead. This is the default behaviour in viewer applications.

DefScale specifies how the page should be scaled into the image buffer. If the width and height of the image buffer corresponds with the scaled width and height of the page, psFitWidth, psFitHeight, and psFitBest produce all the same result (see enum TPDFPageScale). If the width or height are different, the page is scaled into the image according to *DefScale*.

The mode psFitZoom should be used when zooming into a page since it avoids adjustments on the transformation matrix.

When a page is scaled into the destination image it is usually best to draw the page into a frame rectangle. The frame rectangle represents the area outside the page's bounding box but inside the optional clipping rectangle.

Example:
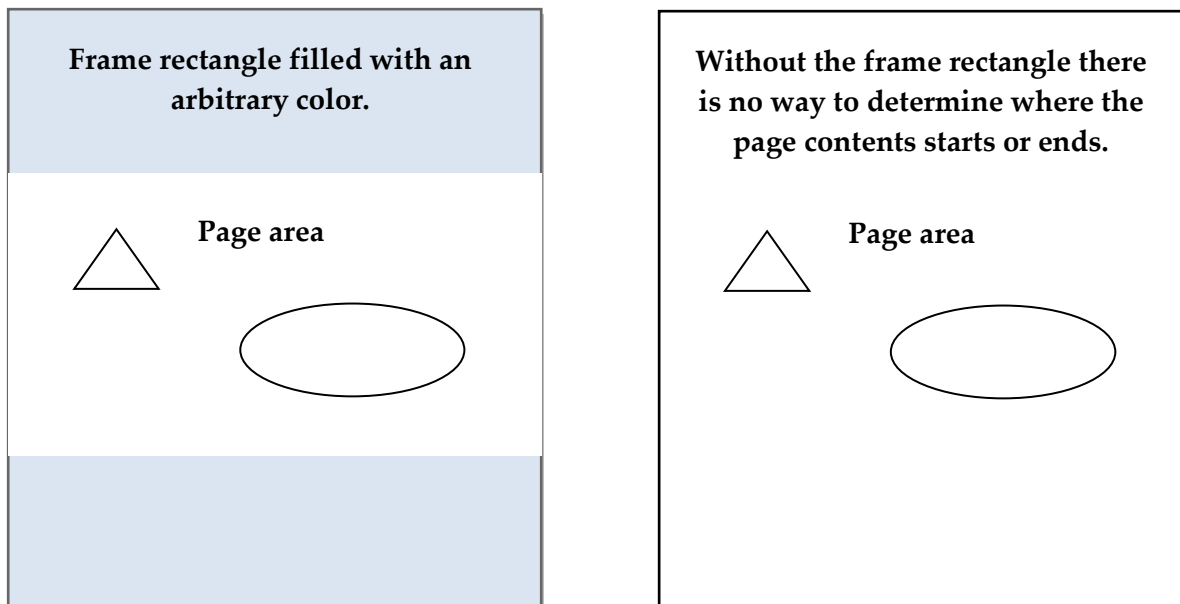
Whether a frame rectangle should be drawn or not depends on the way how a page is rendered into the destination window. If the viewer processes mainly simple PDF files then it is possible to render pages simply on demand. The page can then directly be scaled into the destination image and in this case it is usually best to draw the page into a frame rectangle.

However, scrolling and zooming becomes too slow when more complex pages must be rendered. To achieve optimal processing speed as well as smoth and flicker free scrolling the viewer should render the page into an image in the size of the scaled page format so that the finish image can be scrolled. The background or the frame rectangle must be drawn by the viewer application in this case.

To calculate the required image size in pixels the rendering engine provides helper functions to access the page's bounding boxes as well as the page orientation fast as possible (see GetPageBBox() and GetPageOrientation()).

The image size can be calculated as follows (C++):

```
...
UI32 w = 0, h = 0;
UI32 fw = 2540, fh = 1440; // Size of output rectangle in pixels
// Get a pointer of the page object to enable fast access to the page
// properties.
IPGE* pagePtr = pdfGetPageObject(m_PDF, pageNum);
// CalcPagePixelSize() returns the height of the image in this example.
// RenderPage() calculates the image size in the exact same way when
// called with the same parameters.
rasCalcPagePixelSize(pagePtr,psFitWidth,1.0f,fw,fh,rfDefault,&w,&h);
```

```
// Now you can allocate the image buffer or check whether the existing
// one is large enough.
if (w != lastWidth || h != lastHeight)
{
    ...
}
...
```

Calculating the size of the image buffer is easy, and scrolling is easy too when the zoom factor is 100%. The handling of other zoom factors is the most complex part in a viewer application. For zoom factors between 10% and 200% it is usually best to render the entire page as described above but larger zoom factors cannot be handled in this way because at some point the image becomes too large and the rendering time would become too long.

For larger zoom factors it is mostly better to render the page directly with the required zoom factor into the client rectangle since the rasterizer can then skip invisible objects to speed up processing.

The Transformation Matrix

The transformation matrix can be used to scroll the page up and down, and to scale it with an arbitrary zoom factor, or to rotate it if necessary. The matrix must be defined in device space, that is the coordinate space of the image buffer. This makes it very easy to scroll and zoom into the page since you don't need to consider the coordinate space of the PDF page.

The coordinate space of the rendering engine is top down and not bottom up like the native PDF coordinate space. So, negative y-coordinates move the PDF image to the top direction of the window and positive values move it to the opposite direction.

Example:

```
// This matrix scrolls the PDF image 120 pixels down
img.Matrix.a = 1.0;
img.Matrix.b = 0.0;
img.Matrix.c = 0.0;
img.Matrix.d = 1.0;
img.Matrix.x = 0.0;
img.Matrix.y = -120.0;
```

Zooming into the page is very easy too. Simply set the zoom factor to the a and d coefficients of the transformation matrix:

```
img.Matrix.a = m_Zoom;
img.Matrix.b = 0.0;
img.Matrix.c = 0.0;
img.Matrix.d = m_Zoom;
img.Matrix.x = m_x;
img.Matrix.y = m_y;
```

The zoom factor should take the page format into account and not simply multiply the device coordinates with a specific factor. The zoom factor must be restricted so that no number overflow occurs. A suitable limit is around 32 or 64 (6400%).

The OnUpdateWindow Event

When rendering complex PDF pages in a viewer it is strongly recommended to update the window from time to time with the portion of contents that was already rendered. To achieve this, the structure TPDFRasterImage supports two variables which specify when the window should be updated as well as a corresponding callback function that is executed if one of the limits were reached.

Please note that when the TOnUpdateWindow callback function is set, it is not required to draw the image into the window after RenderPage() was called!

When the callback function is set, RenderPage() fires at least one update event that covers the entire imageble area. In the best case, the page was already fully rendered at this point. Calling SetDIBitsToDevice() or similar system functions after RenderPage() is not required and causes just unnecessary overhead!

This behaviour can be adjusted if necessary with the flags *SKIP_INI_SCREEN*, and *SKIP_UPDATE_SCREEN*. These flags can be combined with the member *StructSize* of the TPDFRasterImage structur as follows:

```
TPDFRasterImage img;
img.StructSize = sizeof(img) | SKIP_INIT_SCREEN | SKIP_UPDATE_SCREEN.
```

 As the name suggessts the flag *SKIP_INIT_SCREEN* omits the initial screen initialisation before rendering starts. *SKIP_UPDATE_SCREEN* disables the final update screen event after the page was fully rendered.

The update area

The callback function returns the area that must be updated. It is strongly recommended to update only this area. This can drastically speed up processing and the application can process more update events with less overhead. The first call covers the entire imageble area when *InitWhite* or *DrawFrameRect* was set to true. Further calls occur only when the page was not already fully rendered at this point.

UpdateOnPathCount limit

The member *UpdateOnPathCount* specifies the number of paths which must be drawn before the callback function should be executed. The value should not be too small to avoid too many update events. When the value is too large then too few update events occur and you cannot see how the page is drawn.

The test containers use a limit of 5000. This is a good compromise between speed and the number of update events on faster computer systems. DynaPDF counts also the number of vertices which were

already rendered. The update event is fired when either the *UpdateOnPathCount* limit was reached or when more than 100.000 vertices were rendered, whatever occurs first. This makes sure that the window becomes updated from time to time when very large but few paths were rendered.

UpdateOnImageCoverage limit

The member *UpdateOnImageCoverage* specifies a factor that is multiplied with a master area of 1,000,000 pixels. So, when the factor is set to 1.0 then the update event is fired when 1,000,000 unscaled source pixels were processed. This calculation makes sure that small images cause not too many update events and the coverage area doesn't depend on the size of the destination image that is rendered. The bit depth of images is also taken into account since rendering 1 bit or gray scale images is much faster than rendering RGB or CMYK images.

In addition to the coverage limit DynaPDF counts also the number of images which were already rendered. The update event is fired when either the coverage limit was reached or when more than 3000 images were rendered, whatever occurs first. This makes sure that the window becomes updated from time to time when many small images were rendered.

The image scaler cannot produce an update event during scaling at this time. The event occurs always after an image was fully scaled. This behaviour will be changed as soon as possible…

The return value

The TOnUpdateWindow callback function can break processing if necessary. The return value must be zero to continue processing. Negative values break processing. However, the return value must be either 0 or -1. Other return values are not permitted and can cause unwanted side effects. The TOnUpdateWindow callback function is not the only way to break processing. When the page is rendered in a separate thread then use the function Abort() to break processing. See next section.

Multi-Threading strategies

PDF pages can be very large and complex; hence, it is not always possible to render pages in just a few milliseconds. When the rendering engine is used in a viewer application then it is usually best to render pages in a separate thread so that the main thread becomes not blocked. A good viewer implementation must also be able to stop rendering as fast as possible whenever necessary, e.g. when the user requests to change the view area, zoom factor, page position or other things.

To achieve this, the rendering engine provides the function Abort() (rasAbort() in C/C++). This function works of course only if RenderPage() is executed asynchronously. The one and only stable and reasonable way to do this is to execute the function RenderPage() in a separate thread.

The development of a stable multi-threaded viewer requires a few design considerations to avoid unnecessary multi-threading issues. Before we can render a PDF page we must usually import a page or create it with DynaPDF functions.

One thing that you must consider is that it is generally **not** allowed to import a page in a separate thread. However, importing a PDF page is very fast and there is generally no need to do so. Future

versions will also support specific flags to further improve the access time to external pages. So, don't load PDF pages in a separate thread!

Notice:

> DynaPDF is thread-safe but the condition is that every thread uses its own PDF instance and anything that should be done must then be done in this thread. When we render pages in a viewer then we must work with one instance in different threads. This tiny difference is very important since it is a huge difference whether a library must protect all functions from competing access or whether it must only isolate its data from one instance to another.

A viewer should only load the page from an external PDF file that should now be rendered. So, don't call a function like ImportPDFFile() to import the entire file when it is opened. Instead, load pages on demand as follows:

- When the user requests to open a PDF file then open it with OpenImportFile().
- Call GetInPageCount() to determine the number of pages in it.
- Create an array of pointers that holds the pointers of pages which were already loaded. Initialize the array with NULL so that it can be used to perform a duplicate check.
- Initialize the scroll bars, zoom factor and so on, and create the image buffer so that the first page can be displayed.
- When a page should be displayed then import it first in the main thread if necessary as follows:

```
if (m_Pages[m_PageNum-1] == NULL)
{
   // Important: Use EditPage() and not Append() to keep the pages
   // sorted! Holes are filled with empty pages by EditPage().
   pdfEditPage(m_PDF, m_PageNum);
   pdfImportPageEx(m_PDF, PageNum, 1.0, 1.0);
   pdfEndPage(m_PDF);
   // Store the page pointer in the duplicate array
   m_Pages[m_PageNum-1] = pdfGetPageObject(m_PDF, m_PageNum);
}
```

- Inititialize the structure TPDFRastertImage and create a new thread in the priority lower or normal.
- Now you can start the thread. The thread executes only the function RenderPage(), finish! The window must be updated via the TOnUpdateWindow callback function.
- When the user requests to load another page or PDF file then call rasAbort(m_RasPtr); from the main thread (m_RasPtr represents the instance pointer of the rasterizer), wait until the thread returns and delete it.
- Now you can load another file, page, or change the zoom settings and so on.

Make sure that you don't execute non-thread-safe code in the TOnUpdateWindow callback function or in the error callback function.

Especially the error callback function should not directly add the error message to a list component or something similar when the component is not thread-safe. Note that such components try to

render the text when it is added to the list. Such an action is critical when performed from another thread. It is mostly better to copy the message into an array and to add it to the list component when the error form is displayed. Such details can avoid a lot of possible multi-threading issues which are very often difficult to reproduce.

In the previous description there is only one thread running at a time. The current implementation of RenderPage() does definitely **not** allow the usage in multiple threads simultaneously. If possible this limitation will be removed in future versions but due to the many possible collusions it is currently not sure whether this makes sense. There are also not many situations in which it would be useful to render pages in background since you don't know what the user wants to do next. In most cases additional threads waste processing time and memory and the application becomes often slower and not faster.

How to save the image on disk?

DynaPDF contains a few helper functions to store the rendered image in a proprietary image format when necessary. These functions are CreateImage(), AddRasImage(), CloseImage(), and GetImageBuffer(). These functions support the creation of single- and multi-page images, as well as in-memory or file output.

The image buffer should be created in the correct pixel format because AddRasImage() performs no color conversion. For example, Windows Bitmaps support the pixel formats BGR and BGRA but no RGB or RGBA. All other image formats require pixels in the natural component order, e.g. RGB or RGBA.

AddRasImage() accepts also image buffers in a wrong component order but the caller is responsible to convert the buffer if necessary before calling the function.

The entire PDF file can be converted to an arbitrary image format with RenderPDFFile().

Remarks:

The function outputs errors and warnings into the error log of the corresponding PDF instance. The caller should check for errors by calling GetErrLogMessageCount() when the function returns. The error log can be cleared when the messages were processed with ClearErrorLog().

Return values:

When the page was fully rendered the return value is 1. When the page was not fully processed, e.g. due to errors, the return value is 0.

Example (C#):

```
int PDFError(IntPtr Data, int ErrCode, IntPtr ErrMessage, int ErrType)
{
    MessageBox.Show(System.Runtime.InteropServices.Marshal.PtrToStringAnsi(ErrMessage));
    return 0;
}

private void Form1_Paint(object sender, PaintEventArgs e)
{
    m_PDF.SetOnErrorProc(IntPtr.Zero, new TErrorProc(PDFError));
```

```csharp
    // We don't create a PDF file in this example
    m_PDF.CreateNewPDF(null);

    m_PDF.SetPageCoords(TPageCoord.pcTopDown);
    m_PDF.Append();
    m_PDF.SetFont("Arial", TFStyle.fsRegular, 20.0, false, TCodepage.cpUnicode);
    m_PDF.WriteFTextEx(  50.0,
                         50.0,
                         495.0,
                         -1.0,
                         TTextAlign.taCenter,
                         "A small example that shows how RenderPage() can be used...");
    m_PDF.EndPage();

    // Get the page object
    IntPtr pagePtr = m_PDF.GetPageObject(1);

    int w = pictureBox1.Width;   // The picture box was placed on the form.
    int h = pictureBox1.Height;


    // Calculate the image size
    m_PDF.CalcPagePixelSize( pagePtr,
                             TPDFPageScale.psFitBest,
                             1.0f,
                             w,
                             h,
                             TRasterFlags.rfDefault,
                             ref w,
                             ref h);

    // Create a bitmap in this size
    Bitmap bmp = new Bitmap(w, h, System.Drawing.Imaging.PixelFormat.Format32bppArgb);
    System.Drawing.Imaging.BitmapData bd = bmp.LockBits(
                                               new Rectangle(0, 0, w, h),
                                               System.Drawing.Imaging.ImageLockMode.WriteOnly,
                                               System.Drawing.Imaging.PixelFormat.Format32bppRgb);

    // Create a rasterizer for the bitmap
    IntPtr ras = m_PDF.CreateRasterizer( IntPtr.Zero,
                                         bd.Scan0,
                                         w,
                                         h,
                                         bd.Stride,
                                         DynaPDF.TPDFPixFormat.pxfBGRA);

    TPDFRasterImage img = new DynaPDF.TPDFRasterImage();
    img.StructSize   = System.Runtime.InteropServices.Marshal.SizeOf(img);
    img.InitWhite    = 1;
    img.DefScale     = TPDFPageScale.psFitBest;
    // A 32 bit image has a transparent background. The flag rfCompositeWhite makes sure that the
    // image gets pre-blended with a white background.
    img.Flags        = TRasterFlags.rfInitBlack | TRasterFlags.rfCompositeWhite;
    img.Matrix.a     = 1.0; // Identity matrix
    img.Matrix.d     = 1.0; // Identity matrix

    // Render the page
    m_PDF.RenderPage(pagePtr, ras, ref img);

    bmp.UnlockBits(bd);
    pictureBox1.Image    = bmp;
    pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage;

    m_PDF.DeleteRasterizer(ref ras);
    m_PDF.FreePDF();
}
```

RenderPageEx (Rendering Engine)

Syntax:

```
LBOOL pdfRenderPageEx(
   const PPDF* IPDF,      // PDF Instance pointer
   const void* DC,        // Handle of a device context (HDC)
   SI32* DestX,           // Destination x-coordinate to blend the image
   SI32* DestY,           // Destination y-coordinate to blend the image
   IPGE* PagePtr,         // Page pointer
   IRAS* RasPtr,          // Pointer oft he rasterizer
   TPDFRasterImage* Img)  // see RenderPage()
```

The function renders a PDF page into an image buffer in the same way as RenderPage(), but instead of using a callback function to update the destination window, the function blends the image directly into a device context. This function is mainly provided for programming languages in which the usage of callback functions is too complicated or too slow, e.g. C#, VB .Net, VB 6, FoxPro and so on.

The function calls internally the system function SetDIBitsToDevice() to copy the image buffer into the device context. This function depends on the correct creation of the image buffer. Therefore, the rendering engine provides additional helper functions to allocate and resize the image buffer if necessary as well as to redraw the bitmap arbitrary often:

- CreateRasterizerEx() creates the rasterizer and a corresponding DIB Section that is compatible to the device context. The function supports the pixel formats pxf1Bit, pxfGray, pxfBGR, and pxfBGRA. The default pixel format on Windows is pxfBGR. The rasterizer is the owner of the image buffer; it will be released when the rasterizer will be deleted.

- ResizeBitmap() can be used to change the size of the pixel buffer that was created with CreateRasterizerEx(). This function is normally called in the OnResize() event of the form or component into which the image is drawn.

- Redraw() can be used to redraw the image on an arbitrary position. This function is required to achieve flicker free scrolling when CreateRasterizerEx() was used to create the image buffer.

It is of course also possible to allocate the image manually and to attach it to a rasterizer that was created with CreateRasterizer(). However, the image buffer should be created with the system function CreateDIBSection() because SetDIBitsToDevice() fails sometimes when the buffer was allocated with malloc or new.

The parameters *DestX* and *DestY* are passed to SetDIBitsToDevice() when the window is updated. The coordinates can be changed at runtime when RenderPageEx() is executed in a separate thread. This makes it possible to move the image aroud while it is rendered. Each time the position was changed the function updates the entire window to avoid artifacts on screen. The destination coordinates can be set to NULL if not required. The destination coordinate are assumed to be 0, 0 in this case.

The *OnUpdateWindow*() callback function is never excuted when rendering into a device context. However, the members *UpdateOnPathCount* and *UpdateOnImageCoverage* should be initialized as usual so that the window becomes updated from time to time. See RenderPage() for further information.

Remarks:

At time of publication the function can be used on Windows only.

Return values:

When the page was fully rendered the return value is 1. When the page was not fully processed, e.g. due to errors, the return value is 0.

RenderPageToImage (Rendering Engine)

Syntax:

```
LBOOL pdfRenderPageToImage(
    const PPDF* IPDF,            // PDF Instance pointer
    UI32 PageNum,                // Page number (numbering starts at 1)
    const char* OutFile,         // Output file name or NULL
    UI32 Resolution,             // Output resolution or 0
    SI32 Width,                  // Output width, 0, or -maxWidth
    SI32 Height,                 // Output height, 0, or -maxHeight
    TRasterFlags Flags,          // See RenderPage()
    TPDFPixFormat PixFmt,        // Output pixel format
    TCompressionFilter Filter,   // See below
    TImageFormat Format)         // See below

typedef enum
{
    cfFlate    = 0,    // TIFF output
    cfJPEG     = 1,    // TIFF or JPEG output
    cfCCITT3   = 2,    // TIFF output, see 1 Bit Rendering
    cfCCITT4   = 3,    // TIFF output, see 1 Bit Rendering
    cfLZW      = 4,    // TIFF output
    cfLZWBW    = 5,    // TIFF, see 1 Bit Rendering
    cfFlateBW  = 6,    // BMP, PNG, TIFF, see 1 Bit Rendering
    cfJP2K     = 7     // JPEG 2000 output
    cfNone     = 255, // TIFF output only
    // These flags can be combined with cfCCITT3/4, cfFlate, and cfLZW.
    cfDitherFloydSteinberg = 0x00001000, // Floyd Steinberg dithering.
    cfConvGrayToOtsu       = 0x00002000, // The Otsu filter is a special filter to produce black &
                                         // white images. It is very useful if an OCR scan should be
                                         // applied on the resulting 1 bit image. The flag will be
                                         // considered if the pixel format was set to pxfGray.
    cfOrderedDithering     = 0x00004000  // Ordered dithering. The difference in comparison to the
                                         // pixel format pxf1Bit is that we render into a GrayA
                                         // buffer instead of a native 1 bit buffer since this one
                                         // produces incorrect results in certain transparency
                                         // calculations.
}TCompressionFilter;

typedef enum
{
    pxf1Bit,  // BMP, PNG, or TIFF output
    pxfGray,  // BMP, PNG, JPEG, JPEG 2000, or TIFF output
    pxfGrayA, // PNG or TIFF output
```

```
    pxfRGB,    // PNG, JPEG, JPEG 2000, or TIFF output
    pxfBGR,    // BMP
    pxfRGBA,   // BMP, PNG, TIFF
    pxfBGRA,   // BMP
    pxfARGB,   // Unsupported
    pxfABGR,   // Unsupported
    pxfCMYK,   // TIFF, JPEG
    pxfCMYKA   // TIFF, JPEG
}TPDFPixFormat;

typedef enum
{
    ifmTIFF,     // 1 bit, gray, RGB, CMYK, CMYKA, CCITT 3/4, JPEG, Flate, LZW
    ifmJPEG,     // Gray, RGB, CMYK, CMYKA, JPEG compression
    ifmPNG,      // 1 bit, gray, RGB, Flate compression
    ifmReserved, // Reserved for future extensions
    ifmBMP,      // 1 bit, gray, RGB, uncompressed
    ifmJPC       // 1 bit, gray, RGB, JPEG 2000 compression
}TImageFormat;
```

The function renders a PDF page to an image. The page that should be rendered must be closed (Append() or EditPage() open a page and EndPage() closes a page).

If the parameter *OutFile* is set to NULL the image is created in memory. In this case call GetImageBuffer() to get the image buffer and call finally FreeImageBuffer() to release it.

Before rendering the first page the application should set the path from which external CMaps can be loaded (see SetCMapDir()). Set the path with the flag lcmDelayed so that the files will only be loaded if necessary.

Rendering speed can be improved by enabling the image cache. See EnableImageCache() for further information.

It is also possible to create the output image beforehand with CreateImage(). In this case the image will be added to this image encoder and the parameter *OutFile* will be ignored. This can be useful when creating multi-page TIFF images because the compression filter can be changed on every page if necessary.

The output image size can be calculated in different ways:

- **Resolution > 0** and Width == 0 and Height == 0
  Pages are rendered according to the given resolution. Note that PDF pages can be very large. Therefore, it is maybe not possible to render the page in the wished resolution.

- **Resolution > 0** and **Width < 0** and or **Height < 0**
  Pages are rendered according to the given resolution. Negative values of *Width* and *Height* are interpreted as maximum width or height if *Resolution* is greater zero. Since PDF pages can be very large, it is recommended to set the maximum width and height to a value that is low enough so that no out of memory exception occurs, e.g. 5000 x 5000 pixels.

- Resolution == 0 and **Width > 0** or **Height > 0**
  Pages are scaled to the given *Width* or *Height*. If *Width* and *Height* are greater zero then pages are scaled to that size independent of the original page format (not recommended). It is

usually best to set the width or height to zero so that the function can calculate the missing value to preserve the aspect ratio.

The pixel format and the output image format must be compatible. Because TIFF is the only image format that supports different compression filters, the parameter *Filter* will be ignored for all other output formats if the pixel format is not set to pxfGray. For gray images, the filter is also used to determine whether the Floyd-Steinberg dithering algorithm should be applied (see 1 Bit Rendering).

If *Resolution* is greater zero the image size will be calculated as follows:

```
imageWidth  = pageWidth  * Resolution / 72.0
imageHeight = pageHeight * Resolution / 72.0
```

The page width and height is calculated according to the bounding boxes and orientation. When rendering PDF pages to a specific resolution it is recommended to set also the maximum width and height (as negative values) so that the image size can be restricted. This makes it possible to render arbitrary PDF files in high resolutions without further considerations.

1 Bit Rendering

If the pixel format pxf1Bit is used, the page will be rendered with a simple 1 bit ordered dither matrix algorithm. This kind of rendering is fast but not very accurate. Since this pixel format has no alpha channel (which is sometimes required), the output can be wrong if the file uses transparency.

Dithering

To avoid issues with transparency it is recommended to render into a gray pixel format (pxfGray or pxfGrayA) and to apply a dithering algorithm on the result. This can be achieved by combining the compression filter with one of the available dithering algorithms:

- `cfOrderedDithering     -> Simple ordered dithering`
- `cfDitherFloydSteinberg -> Floyd-Steinberg dithering`
- `cfConvGrayToOtsu       -> Optimal filter for OCR engines`

In order to compress the image with CCITT Fax 4, for example, and to apply a simple ordered dithering algorithm, combine the compression filter and dithering flag with a binary or operator, e.g. TCompressionFilter(cfCCITT4 | cfOrderedDithering).

Anti-Aliasing should be disabled when creating 1 bit images to avoid anti-aliasing artifacts.

The Otsu filter is useful if an OCR scan should be applied on the resulting 1 bit image. Most OCR engines produce best results with this filter.

If the output format is Bitmap (BMP) the compression filter is used only to determine whether a dithering algorithm should be applied. The compression filter can be set to any of the above values.

Remarks:

This function is available in an Ansi and Unicode compatible version. Unicode file paths are converted to UTF-8 on non-Windows operating systems.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

RenderPDFFile (obsolete)

Syntax:

```
LBOOL pdfRenderPDFFile(
   const PPDF* IPDF,          // PDF Instance pointer
   const char* OutFile,       // File name or directory
   UI32 Resolution,           // Resolution in DPI
   TRasterFlags Flags,        // Optional flags
   TPDFPixFormat PixFmt,      // Output pixel format
   TCompressionFilter Filter, // Compression filter for TIFF output
   TImageFormat Format)       // Output image format
```

This function is obsolete and should no longer be used. New applications should use RenderPDFFileEx() instead.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

RenderPDFFileEx

Syntax:

```
LBOOL pdfRenderPDFFileEx(
   const PPDF* IPDF,          // PDF Instance pointer
   const char* OutFile,       // File name or directory
   UI32 Resolution,           // Resolution in DPI
   SI32 Width,                // or output width
   SI32 Height,               // or output height
   TRasterFlags Flags,        // Optional flags
   TPDFPixFormat PixFmt,      // Output pixel format
   TCompressionFilter Filter, // Compression filter for TIFF output
   TImageFormat Format)       // Output image format
```

The function renders all PDF pages which are currently in memory and stores the result in a proprietary image format. The pages in memory could be imported from one or more external PDF files, e.g. with ImportPDFFile(), created with DynaPDF functions, or a combination of both.

The parameter *OutFile* can be a path to an existing directory or the file name of the output image. The latter type can be used with TIFF images because this format supports multi-page output. When a file path is used with a single page image format only the first page will be rendered.

The function calls the Ansi C function stat() to determine whether the path is a directory or a file name. A path to a directory must **not** end with a slash or backslash because stat() cannot identify a directory in this case.

RenderPDFFile() and RenderPageToImage() use internally the same code and support therefore the same features. Please read also the description of RenderPageToImage().

The function can render pages in a specific resolution, or scale them to a given width or height. Depending on which parameters are set the image size is calculated as follows:

- **Resolution > 0** and Width == 0 and Height == 0
  Pages are rendered according to the given resolution. Note that PDF pages can be very large. Therefore, it is maybe not possible to render all pages in the wished resolution.

- **Resolution > 0** and **Width < 0** and or **Height < 0**
  Pages are rendered according to the given resolution. Negative values of *Width* and *Height* are interpreted as maximum width or height if *Resolution* is greater zero. Since PDF pages can be very large, it is recommended to set the maximum width and height to a value that is low enough so that no out of memory exception occurs, e.g. 5000 x 5000 pixels.

- Resolution == 0 and **Width > 0** or **Height > 0**
  Pages are scaled to the given *Width* or *Height*. If *Width* and *Height* are greater zero then pages are scaled to that size independent of the original page format (not recommended). It is usually best to set the width or height to zero so that the function can calculate the missing value to preserve the aspect ratio.

On a 32 bit system it is possible to render PDF pages in RGB with up to around 1200 DPI, depending on the page format and available memory. The resolution of gray images can be higher but the encoder must be able to handle such large images. The PNG and bitmap encoders accept images in almost arbitrary resolutions but all other encoders can fail if the resolution is larger than about 2000 DPI.

Remarks:

The function calls the TInitProgressProc() callback function, if set, to initialize a progress bar. The parameter *ProgType* is set to ptWritePage. The TProgressProc() callback function is executed before a page is rendered. The function breaks processing if the return value of the progress callback function is non-zero. See SetProgressProc() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ReplaceFont

Syntax:

```
SI32 pdfReplaceFont(
   const PPDF* IPDF,        // Instance pointer
   const PFNT* PDFFont,     // Parameter of the OnFontNotFount callback
   const char* Name,        // Family or Postscript name of the font
   TFStyle Style,           // Font style to load
   LBOOL NameIsFamilyName)  // If true, Name is a Family name
```

The function replaces a PDF font with another one. The function can be called in the OnFontNotFound callback function of the function CheckConformance(). The return value of the callback function should be the return value of this function. The parameter *PDFFont* is a parameter of the callback function. The pointer is required and must be passed unchanged to the function.

On Linux or Unix system fonts must be loaded with AddFontSearchPath() before this function can be called. This should be done before the first PDF file is imported.

Return value:

If the function succeeds the return value is zero. If the function fails the return value is a negative error code.

ReOpenImportFile

Syntax:

```
LBOOL pdfReOpenImportFile(
   const PPDF* IPDF, // Instance pointer
   UI32 Handle)      // File handle
```

The function re-opens a PDf file so that further contents can be imported from it. The parameter *Handle* must be a valid file handle that OpenImportFile() or OpenImportBuffer() returned.

When the file is no longer needed close the parser instance with CloseImportFileEx().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ReplaceFontEx

Syntax:

```
SI32 pdfReplaceFontEx(
   const PPDF* IPDF,      // Instance pointer
   const PFNT* PDFFont,   // Parameter of the OnFontNotFount callback
   const char* FontFile,  // Path of the font file to be loaded
   LBOOL Embed)           // Must be true when creating PDF/A files
```

The function replaces a PDF font with another one. The font file can be loaded directly. It is not required to install the font on the system. The function can be called in the OnFontNotFound callback function of the function CheckConformance(). The return value of the callback function should be the return value of this function. The parameter *PDFFont* is a parameter of the callback function. The pointer is required and must be passed unchanged to the function.

Return value:

If the function succeeds the return value is zero. If the function fails the return value is a negative error code.

ReplaceICCProfile

Syntax:

```
SI32 pdfReplaceICCProfile(
    const PPDF* IPDF,    // Instance pointer
    UI32 ColorSpace,     // Color space handle
    const char* ICCFile) // File path to an ICC profile
```

This function is used to dynamically create ICC based color spaces within the callback function OnReplaceICCProfile of CheckConformance().

The parameter Type of the callback function contains the profile type that must be inserted. At time of publication only Gray, RGB and CMYK profiles are requested. Because ICC profiles for grayscale color spaces are rarely available a RGB profile can be used instead.

Notice:

The callback function OnReplaceICCProfile should return the value of this function!

Remarks:

This function is available in an Ansi and Unicode compatible version. Because UTF-16 Unicode file paths are not supported under Linux or UNIX the file path is converted to UTF-8 before trying to open the file on such operating systems. However, the usage of the Ansi version with an UTF-8 string is recommended on non-Windows operating systems.

Return values:

If the function succeeds the return value is zero. If the function fails the return value is a negative error code.

ReplaceICCProfileEx

Syntax:

```
SI32 pdfReplaceICCProfileEx(
    const PPDF* IPDF,    // Instance pointer
    UI32 ColorSpace,     // Color space handle
    const void* Buffer,  // File buffer of an ICC profile
    UI32 BufSize)        // Buffer size in bytes
```

The function replaces an ICC profile exactly in the same way as ReplaceICCProfile() but accepts a file buffer as input.

Return values:

If the function succeeds the return value is zero. If the function fails the return value is a negative error code.

ReplaceImage

Syntax:

```
LBOOL pdfReplaceImage(
    const PPDF* IPDF,         // Instance pointer
    const PIMG* Source,       // Pointer of the image object
    const char* Image,        // Path of the new image
    UI32 Index,               // The first image is denoted by 1
    TExtColorSpace CS,        // Image color space
    SI32 CSHandle,            // Color space handle or -1 for device spaces
    TReplaceImageFlags Flags) // See below.

typedef enum
{
    rifDefault       = 0,  // Nothing special to do.
    rifDeleteAltImages = 1,  // Delete all alternate images if any.
    rifDeleteMetadata  = 2,  // Delete the meta data that was associated with the image.
    rifDeleteOCG       = 4,  // Delete the Optional Content Group if any. This changes the visibility
                             // state of the image. Normally the OCG should be left as is.
    rifDeleteSoftMask  = 8   // An image can contain a soft mask that acts as an alpha channel. This
                             // mask can be deleted or left as is. The mask will always be deleted if
                             // the new image contains an alpha channel.
}TReplaceImageFlags;
```

The function replaces an image with another image.

The parameter *Source* must be a valid pointer of an image object. This pointer is part of the TPDFImage structure (member *ObjectPtr*) which is retrieved by functions like GetImageObj() or ParseContent().

The resolution, aspect ratio, color space and so on can be freely chosen. However, note that this function does not change the output position or size. If the aspect ratio of the new image is different, then it will be stretched or shrunk to fit into the output rectangle.

The parameter *CS* specifies the destination color space into which the image should be converted or saved, if the image is already defined in that color space. *CSHandle* must be the handle of that color space if a non-device space is used. The will be ignored for devices spaces.

The function works in the very same way as InsertImageEx() with the following differences:

- The color of an image mask cannot be set or changed because this would require changes on the content stream in which the image is used. An image mask will be created if the image color depth is 1 bit and if color key masking is enabled (see SetUseTransparency()). SetUseTransparency() should normally be set to false before calling this function.

- The image will never be downscaled, independent of the current resolution, because the size of the output rectangle is not known.

If the image that should be replaced is a soft mask of another base image, then make sure that the destination color space is set to esDeviceGray, CalGray, or to a one channel ICC based color space because a soft mask must not contain more than one color channel.

The flags gfUseImageColorSpace, gfIgnoreICCProfiles, gfRealPassThrough, and gfNoBitmapAlpha are all supported. See SetGStateFlags() for further information.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Unicode paths are converted to UTF-8 on non-Windows operating systems.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ReplaceImageEx

Syntax:

```
LBOOL pdfReplaceImageEx(
   const PPDF* IPDF,            // Instance pointer
   const PIMG* Source,          // Pointer of the image object
   const void* Buffer,          // Pointer to image buffer
   UI32 BufSize,                // Buffer size in bytes
   UI32 Index,                  // The first image is denoted by 1
   TExtColorSpace CS,           // Image color space
   SI32 CSHandle,               // Color space handle or -1 for device spaces
   TReplaceImageFlags Flags)    // See below.
```

The function replaces an image with a new image exactly like ReplaceImage() but accepts a file buffer as input. See ReplaceImage() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ReplacePageText

Syntax:

```
LBOOL pdfReplacePageText(
   const PPDF* IPDF,            // Instance pointer
   const char* NewText,         // The text or NULL to delete the string
   struct TPDFStack* Stack)     // Operation stack
```

The function deletes or replaces a text string of a content stream that was found by the function GetPageText() beforehand. If *NewText* is NULL the string will be deleted. The deletion of strings is possible independent of the used font format.

However, if you want to replace a string with a new one you must check the members *Embedded* and *CIDFont* of the structure TPDFStack to determine whether the font is **not** embedded and that no CID font is in use. Because the function does not more than replacing the original string value with the new one, it is recommended that the used font is not embedded. Embedded fonts are usually stored as subset; only the used characters are embedded. If the new string contains characters which are

not included in the font, the string looks misplaced and unsupported characters will be replaced with the .notdef character when viewing the file.

Strings of CID fonts cannot be replaced with this function because such strings are binary strings which cannot be created outside of the library.

However, it is usually best to use this function to delete the original string if necessary. The new text can be written with the function WriteTextMatrixEx() or use ReplacePageTextEx() instead.

If a text record consists of more than one kerning record, it is possible to preserve an arbitrary number of kerning records from deletion. This can be archieved by setting the member *DeleteKerningAt* of the structure TPDFStack to the wished kerning record number before calling this function. All kerning records above this number will be deleted. Note that the parameter *NewText* is ignored in this case.

DynaPDF is delivered with the example edit_text which demonstrates how texts can be replaced in a document. If you want to develop a text replacement algorithm please take a look into this example.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ReplacePageTextEx

Syntax:

```
LBOOL pdfReplacePageTextEx(
    const PPDF* IPDF,          // Instance pointer
    const char* NewText,       // New text or NULL to delete the string
    struct TPDFStack* Stack)   // Operation stack
```

The function deletes or replaces a text string of a content stream that was found by the function GetPageText() beforehand. The function requires a font that must be set with SetFont() or SetFontEx() before calling this function. The new text is internally written with the function WriteTextMatrixEx().

This function changes certain members of the graphics state such as the fill and stroke color, the color space and the text draw mode. If further contents must be written to the file make sure that the color space, text draw mode, and the wished fill and stroke color is initialized with the wished values before writing new contents to the file.

DynaPDF is delivered with the example edit_text which demonstrates how texts can be replaced in a document. If you want to develop a text replacement algorithm please take a look into this example.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ResetAnnotAP

Syntax:

```
LBOOL pdfResetAnnotAP(
    const PPDF* IPDF, // Instance pointer
    SI32 Handle)      // Annotation handle or -1
```

The function forces a rebuild of the appearance stream of the specified annotation. If *Handle* is set to -1 all annotation appearances will be rebuild. This function be useful if an imported annotation has an invalid or empty appearance stream.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ResetEncryptionSettings

Syntax:

```
LBOOL pdfResetEncryptionSettings(
    const PPDF* IPDF) // Instance pointer
```

The function resets the encryption settings which were imported from an external PDF file so that the PDF file in memory can be saved unencrypted.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ResetLineDashPattern

Syntax:

```
LBOOL pdfResetLineDashPattern(
    const PPDF* IPDF) // Instance pointer
```

The function resets a previously defined line dash pattern to its default value (straight line). A line dash pattern can also be removed or reset with the following function call:

```
SetLineDashPattern(pdf, NULL, 0);
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ResizeBitmap (Rendering Engine)

Syntax:

```
LBOOL rasResizeBitmap(
   IRAS* RasPtr,    // Pointer of the rasterizer
   const void* DC,  // Device Context (HDC)
   UI32 Width,      // New width in pixels
   UI32 Height)     // New height in pixels
```

The function changes the size of the internal image buffer that was created with CreateRasterizerEx(). The function should be called in the OnResize() event of the form or component into which the page is rendered.

The new width and height must greater zero.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

RestoreGraphicState

Syntax:

```
LBOOL pdfRestoreGraphicState(
   const PPDF* IPDF) // Instance pointer
```

The function restores a previously saved graphics state. A PDF viewer application maintains an internal data structure called the *graphics state* that holds current graphics control parameters. These parameters define the global framework within which the graphics operators execute.

A well-structured PDF document typically contains many graphical elements that are essentially independent of each other and sometimes nested to multiple levels. The graphics state stack allows these elements to make local changes to the graphics state without disturbing the graphics state of the surrounding environment. The stack is a LIFO (last in, first out) data structure in which the contents of the graphics state can be saved and later restored using the following functions:

- SaveGraphicState() pushes a copy of the entire graphics state onto the stack.
- RestoreGraphicState() restores the entire graphics state to its former value by popping it from the stack.

These functions can be used to encapsulate a graphical element so that it can modify parameters of the graphics state and later restore them to their previous values.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

RotateCoords

Syntax:

```
LBOOL pdfRotateCoords(
   const PPDF* IPDF, // Instance pointer
   double alpha,     // Angle alpha in degrees
   double OriginX,   // Origin of the x-axis
   double OriginY)   // Origin of the y-axis
```

The function rotates the coordinate system at the point *OriginX*, *OriginY* by applying a transformation matrix. It is highly recommended to save the graphics state beforehand, otherwise it is very difficult or impossible to restore the coordinate system later.

After the coordinate system was changed by the function, bottom-up coordinates are active. It is not possible to use top-down coordinates with a rotated coordinate system.

A rotation is internally calculated as follows:

```
TCTM M; // Transformation matrix, the data type is declared in dynapdf.h
double si, co;
si  = sin(alpha * PI / 180.0);
co  = cos(alpha * PI / 180.0);
M.a = co;
M.b = si;
M.c = -si;
M.d = co;
M.x = OriginX;
m.y = OriginY;
```

Note that the origin of the rotation must be set to that position where the coordinates should be rotated. Consider also the new coordinate origin when printing objects into the rotated coordinate system.

Example:

```
pdfSaveGraphicState(pdf);
pdfRotateCoords(pdf, 30.0, 150.0, 450.0); // Set the wished origin
// We don't want to move the rectangle inside the rotated coordinate
// system; the rectangle must be printed at 0, 0 because the origin was
// already set
pdfRectangle(pdf, 0.0, 0.0, 200.0, 100.0, fmStroke);
pdfRestoreGraphicState(pdf);
```

Remarks:

If the graphics state was not saved beforehand the function sets a warning but the transformation is applied.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

RoundRect

Syntax:

```
LBOOL pdfRoundRect(
   const PPDF* IPDF,        // Instance pointer
   double PosX,             // X-Coordinate of rectangle
   double PosY,             // Y-Coordinate of rectangle
   double Width,            // Width of the rectangle
   double Height,           // Height of the rectangle
   double Radius,           // Radius of rounded corners
   TPathFillMode FillMode)  // Fill mode
```

The function draws a rectangle with rounded corners. The radius must not be greater than the half with or height of the rectangle.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the rectangle. If the coordinate system is top-down it defines the upper left corner.

The draw direction can be changed with the function SetDrawDirection().

A rectangle is a closed path that can be filled, stroked or both. It is also possible to draw a rectangle invisible to apply the filling rules nonzero winding number or even-odd. The filling rules are described under ClipPath(). The parameter *FillMode* is ignored if the rectangle is drawn inside a clipping path. The fill modes are described under ClosePath().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example:

RoundRectEx

Syntax:

```
LBOOL pdfRoundRectEx(
    const PPDF* IPDF,        // Instance pointer
    double PosX,             // X-Coordinate of rectangle
    double PosY,             // Y-Coordinate of rectangle
    double Width,            // Width of the rectangle
    double Height,           // Height of the rectangle
    double rWidth,           // Width of an elliptical corner
    double rHeight,          // Height of an elliptical corner
    TPathFillMode FillMode) // Fill mode
```

The function draws a rectangle with elliptical corners. The parameter *rWidth* must not be greater than the half width of the rectangle and the parameter *rHeight* must not be greater than the half height of the rectangle.

If the coordinate system is bottom-up the point *PosX*, *PosY* defines the lower left corner of the rectangle. If the coordinate system is top-down it defines the upper left corner.

The draw direction can be changed with the function SetDrawDirection().

A rectangle is a closed path that can be filled, stroked or both. It is also possible to draw a rectangle invisible to apply the filling rules nonzero winding number or even-odd. The filling rules are described under ClipPath(). The parameter *FillMode* is ignored if the rectangle is drawn inside a clipping path. The fill modes are described under ClosePath().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example:

SaveGraphicState

Syntax:

```
LBOOL pdfSaveGraphicState(
   const PPDF* IPDF) // Instance pointer
```

The function saves the current graphics state. A saved graphics state must be restored before the page, template or pattern will be closed.

DynaPDF maintains an internal data structure called the *graphics state* that holds current graphics control parameters. These parameters define the global framework within which the graphics operators execute.

A well-structured PDF document typically contains many graphical elements that are essentially independent of each other and sometimes nested to multiple levels. The graphics state stack allows these elements to make local changes to the graphics state without disturbing the graphics state of the surrounding environment. The stack is a LIFO (last in, first out) data structure in which the contents of the graphics state can be saved and later restored using the following functions:

- SaveGraphicState() pushes a copy of the graphics state onto the stack.
- RestoreGraphicState() restores the graphics state to its former value by popping it from the stack.

These functions can be used to encapsulate a graphical element so that it can modify parameters of the graphics state and later restore them to their previous values.

The maximum allowed nesting level is 28 in PDF 1.7. This is not a limit as such, but arises from the fact that the corresponding q and Q operators are implemented by the PostScript gsave and grestore operators when generating PostScript output.

The graphics state consists of the following variables:

Parameter	Data type	Initial Value
CharSpacing	float	0.0f
Clipping Path	Vector array	Crop box or Media box
DashPattern	double*	NULL (Solid line)
DashPhase	UI32	0
FillColor	BYTE[32]	Black
FillColorSpace	TExtColorSpace, IColorSpace*	esDeviceRGB, NULL
FillPattern	IPattern*	NULL
Font	IFont*	NULL
Leading	float	0.0f
LineCapStyle	TLineCapStyle	csButtCap
LineJoinStyle	TLineJoinStyle	jsMiterJoin
LineWidth	float	1.0f
Matrix	TCTM	{1, 0, 0, 1, 0, 0}
MiterLimit	float	10.0f
StrokeColor	BYTE[32]	Black
StrokeColorSpace	TExtColorSpace, IColorSpace*	esDeviceRGB, NULL

Parameter	Data type	Initial Value
StrokePattern	IPattern*	NULL
TextDrawMode	TDrawMode	dmNormal
TextScale	float	100.0f
WordSpacing	float	0.0f

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

ScaleCoords

Syntax:

```
LBOOL pdfScaleCoords(
    const PPDF* IPDF, // Instance pointer
    double sx,        // Scale factor of the x-axis
    double sy)        // Scale factor of the y-axis
```

The function scales the coordinate system by applying a transformation matrix. It is highly recommended to save the graphics state beforehand, otherwise it is very difficult or impossible to restore the coordinate system later.

After scaling the coordinate system, bottom-up coordinates are active. It is not possible to use top-down coordinates inside a scaled coordinate system.

The parameters *sx* and *sy* are set directly to *a* and *d* members of the transformation matrix:

```
TCTM M; // Transformation matrix, the data type is declared in dynapdf.h
M.a = sx;
M.b = 0.0;
M.c = 0.0;
M.d = sy;
M.x = 0.0;
M.y = 0.0;
```

The parameters sx and sy must not be zero.

Remarks:

If the graphics state was not saved beforehand the function produces a warning but the transformation is applied.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SelfTest

Syntax:

```
LBOOL pdfSelfTest(
    const PPDF* IPDF) // Instance pointer
```

The function checks the size of all required data types and checks whether the endian configuration of the library is correct. The test results are printed to stderr. The function prints also a MD 5 hash which must be the same as under Windows.

This function can be used under Linux and UNIX operation systems to check whether the library was correctly compiled. Precompiled libraries are available for the most important operating systems. The makefiles for a specific OS are always delivered by DynaForms with the source code license for this OS. Send a mail to support@dynaforms.com for further information.

Set3DAnnotProps

Syntax:

```
LBOOL pdfSet3DAnnotProps(
    const PPDF* IPDF,               // Instance pointer
    UI32 Handle,                    // Annotation handle
    T3DActivationType ActType,      // Activation type
    T3DDeActivateType DeActType,    // Deactivation type
    T3DInstanceType InstType,       // Instantiation type
    T3DDeActInstance DeInstType,    // De-instantiation type
    LBOOL DisplToolbar,             // Display the 3D toolbar?
    LBOOL DisplModelTree)           // Display the model tree?

// Activation type
typedef enum
{
    at3D_AppDefault,  // Use the default of the viewer
    at3D_PageOpen,    // Activate it when the page is opened.
    at3D_PageVisible, // Activate it if the page becomes visible.
    at3D_Explicit,    // Inactive until explicitly activated (default).
}T3DActivationType;

// Deactivation type
typedef enum
{
    dt3D_AppDefault,   // Use the default of the viewer
    dt3D_PageClosed,   // As soon as the page is closed.
    dt3D_PageInvisible, // When the page becomes invisible (default).
    dt3D_Explicit      // Until explicitly deactivated.
}T3DDeActivateType;

// What should be done with the instance when the annotation becomes
// deactivated?
typedef enum
{
    di3D_AppDefault,      // Use the default of the viewer
    di3D_UnInstantiated,  // The annotation will be uninstantiated (default)
    di3D_Instantiated,    // The annotation is left instantiated
    di3D_Live             // Animations stay live
}T3DDeActInstance;
```

```
// How instantiate the annotation?
typedef enum
{
    it3D_AppDefault,   // Use the default of the viewer
    it3D_Instantiated, // Instantiated but animations are disabled.
    it3D_Live          // Instantiated, animations are enabled (default).
}T3DInstanceType;

// This flag can be combined with the annotation handle
#define TRANSP_3D_ANNOT 0x40000000
```

The function sets or changes several important properties of a 3D annotation. The parameter Handle must be a valid handle of a 3D annotation.

Extensions:

Adobe introduced an extension in PDF 1.7, Extension Level 3, to enable the creation of 3D annotations with a transparent background. This extension has been reflected in the parameter *Annot*. To create a 3D annotation with a transparent background combine the annotation handle with the flag TRANSP_3D_ANNOT with a binary or operator, e.g. annotHandle | TRANSP_3D_ANNOT.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Set3DAnnotScript

Syntax:

```
LBOOL pdfSet3DAnnotScript(
  const PPDF* IPDF,   // Instance pointer
  UI32 Annot,         // Handle of a 3D annotation
  const char* Value,  // JavaScript that should be executed
  UI32 Len)           // Length of the script
```

The function assigns a JavaScript to a 3D annotation. JavaScripts which are assigned with a 3D annotation have direct access to the 3D context. Global JavaScripts have only limited access to the 3D context of a 3D annotation but it is possible to execute a script that is assigned with the annotation. Note that such JavaScripts must be accessed over the annotation:

```
annot = getAnnots3D(0)[0].context3D; // Get the annotation context
// This function must be defined in the JavaScript of the 3D annotation.
annot.Test();
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAllocBy

Syntax:

```
LBOOL pdfSetAllocBy(
   const PPDF* IPDF, // Instance pointer
   SI32 Value)       // Value in KB
```

The function sets the size of the memory blocks in kilo bytes, which will be allocated if memory must be allocated for page content streams. Memory allocation is slow, especially re-allocation of memory by using realloc(). Because of this all content streams are buffered by DynaPDF to avoid too much memory allocation calls. The property *AllocBy* defines the size of memory that is allocated each time when more memory must be allocated.

The processing speed depends heavily on memory allocation. The default size of pre-allocated memory for content streams is 16 KB. It can be set to a larger value if necessary to improve processing speed.

However, if too much memory is allocated at runtime, processing speed will be slower and an out of memory exception can occur. If the size is too small, processing speed is slower too because of the many memory allocation calls.

In most cases, it is not required to change the property *AllocBy*.

SetAltFonts

Syntax:

```
LBOOL pdfSetAltFonts(
   const PPDF* IPDF,   // Instance pointer
   UI32 ListHandle,    // Handle returned by CreateAltFontList().
   struct char** List, // Array of font names
   UI32 Count)         // Number of values in the array
```

The function fills an alternate font list created by CreateAltFontList() with data. Alternate font list are tested if one more glyphs were not found in the current font. Since the fonts in the list are used as fallback they should be compatible to the active font in regard to properties like serif, sans serif, proportional or fixed pitch, and so on.

The names in a font list must be Family Names since DynaPDF must be able to load different styles of a font.

A list should be sorted by priority and not by name. That means index 0 should contain a font name that is most probably the best match. If this one was not available or incompatible then the next font in the list will be tested until the end of the list was reached. If no suitable font was found then system fonts are tested.

The fonts in the list should support the language that must be output. Otherwise, the fonts can never be selected. An alternate font list can be arbitrary large but it is usually best to restrict the number fonts in a list to speed up processing.

A list can contain font names which are not available on the system. This causes no error. Non existing fonts are simply ignored. This makes it possible to define suitable alternate font lists statically. Such lists can then be loaded on demand.

After a list was filled with data it can be activated with ActivateAltFontList(). If a list is no longer needed delete it with DeleteAltFontList().

Remarks:

This function is implemented in an Ansi and Unicode compatible variant. Since font names are Unicode strings, the Unicode variant is preferred.

Please note that alternate font lists are used only, if complex text layout was enabled. This can be done by setting the flag *gfComplexText* with SetGStateFlags().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotBorderEffect

Syntax:

```
LBOOL pdfSetAnnotBorderEffect(
    const PPDF* IPDF,     // Instance pointer
    UI32 Handle,          // Annotation handle
    TBorderEffect Value)  // See below

typedef enum
{
    beSolid,   // Default
    beCloudy1, // Circle diameter 9 units
    beCloudy2  // Circle diameter 17 units
}TBorderEffect;
```

The function sets or changes the border effect of a Square, FreeText, or Polygon annotation.

Circle annotations support the cloudy border effect too but this is not yet implemented.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotBorderStyle

Syntax:

```
LBOOL pdfSetAnnotBorderStyle(
    const PPDF* IPDF,   // Instance pointer
    UI32 Handle,        // Annotation handle
    TBorderStyle Style) // Border style

typedef enum
{
    bsSolid       = 0, // Solid border
    bsBevelled    = 1, // Bevelled border
    bsInset       = 2, // Inset border
    bsUnderline   = 3, // Underline only
    bsDashed      = 4, // Dashed border
    bsUserDefined = 5  // Not allowed
}TBorderStyle;
```

The function sets or changes the border style of an annotation. Note that the border style has no effect if the border width is set to zero or if the border color is set to NO_COLOR (see SetAnnotColor(), SetAnnotBorderWidth()).

Note also that not all styles are meaningful for all annotation types. For example, while link annotations support all border styles, FreeText annotations support the styles bsSolid and bsDashed only. File Attach, Sound, Stamp, or Text annotations for example support no border style because these annotation types contain no appearance on which a border style could be applied.

Setting the border style to an unsupported value causes no error; the value will be ignored instead.

Remarks:

When changing the border style of a Free Text annotation DynaPDF must rebuild the appearance stream of the annotation. The text position can slightly change especially if the border width was changed too. See also FreeTextAnnot().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotBorderWidth

Syntax:

```
LBOOL pdfSetAnnotBorderWidth(
  const PPDF* IPDF, // Instance pointer
  UI32 Handle,      // Annotation handle
  double LineWidth) // Line width of the border or zero
```

The function sets or changes the border width of an annotation. Although the line width can be set to any positive floating point number, it should be set to full integer values between 0 through 12.

If no border should be drawn set the line width to zero.

Remarks:

When changing the border width of a Free Text annotation DynaPDF must rebuild the appearance stream of the annotation. The text position depends on the line width of the border. Changing the border width changes also the position of the text. See also FreeTextAnnot().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotColor

Syntax:

```
LBOOL pdfSetAnnotColor(
  const PPDF* IPDF,       // Instance pointer
  UI32 Handle,            // Annotation handle
  TAnnotColor ColorType,  // Color to be change
  TPDFColorSpace CS,      // Color space
  UI32 Color)             // Color to be set
typedef enum
{
    fcBackColor   = 0, // Link annotations support no background color!
    fcBorderColor = 1,
    fcTextColor   = 2  // Free Text annotation's only
}TFieldColor, TAnnotColor;
```

The function sets or changes the color of an annotation. The color can be defined in any device color space. However, at time of publication the function converts the color back to DeviceRGB.

Note that not all annotation types support a background or border color. For example, link annotations support a border color but no background or text color.

Remarks:

When changing a color of a Free Text annotation DynaPDF must rebuild the appearance stream of the annotation. This can cause slightly changes in the text position. See also FreeTextAnnot().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotFlags

Syntax:

```
LBOOL pdfSetAnnotFlags(
   const PPDF* IPDF,  // Instance pointer
   TAnnotFlags Flags) // Flags see below
typedef UI32 TAnnotFlags;
#define afNone           0x00000000 // No flags are set
#define afInvisible      0x00000001 // see below
#define afHidden         0x00000002 // see below
#define afPrint          0x00000004 // Annotation is printable
#define afNoZoom         0x00000008 // Do not zoom the annotation
#define afNoRotate       0x00000010 // Do not rotate the annotation
#define afNoView         0x00000020 // See description below
#define afReadOnly       0x00000040 // Changes are not allowed
#define afLocked         0x00000080 // See description below
#define afToggleNoView   0x00000100 // See description below
#define afLockedContents 0x00000200 // See description below
```

The function sets the default flags used for new annotations. The parameter *Flags* is a bit mask; multiple flags can be set with a binary or operator (e.g. afPrint | afReadOnly). It is also possible to add each flag separately; the previous flags are only deleted if afNone is used.

```
pdfSetAnnotFlags(pdf, afPrint | afReadOnly);
```

```
// or set each flag separately, the result is the same
pdfSetAnnotFlags(pdf, afPrint);
pdfSetAnnotFlags(pdf, afReadOnly);
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Flag	Description
afNone	No flags are set.
afInvisible	If set, do not display the annotation if it does not belong to one of the standard annotation types and no annotation handler is available.
afHidden	*(PDF 1.2)* If set, do not display or print the annotation or allow it to interact with the user, regardless of its annotation type or whether an annotation handler is available.
afPrint	*(PDF 1.2)* If set, print the annotation when the page is printed. If clear, never print the annotation, regardless of whether it is displayed on the screen. This can be useful, for example, for annotations representing interactive pushbuttons, which would serve no meaningful purpose on the printed page.
afNoZoom	*(PDF 1.3)* If set, do not scale the annotation's appearance to match the magnification of the page. The location of the annotation on the page (defined by the upper-left corner of its annotation rectangle) remains fixed, regardless of the page magnification.
afNoRotate	*(PDF 1.3)* If set, do not rotate the annotation's appearance to match the rotation of the page. The upper-left corner of the annotation rectangle remains in a fixed location on the page, regardless of the page rotation.
afNoView	*(PDF 1.3)* If set, do not display the annotation on the screen or allow it to interact with the user. The annotation may be printed (depending on the setting of the afPrint flag), but should be considered hidden for purposes of on-screen display and user interaction.
afReadOnly	*(PDF 1.3)* If set, do not allow the annotation to interact with the user. The annotation may be displayed or printed (depending on the settings of the afNoView and afPrint flags), but should not respond to mouse clicks or change its appearance in response to mouse motions.
afLocked	*(PDF 1.4)* If set, do not allow the annotation to be deleted or its properties (including position and size) to be modified by the user. However, this flag does not restrict changes to the annotation's contents.

`afToggleNoView`	*(PDF 1.5)* If set, invert the interpretation of the NoView flag for certain events.
`afLockedContents`	*(PDF 1.7)* If set, do not allow the contents of the annotation to be modified by the user. This flag does not restrict deletion of the annotation or changes to other annotation properties, such as position and size.

SetAnnotFlagsEx

Syntax:

```
LBOOL pdfSetAnnotFlagsEx(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle,      // Annotation handle
    SI32 Flags)       // New Flags
```

The function sets or changes the flags of an annotation. The available flags are described in detail at SetAnnotFlags().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotHighlightMode

Syntax:

```
LBOOL pdfSetAnnotHighlightMode(
    const PPDF* IPDF,     // Instance pointer
    UI32 Handle,          // Link annotation handle
    THighlightMode Mode)  // New highlight mode
```

The function sets or changes the highlight mode of a link annotation. The highlight mode is applied when clicking with the mouse on the annotation. Only link annotations support a highlight mode. However, setting the mode to other annotation types causes no error; the value will be ignored instead.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotIcon

Syntax:

```
LBOOL pdfSetAnnotIcon(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle,      // Text annotation handle
    TAnnotIcon Icon)  // New icon
typedef enum
{
    aiComment,
    aiHelp,
    aiInsert,
    aiKey,
    aiNewParagraph,
    aiNote,
    aiParagraph,
    aiUserDefined // Not usable
}TAnnotIcon;
```

The function sets or changes the icon of a text annotation. The parameter *Handle* must be a valid handle of a Text annotation. See also TextAnnot().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotLineEndStyle

Syntax:

```
LBOOL pdfSetAnnotLineEndStyle(
    const PPDF* IPDF,      // Instance pointer
    UI32 Handle,           // Handle of a Line or PolyLine annotation
    TLineEndStyle Start,   // End line style of the start point
    TLineEndStyle End)     // End line style of the end point
typedef enum
{
    leNone,
    leButt,
    leCircle,
    leClosedArrow,
    leDiamond,
    leOpenArrow,
    leRClosedArrow,
    leROpenArrow,
    leSlash,
    leSquare
}TLineEndStyle;
```

The function sets or changes the line end styles of a Line or PolyLine annotation.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotLineDashPattern

Syntax:

```
LBOOL pdfSetAnnotLineDashPattern(
    const PPDF* IPDF,   // Instance pointer
    UI32 Handle,        // Annotation handle
    const float* Dash,  // Dash array or NULL
    UI32 NumValues)     // Number of array values (can be zero)
```

The function sets or deletes the line dash pattern of an annotation. The following annotation types support line dash patterns:

- `atCircle`
- `atInk`
- `atLine`
- `atPolygon`
- `atPolyLine`
- `atSquare`

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotMigrationState

Syntax:

```
SI32 pdfSetAnnotMigrationState(
    const PPDF* IPDF,   // Instance pointer
    UI32 Annot,         // Handle of a markup annotation
    TAnnotState State,  // See below
    const char* User)   // The user who sets the state

typedef enum
{
    asNone,
    asAccepted,
    asRejected,
    asCancelled,
    asCompleted,
    asCreateReply
}TAnnotState;
```

The function adds a migration state or a reply to a markup annotation. All annotation types with exception of 3D, Link, Movie, Screen, PrinterMark, TrapNet, and Watermark are markup annotations.

Migration states are stored in text annotations as reply to the base annotation. Further states are stored as reply to the last reply and so on; the result is a single linked list. Because the base annotation contains no reference to the last reply or migration state the function must search for it. To speed up processing the function returns the handle of the text annotation so that the next state can directly be added to this annotation. This handle is also required if a reply should be created.

A reply contains usually a comment or text from a user. This string can be set with SetAnnotString() (the string *Content* must be set).

Example (C++):

```
SI32 annot, reply;
PPDF* pdf = pdfNewPDF();
if (!pdf) return -1;

pdfCreateNewPDF(pdf, "out.pdf");
pdfSetPageCoords(pdf, pcTopDown);
pdfAppend(pdf);
    annot = pdfSquareAnnot(pdf, 50.0, 50.0, 200.0, 100.0, 1.0, NO_COLOR,
                255, csDeviceRGB, "Jim", "Test", "Just test...");
    reply = pdfSetAnnotMigrationState(pdf, annot, asCreateReply, "Harry");
    pdfSetAnnotString(pdf, reply, asContent, "This is a reply!");
pdfEndPage(pdf);

pdfCloseFile(pdf);
pdfDeletePDF(pdf);
```

Remarks:

This function is implemented in an Ansi and Unicode compatible variant. A migration state or annotation reply is defined since PDF 1.5. The function adjusts the PDF version automatically if it is lower than PDF 1.5.

Return values:

If the function succeeds the return value is the handle of the text annotation, a value greater or equal zero. If the function fails the return value is a negative error code.

SetAnnotOpacity

Syntax:

```
LBOOL pdfSetAnnotOpacity(
   const PPDF* IPDF, // Instance pointer
   UI32 Handle,      // Annotation handle
   double Value)     // New value must be in the range 0.0..1.0.
```

The function changes the opacity of a markup annotation. The function GetAnnotEx() or GetPageAnnotEx() can be used to determine whether an annotation is a markup annotation.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotOpenState

```
LBOOL pdfSetAnnotOpenState(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle,      // Markup or PopUp annotation handle
    LBOOL Open)       // New state
```

The function sets the open state of a markup or PopUp annotation. Markup annotations can be connected with a PopUp annotation that displays additional comments in a window (like a Text annotation). The open state can only be changed if the base annotation is connected with a PopUp annotation. If the base annotation contains no PopUp annotation or if the handle refers to a non-markup annotation the function does nothing.

All annotation types with exception of 3D, Link, Movie, Screen, PrinterMark, TrapNet, and Watermark are markup annotations.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotOrFieldDate

Syntax:

```
LBOOL pdfSetAnnotOrFieldDate(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle,      // Annotation or field handle
    LBOOL IsField,    // If true, Handle is a field handle
    TDateType Type,   // See below
    UI32 DateTime)    // Number of seconds elapsed since January 1, 1970

typedef enum
{
    dtCreationDate, // Markup annotations only
    dtModDate       // Annotations or form fields
}TDateType;
```

The function sets or changes the creation or modification date of an annotation or form field. If the parameter *IsField* is set to true, a valid field handle must be passed to the parameter *Handle*. An annotation handle is expected otherwise.

Annotations and form fields support a modification date but a creation date is supported by markup annotations only.

The function GetAnnotEx() or GetPageAnnotEx() can be used to determine whether an annotation is a markup annotation.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.
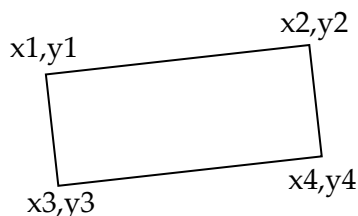
SetAnnotQuadPoints

Syntax:

```
LBOOL pdfSetAnnotQuadPoints(
    const PPDF* IPDF,         // Instance pointer
    UI32 Handle,              // Annotation handle
    struct TFltPoint* Value,  // Array of 4 x n points
    UI32 Count)               // Number of points
```

The function sets or changes the quad points definition of a *Highlight*, *Link*, *Redakt*, *Squiggly*, *StrikeOut*, or *Underline* annotation. The function must be called within an open page.

Independent of the used coordinate system (bottom up or top down), the points must be defined in the following order:

The coordinates of the vertices are interpreted in current user space. Any transformation that was applied on the coordinate system will be taken into account.

The number of points must be 4 or a multiple of 4.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotString

Syntax:

```
LBOOL pdfSetAnnotString(
    const PPDF* IPDF,           // Instance pointer
    UI32 Handle,                // Annotation handle
    TAnnotString StringType,    // String type that should be changed
    const char* Value)          // The new value or NULL to delete it

typedef enum
{
    asAuthor,
    asContent,
    asName,
    asSubject,
    asRichStyle, // Default style string. -> FreeText annotations only.
    asRichText   // Rich text string. -> Supported by markup annotations.
}TAnnotString;
```

The function sets, changes, or deletes a string of an annotation. The string *RichText* or *RichStyle* must be set after the annotation content was changed because these strings will be deleted when changing

the contents of an annotation. Note also that DynaPDF does not use the rich text to create the appearance stream. However, the rich text should still coincide with the annotations content.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetAnnotSubject

Syntax:

```
LBOOL pdfSetAnnotSubject(
   const PPDF* IPDF,    // Instance pointer
   UI32 Handle,         // Annotation handle
   const char* Value)   // The subject to be set
```

The function sets or changes the optional subject string of an annotation. The subject is defined since PDF 1.5. The function adjusts the PDF version automatically if it is lower than PDF 1.5.

The annotation types atFileLink, atPageLink, atPopUp, and atWebLink do not support a subject string. The function returns with an error when trying to set the subject on these annotation types.

If the parameter Value is NULL or if it contains an empty string the subject string is deleted in the annotation.

Remarks:

This function is implemented in an Ansi and Unicode compatible variant.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetBBox

Syntax:

```
LBOOL pdfSetBBox(
   const PPDF* IPDF,        // Instance pointer
   TPageBoundary Boundary,  // Bounding box
   double LeftX,            // lower left x
   double LeftY,            // lower left y
   double RightX,           // upper-right x
   double RightY)           // upper-right y

typedef enum
{
   pbArtBox,    // Art box
   pbBleedBox,  // Bleed box
   pbCropBox,   // Crop box
   pbTrimBox,   // Trim box
   pbMediaBox   // Media box
}TPageBoundary;
```

The function sets a specific bounding box of the current open page. A PDF page may be prepared either for a finished medium, such as a sheet of paper, or as part of a prepress process in which the

content of the page is placed on an intermediate medium, such as film or an imposed reproduction plate.

In the latter case, it is important to distinguish between the intermediate page and the finished page. The intermediate page may often include additional production-related content, such as bleeds or printer marks that falls outside the boundaries of the finished page. To handle such cases, a PDF page can define as many as five separate boundaries to control various aspects of the imaging process.

A bounding box is defined as rectangle giving the coordinates of a pair of diagonally opposite corners. The media box of a page is normally expressed in a normalized form where the coordinates of the lower-left point are set to zero.

A normalized bounding box can be easily defined as follows:

```
pdfSetBBox(pdf, pbMediaBox, 0, 0, 612, 792);
```

The parameters *RightX, RightY* represent the width and height of the media box (this is the paper format if no crop box is present). The media box should normally be defined in the normalized form because the bounding box defines also the coordinate origin of the page. Predefined paper formats can also be set with the function SetPageFormat().

It is also possible to change the bounding box of a template or pattern with this function. These object types support the media box only; the other bounding boxes are ignored. Note that the coordinates of the objects inside a page, template or pattern depends on the coordinate origin of the media box.

The crop box crops the page as the name suggests. The crop box represents also the paper format if present.

If the width of a bounding box is less than 1 unit the function deletes the bounding box from the object. The media box is required, it cannot be deleted.

The minimum page size is 3 x 3 Units; the maximum is 14,400 x 14,400 Units. The minimum size of a template or pattern is 1 x 1 Units and the maximum is 14,400 x 14,400 Units.

If a page with more than 14,400 Units should be created then it is possible to scale the page format with SetUserUnit(). A User Unit is an additional scaling factor that is used to calculate the page format. This makes it possible to create extremely large page formats without exceeding the limits of the page coordinate system.
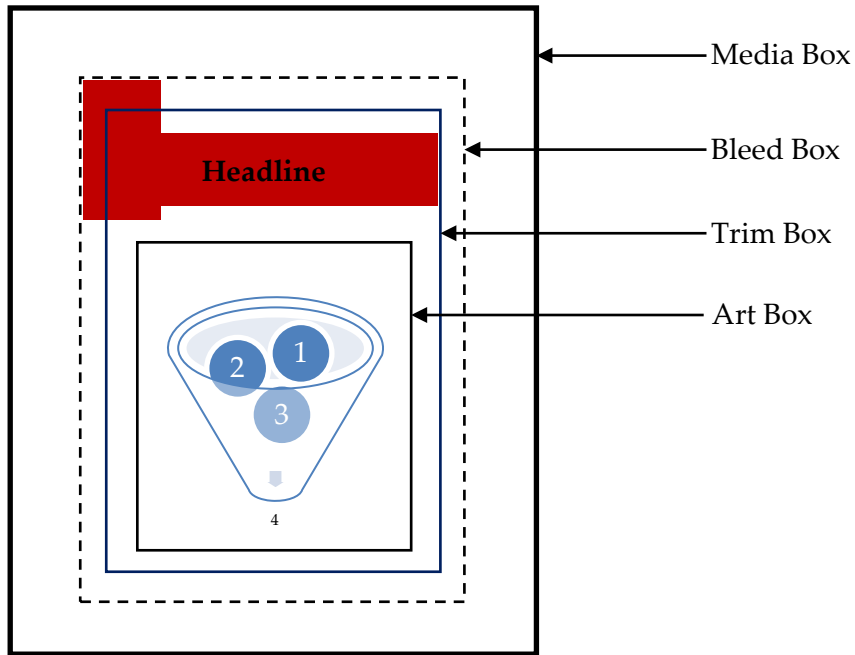
Remarks:

No bounding box of a page should be larger than the media box.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Bounding boxes:

SetBidiMode

Syntax:

```
LBOOL pdfSetBidiMode(
    const PPDF* IPDF,  // Instance pointer
    TPDFBidiMode Mode) // Bidirectional mode

typedef enum
{
    bmLeftToRight = 0, // Apply the bidi algorithm in Left to Right layout
    bmRightToLeft = 1, // Apply the bidi algorithm in Right to Left layout
    bmNone        = 2  // Default -> do not apply the bidi algorithm
}TPDFBidiMode;
```

The function sets or changes the bidirectional mode. When using a bidirectional 8 bit code page the bidirectional algorithm is applied by default in Left to Right mode also if the bidi mode is set to bmNone (default). This mode produces identical results in comparison to applications like Edit or WordPad from Microsoft Windows.

If the current font was loaded with the code page cpUnicode the bidirectional algorithm is applied only if the bidi mode is not set to bmNone.

Remarks:

DynaPDF uses internally the Reference Bidi Algorithm of the Unicode Consortium to process bidirectional strings. The algorithm supports the entire UCS-2 range and it is used by many applications. However, the Right to Left mode produces different results in comparison to Microsoft's Uniscribe. If you need the same result in Right to Left mode as Uniscribe would produce then you must pre-process the strings with this library and output the resulting Unicode strings with disabled bidirectional algorithm. The font must be loaded with the code page cpUnicode in this case.

Notice:

The bidi mode is ignored during EMF conversion. To enable the bidirectional algorithm during EMF conversion you must set the flag mfApplyBidiAlgo with SetMetaConvFlags().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetBookmarkDest

Syntax:

```
LBOOL pdfSetBookmarkDest(
    const PPDF* IPDF,   // Instance pointer
    SI32 ABmk,          // Bookmark handle
    TDestType DestType, // see below
    double a,           // Various, depends on destination type
    double b,           // Various, depends on destination type
    double c,           // Various, depends on destination type
    double d)           // Various, depends on destination type

typedef enum
{
    dtXY_Zoom,   // Three parameters (a, b, c) -> (X, Y, Zoom)
    dtFit,       // No parameters
    dtFitH_Top,  // One parameter    (a)
    dtFitV_Left, // One parameter    (a)
    dtFit_Rect,  // Four parameters  (left, bottom, right, top)
    dtFitB,      // No parameters
    dtFitBH_Top, // One parameter    (a)
    dtFitBV_Left // One parameter    (a)
}TDestType;
```

The function sets or changes the destination of a bookmark. The parameter *ABmk* must be a valid bookmark handle. The destination page will not be changed by this function it must be set correctly with the function AddBookmark(). If it should be changed use the function ChangeBookmark().

The destination created by this function allows jumping to a specific position in a document instead of simply opening a page such as a normal bookmark does.

Destination types	Description

dtXY_Zoom	Display the page designated by page with the coordinates (left top) positioned at the top-left corner of the window and the contents of the page magnified by the factor zoom. A zero value for any of the parameters left top or zoom specifies that the current value of that parameter is to be retained unchanged.
	
	Example:
	
	`// The zoom factor is left unchanged`
	`SetBookmarkDest(pdf, bmk, dtXY_Zoom, 50, 750, 0, 0);`

dtFit	Display the page designated by page with its contents magnified just enough to fit the entire page within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the page within the window in the other dimension. This destination type has no parameters, the values of a, b, c, d are ignored.

`dtFitH_Top`	Display the page designated by page with the vertical coordinate top positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of the page within the window.
	
	Example:
	
	`// The parameter a specifies the top coordinate`
	`pdfSetBookmarkDest(pdf, bmk, dtFitH_Top, 750, 0, 0, 0);`

`dtFitV_Left:`	Display the page designated by page with the horizontal coordinate left positioned at the left edge of the window and the contents of the page magnified just enough to fit the entire height of the page within the window.
	
	Example:
	
	`// The parameter a specifies the left edge`
	`pdfSetBookmarkDest(pdf, bmk, dtFitV_Left, 50, 0, 0, 0);`

`dtFit_Rect`	Display the page designated by page with its contents magnified just enough to fit the rectangle specified by the coordinates left bottom right and top entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the rectangle within the window in the other dimension. Note, the maximum zoom factor supported by Adobe's Acrobat is limited to 16 (Acrobat 4/5) or 64 if Acrobat 6 is used. It is not possible to zoom into the rectangle if it is too small. **Example**: `pdfSetBookmarkDest(pdf, bmk,dtFit_Rect,150,550,450,700);`

The destination types *dtFitB, dtFitBH_Top and dtFitBV_Left* use always the media box of the page to fit the page into the window. All other destination types use the crop box if any.

As you can see above that the usage of the destination types are the same as for a go-to action. The function creates in real a go-to action which is executed by the bookmark. However, the action is stored in a more compact format and cannot be shared with other objects.

If a destination should be used with multiple objects such as page links, create a go-to action instead and add it to the bookmark with the function AddActionToObj(). The same action can then be added to other objects.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetBookmarkStyle

Syntax:

```
LBOOL pdfSetBookmarkStyle(
    const PPDF* IPDF, // Instance pointer
    SI32 ABmk,        // Bookmark handle
    TFStyle Style,    // Style beeing used to display the text
    UI32 RGBColor)    // Color of the bookmark

typedef SI32 TFStyle;
#define SI32 fsNone       = 0; // Default
#define SI32 fsItalic     = 1; // Italic
#define SI32 fsBold       = 2; // Bold
#define SI32 fsUnderlined = 4; // Unsupported
#define SI32 fsStriked    = 8; // Unsupported
```

Since Acrobat 5 bookmarks support a user defined color and the text style can be changed to italic, bold and so on. Older versions of Adobe's Acrobat ignore the style information. Bookmarks support RGB colors only. The parameter Style is a bit mask, the flags can be combined, e.g. to create a bolditalic bookmark.

Example:

pdfSetBookmarkStyle(pdf, bmk, fsBold | fsItalic, PDF_RGB(45, 144, 54));

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetBorderStyle

Syntax:

```
LBOOL pdfSetBorderStyle(
    const PPDF* IPDF,  // Instance pointer
    TBorderStyle Style) // Border style

typedef enum
{
    bsSolid       = 0, // Solid border
    bsBevelled    = 1, // Bevelled border
    bsInset       = 2, // Inset border
    bsUnderline   = 3, // Underline only
    bsDashed      = 4, // Dashed border
    bsUserDefined = 5  // Not allowed
}TBorderStyle;
```

The function set the global border style which is used for newly created form fields. It is also possible to change the style of a specific form field, see SetFieldBorderStyle() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCharacterSpacing

Syntax:

```
LBOOL pdfSetCharacterSpacing(
    const PPDF* IPDF, // Instance pointer
    double Value)     // Character spacing
```

The function sets the current character spacing. The function requires an open page, template or pattern.

```
Default value = 0
```

`Value = 0`	Character
`Value = 10.0`	C h a r a c t e r

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCheckBoxChar

Syntax:

```
LBOOL pdfSetCheckBoxChar(
    const PPDF* IPDF,                // Instance pointer
    TCheckboxChar CheckBoxChar) // see below
```

```
typedef enum
{
    ccCheck,
    ccCircle,
    ccCross1,
    ccCross2,
    ccCross3,
    ccCross4,
    ccDiamond,
    ccSquare,
    ccStar
}TCheckBoxChar;
```

The function sets the character which should be used for newly created check boxes.

```
Default value = ccCheck
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Check box characters

ccCheck

ccCircle

ccCross1

ccCross2

ccCross3

ccCross4

ccDiamond

ccSquare

ccStar

SetCheckBoxDefState

Syntax:

```
LBOOL pdfSetCheckBoxDefState(
   const PPDF* IPDF, // Instance pointer
   UI32 AField,      // Field handle
   LBOOL Checked)    // Default state
```

This function changes the default state of a check box; it can differ from the current visible state of the check box. The default state is used when the form is reset with a Reset Form Action.

The default state of a new check box is always identical with the visible state.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCheckBoxState

Syntax:

```
LBOOL pdfSetCheckboxState(
   const PPDF* IPDF, // Instance pointer
   UI32 AField,      // Field handle
   LBOOL Checked)    // State
```

The function changes the state of a check box. The parameter *AField* must be a valid check box handle.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCIDFont

Syntax:

```
SI32 pdfSetCIDFont(
   const PPDF* IPDF, // Instance pointer
   UI32 CMapHandle,  // Handle of a CMap returned by LoadCMap()
   const char* Name, // Font name
   TFStyle Style,    // Font style
   double Size,      // Font size
   LBOOL Embed)      // If true, the font will be embedded
```

The function loads an OpenType or TrueType font with an external CMap. A font that is used with an external CMap is called a composite font or CID font. A CID-keyed font is an OpenType font (with Postscript outlines) that contains CID font operators.

CID-keyed fonts provide a convenient and efficient method for defining multiple-byte character encodings as wells as fonts with a large number of glyphs. These capabilities provide great flexibility for representing text in writing systems for languages with large character sets, such as Chinese, Japanese, and Korean (CJK).

The CID-keyed font architecture specifies the external representation of certain font programs, called CMap and CIDFont files, along with some conventions for combining and using those files. PDF does not support the entire CID-keyed font architecture, which is independent of PDF; CID-keyed fonts can also be used in other environments. For complete documentation on the architecture and the file formats, see Adobe Technical Notes #5092, CID-Keyed Font Technology Overview, and #5014, Adobe CMap and CIDFont Files Specification.

The term CID-keyed font reflects the fact that CID numbers (character identifiers) are used to index and access the glyph descriptions in the font. A CID refers always into a predefined character collection.

A character collection is an ordered set of all glyphs needed to support one or more popular character sets for a particular language. The order of the glyphs in the character collection

determines the CID number for each glyph. Each CID-keyed font explicitly references the character collection on which its CID numbers are based.

A CMap (character map) file specifies the correspondence between character codes and the CID numbers used to identify glyphs. It is equivalent to the concept of an encoding in simple fonts. Whereas a simple font allows a maximum of 256 glyphs to be encoded and accessible at one time, a CMap can describe a mapping from multiple-byte codes to thousands of glyphs in a large CID-keyed font. For example, it can describe Shift-JIS, one of several widely used encodings for Japanese. See also LoadCMap().

The supported character collection of a CID-keyed font must match the one of the CMap file. If the CMap file provides a mapping into another character collection as the font supports, then the function returns with an error and the font will not be loaded.

Although TrueType fonts have no native notation of CIDs it is also possible to use TrueType fonts with external CMaps as long as the font contains a CMap in the format 0, 4, 6, 10, or 12. TrueType fonts which contain only a CMap in format 2 cannot be used with external CMap files (format 2 is a mixed 8/16 bit encoding that is supported in certain pure CJK fonts). However, such fonts can be loaded with SetFont().

A key feature of the CID-keyed font architecture is that it works also with non-embedded fonts. Due the predefined character collections it is possible to display the text also if the font is not available on the system.

Documents in western writing systems use about 80 through 100 glyphs of a font on average, while Asian documents use often more than 400 through several thousand glyphs in one document. The amount of font data that must be stored in Asian documents is not comparable with western documents.

Although font embedding is still recommended, the amount of data is sometimes too large to embed all fonts. If font embedding is not possible it is recommended to use fonts which are probably available on the user's operating system.

Please note that DynaPDF knows nothing about the encoding when using a font with an external CMap. Functions which output formatted text like WriteFText() can only be used with CMaps which map Unicode (UTF-16) to CIDs. The wide string version of the function must be used in this case.

Word Spacing

Word spacing applies to the space character of a string. However, most CJK encodings support more than one space character, such as a proportional space, ideographic space, full width space, half width space and many more. DynaPDF treats always CID 1 as space character, independent of the used character collection. All other CIDs are treated as ordinary glyphs.

Vertical Writing Mode

When using a CMap for vertical writing mode the text extends always from top to bottom. The coordinate origin of the text depends on the font origin (see SetFontOrigin()). If the font origin is set to orDownLeft the first character is placed on the font's baseline. This can be somewhat confusing because the remaining text is of course placed below the first character. It is usually best to set the font origin to orTopLeft, also if bottom up coordinates are used.

Because the text width is simply the font size, GetTextWidth() returns the text height in vertical writing mode.

Encodings Identity-H and Identity-V

Iit is also possible to load TrueType and OpenType fonts with the special encodings Identity-H or Identity-V with this function. These are encodings and no predefined external CMaps. That means the CID-keyed font architecture is not used.

A font that was loaded with these encodings can be used as if it were loaded with SetFont() with the code page cpUnicode. The only difference is that the font can be left unembedded. However, the PDF file stores glyph indexes instead of CIDs and glyph indexes are private for every font. That means if the original font is not available on the system then it is not possible to display the text!

It is **not** recommended to use these encodings with non-embedded fonts, although it works. If you want to use fonts with these encodings then make sure that you use only standard fonts which are normally available on the users system. The encodings Idenitity-H and Identity-V require Acrobat 6 or higher. Due to a change in the font search algorithm in Acrobat 10 it is no longer possible to display such files with Acrobat 4/5.

Known issues:

- At time of publication CID fonts cannot be used with form fields.
- Vertical writing mode does not work with WriteFText().

Remarks:

All functions which output text report a warning if one or more characters cannot be found. Call GetMissingGlyphs() to determine which characters could not be found.

Return values:

If the function succeeds the return value is the font handle, a value greater or equal zero. If the function fails, the return value is a negative error code.

SetCMapDir

Syntax:

```
SI32 pdfSetCMapDir(
    const PPDF* IPDF,      // Instance pointer
    const char* Path,      // Directory that contains CMap files
    TLoadCMapFlags Flags) // See below

typedef enum
{
    lcmDefault   = 0, // Load the cmaps in the directory now
    lcmRecursive = 1, // Load sub directories recursively
    lcmDelayed   = 2  // Load the files when required
}TLoadCMapFlags;
```

The function sets a search path from which external CMap files can be loaded. External CMap files are sometimes required, e.g. when extracting text from Asian PDF files or when rendering PDF pages. However, it is also possible to load external CMaps with OpenType or TrueType fonts. This is a key feature when creating Asian PDF files (see SetCIDFont() for further information).

Although the function name suggests that only one directory can set at time it is possible to load an arbitrary number of directories. Each time the function is called a duplicate check is performed so that only unique CMap files reside in the CMap cache.

The parameter *Flags* specifies how and when the CMaps should be loaded. When the flag lcmDelayed is set, the function adds the path to the internal array of search paths, but the CMap files will be loaded when a font requires an external CMap, e.g. during text extraction or rendering. The flag lcmDelayed should be set in viewer applications so that the first page can be loaded as fast as possible.

CMap files can be divided into three categories:

- CMaps which provide a mapping from an arbitrary character encoding to a predefined character collection.

- CMaps which provide an identity mapping to a predefined character collection. Such CMaps should contain the DSC comment "%%BeginResource: CMap (Identity)".

- CMaps which provide a CID to Unicode mapping of a character collection.

If a CMap should be loaded for use with an OpenType or TrueType font, two CMap files are required: the one that provides the mapping into the character collection, and the one that provides the CID to Unicode mapping. If one of these CMap files base in turn on another CMap, the corresponding base CMap must be available too.

The following notes should be considered when working with external CMaps:

- To improve processing speed the directory in which CMap files are stored should not contain other file types.

- CMap files should be loaded once per PDF instance and one PDF instance should be used as long as possible to create an arbitrary number of PDF files.

- The CMap name and the file name should be identical.

- A valid CMap file must begin with the DSC comment "%!PS-Adobe-3.0 Resource-CMap" (excluding double quotes). Other DSC comments are optional.

- The keys /CMapType, /WMode, /CMapName and /CIDSystemInfo must be fully defined within the first 4096 bytes.

- DynaPDF uses the value of /CMapName and the CIDSystemInfo dictionary to identify a CMap file.

Remarks:

The parameter *Path* must be an absolute path if the CMaps are loaded delayed. Otherwise it is maybe not possible to load the files, e.g. if the current directory was already changed at the time the files must be loaded.

The function performs a duplicate check during execution. If two CMap files with the same name, registry, and ordering will be found, the one with the higher supplement number will be added to the cache. The function parses the CMap's header information only. A CMap file must be loaded into memory with LoadCMap() before it can be used with SetCIDFont(). However, the CMap's header information can be accessed with GetCMap() without loading the CMap file.

The function is implemented in an Ansi and Unicode compatible version. Unicode paths are converted to UTF-8 on non-Windows operating systems.

Return values:

If the function succeeds the return value is the number of CMap files in the cache. This value can directly be used to access the CMap headers with GetCMap(). If the function fails the return value is a negative error code. The only reason why this function can fail is out of memory.

SetColDefFile

Syntax:

```
LBOOL pdfSetColDefFile(
   const PPDF* IPDF, // Instance pointer
   UI32 EmbFile)     // Handle of an embedded file
```

The function sets the initial document of a portable collection that should be opened in the viewer application. See also CreateCollection().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetColSortField

Syntax:

```
LBOOL pdfSetColSortField(
   const PPDF* IPDF,     // Instance pointer
   UI32 ColField,        // Handle of a collection field
   LBOOL AscendingOrder) // If true, sort the list in ascending order
```

The function sets the collection field that should be used to sort the list of embedded files.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetColorMask

Syntax:

```
LBOOL pdfSetColorMask(
   const PPDF* IPDF, // Instance pointer
   UI32 ImageHandle, // Image handle
   SI32* Mask,       // Mask values (min / max per component) or NULL
   UI32 Count)       // Number of mask values
```

The functions sets or overrides the color mask of an image. The parameter *ImageHandle* must be a valid handle of an image that should get the mask. The bit depth of the image must be higher than 1.

The parameter *Mask* must be an array of min / max pairs for every color channel. Each integer must be in the range 0 to $2^{BitsPerComponent}-1$.

If *Mask* is NULL or if *Count* is zero an existing color mask will be deleted if any. *Count* holds the full length of the array which must be 2 x NumComponents.

Remarks:

Color Key Masking depends on exact color values. Since JPEG or JPEG 2000 compressed images produce interpolated colors, these filters should not be used with a color mask. To achieve predictable results use Flate compression instead (see SetCompressionFilter()).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetColors

Syntax:

```
LBOOL pdfSetColors(
   const PPDF* IPDF, // Instance pointer
   UI32 Color)       // Color value defined in the current color space
```

The function sets the fill and stroke color. The parameter *Color* must be defined in the current color space. For example, if the current color space is DeviceGray the color value must be in the range 0 to 255. CMYK colors can be constructed with the macro PDF_CMYK() or with the function CMYK() which is available in most programming languages. RGB colors can be constructed with the macro PDF_RGB() or with the function RGB() which is available in most programming languages.

The function requires an open page, template, or pattern.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetColorSpace

Syntax:

```
LBOOL pdfSetColorSpace(
   const PPDF* IPDF,              // Instance pointer
   TPDFColorSpace ColorSpace) // see below

typedef enum
{
   csDeviceRGB  = 0,
   csDeviceCMYK = 1,
   csDeviceGray = 2
}TPDFColorSpace;
```

The function activates a device color space in the graphics state. All color values must be defined in the current color space. Images or EMF graphics are automatically converted to the current color space. The default color conversion rules can be modified with the function SetGStateFlags().

Extended color spaces can be set with SetExtColorSpace(), SetExtFillColorSpace(), and SetExtStrokeColorSpace().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCompressionFilter

Syntax:

```
LBOOL pdfSetCompressionFilter(
    const PPDF* IPDF,              // Instance pointer
    TCompressionFilter Filter) // see below

typedef enum
{
    cfFlate    = 0,
    cfJPEG     = 1,
    cfCCITT3   = 2, // PDF or TIFF output
    cfCCITT4   = 3, // PDF or TIFF output
    cfLZW      = 4, // TIFF or GIF output
    cfReserved = 5, // Reserved for future extensions.
    cfFlateBW  = 6, // TIFF, PNG, or BMP output
    cfJP2K     = 7, // PDF or JPEG2000 output
    cfJBIG2    = 8, // PDF output only
    // Special flags for AddRasImage(), RenderPageToImage(), and RenderPDFFile().
    cfDitherFloydSteinberg = 0x00001000, // Floyd Steinberg Dithering.
    cfConvGrayToOtsu       = 0x00002000, // Optimal filter for an OCR scan.
    // Special flags for Optimize()
    cfPresLosslessFilter = 0x00020000, // Preserve loss-less compression filters. Use the specified
                                       // filter otherwise.
    cfPresLossyFilter    = 0x00040000  // Preserve lossy compression filters. Use the specified
                                       // filter otherwise.
}TCompressionFilter;
```

The function sets the compression filter that should be used to compress images. The filters are described in detail at InsertImageEx().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetCompressionLevel

Syntax:

```
LBOOL pdfSetCompressionLevel(
    const PPDF* IPDF,                    // Instance pointer
    TCompressionLevel CompressLevel) // see below

typedef enum
{
    clNone    = 0, // No compression (ignored for images)
    clDefault = 1, // Normal compression ratio
    clFastest = 2, // Less compression ratio but faster
    clMax     = 3  // Maximum compression ratio but slower
}TCompressionLevel;
```

The function sets the current compression level. If the compression level is clNone, content streams will be left uncompressed. This value is useful if a content stream must be debugged, the value will be ignored for image streams. The compression level is also used by certain image compression filters, see InsertImage() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetContent

Syntax:

```
LBOOL pdfSetContent(
    const PPDF* IPDF,   // Instance pointer
    const char* Buffer, // New content stream
    UI32 BufSize)       // Buffer size in bytes
```

This function replaces the content stream of the currently open page or template with a new one. If the parameter *Buffer* is NULL the content stream of the page or template will be deleted. In the latter case page resources such as fonts, images and so on will be deleted from the page object too.

Remarks:

Use this function in combination with GetContent() if you need to change a content stream in a manner that DynaPDF does not supports. Do never change a content stream when you don't know exactly what you are doing.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetDateTimeFormat

Syntax:

```
LBOOL pdfSetDateTimeFormat(
    const PPDF* IPDF, // Instance pointer
    UI32 TxtField,    // Text field handle
    TPDFDateTime Fmt) // Date or time format
typedef enum
{
    dfMM_D            = 0,
    dfM_D_YY          = 1,
    dfMM_DD_YY        = 2,
    dfMM_YY           = 3,
    dfD_MMM           = 4,
    dfD_MMM_YY        = 5,
    dfDD_MMM_YY       = 6,
    dfYY_MM_DD        = 7,
    dfMMM_YY          = 8,
    dfMMMM_YY         = 9,
    dfMMM_D_YYYY      = 10,
    dfMMMM_D_YYYY     = 11,
    dfM_D_YY_H_MM_TT  = 12,
    dfM_D_YY_HH_MM    = 13,
    /* time formats */
    df24HR_MM         = 14,
    df12HR_MM         = 15,
    df24HR_MM_SS      = 16,
    df12HR_MM_SS      = 17
}TPDFDateTime;
```

The function restricts the allowed value of a text field to a date time format and applies this format if the value was valid. A date time format is represented as two separate JavaScript actions in PDF which are automatically created and added to the text field by this function.

The same formats can also be applied manually by creating two JavaScript actions, one for the OnKeyStroke event, and one for the OnFormat event of the text field. See AddActionToObj() for a description of the events.

DynaPDF uses the JavaScript functions *AFDate_Keystroke() / AFDate_KeystrokeEx()* and *AFDate_Format() / AFDate_FormatEx()* to apply a date time format. The functions are described in the JavaScript scripting reference which is available at http://www.adobe.com.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetDefBitsPerPixel

Syntax:

```
LBOOL pdfSetDefBitsPerPixel(
    const PPDF* IPDF, // Instance pointer
    SI32 Value)       // Currently supported values are 8 and 24 bit
```

The function sets the default color depth in bits per pixel, which determines whether images should be downsampled. If the property is to 8 bits per pixel images are converted to 256 indexed color images. At time of publication only two values are supported:

- *24 bit*: No conversion
- *8 bit*: Conversion to 256 indexed color image

```
Default value = 24
```

Downsampling will only be applied if Flate compression is used. The color table is always created in the current color space.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetDocInfo

Syntax:

```
LBOOL pdfSetDocInfo(
    const PPDF* IPDF,     // Instance pointer
    TDocumentInfo DInfo, // see below
    const char* Value)   // Info string
typedef enum
{
    diAuthor      = 0,
    diCreator     = 1,
    diKeywords    = 2,
    diProducer    = 3,
```

```
    diSubject     = 4,
    diTitle       = 5,
    diCompany     = 6,
    diPDFX_Ver    = 7,  // GetInDocInfo() or GetInDocInfoEx()
    diCustom      = 8,  // Not supported -> Use SetDocInfoEx()
    diPDFX_Conf   = 9,  // GetInDocInfo() or GetInDocInfoEx()
    diCreationDate = 10, // Available after a PDF file was imported
    diModDate     = 11  // GetInDocInfo() or GetInDocInfoEx()
}TDocumentInfo;
```

The function sets or changes a document info entry. This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252. To create a documents info entry in an arbitrary 8 bit or CJK encoding convert the string to Unicode with the function ConvToIncode() and use the Unicode version to set the entry.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetDocInfoEx

Syntax:

```
LBOOL pdfSetDocInfoEx(
    const PPDF* IPDF,     // Instance pointer
    TDocumentInfo DInfo,  // see SetDocInfo() above
    const char* Key,      // User defined key
    const char* Value)    // The value for a specific key
```

The function sets or changes a document info entry. This function supports also user defined keys which can be created or changed if necessary. If the parameter *DInfo* is set to diCustom the parameter *Key* must contain a unique key.

The following keywords are reserved and must not be used as user defined keys:

- Author
- CreationDate
- GTS_PDFXVersion
- Keywords
- ModifyDate
- Producer
- Subject
- Title
- Trapped

Note that the function does not check whether a reserved key is used. Using such a key causes maybe errors which are mostly not reported in viewer applications.

User defined keys should be defined as 7 bit ASCII string and the usage of special characters like /, \, #, or character codes higher than 127 should be avoided. The value of the document info entry should not contain binary data.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252 only. To create a documents info entry in an arbitrary 8 bit or CJK encoding convert the string to Unicode with the function ConvToIncode() and use the Unicode version to set the document info entry.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetDrawDirection

Syntax:

```
LBOOL pdfSetDrawDirection(
   const PPDF* IPDF,           // Instance pointer
   TDrawDirection Direction)   // Draw direction

typedef enum
{
  ddCounterClockwise = 0,
  ddClockwise        = 1
}TDrawDirection;
```

The function sets the draw direction of closed vector graphics such as rectangles, ellipses, triangles and so on. The draw direction is important if a path should be filled with the nonzero winding number rule or even-odd rule. Both modes are described in detail under ClipPath().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetEMFFrameDPI

Syntax:

```
LBOOL pdfSetEMFFrameDPI(
   const PPDF* IPDF, // Instance pointer
   UI32 DPIX,        // Horizontal DPI per inch
   UI32 DPIY)        // Vertical DPI per inch
```

This function can be used to adjust DPI value which is used to calculate the picture size of an EMF file. The values of *DPIX* and *DPIY* are only used if the flag mfUseRclFrame is set (see SetMetaConvFlags()).

The flag mfUseRclFrame is primarily used to convert EMF files which where originally created from non-portable WMF files. Such files contain often a wrongly calculated picture size (rclBounds) so that the picture size must be calculated from the rectangle rclFrame of the EMF file header. This is often the one and only way to convert those files successfully to PDF.

The default DPI value is 100. If the EMF picture appears too large then decrease the DPI value, e.g. to 72.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetEMFPatternDistance

Syntax:

```
LBOOL pdfSetEMFPatternDistance(
    const PPDF* IPDF, // Instance pointer
    double Value)     // Distance between pattern lines in units
```

The function changes the default distance between lines of standard patterns during EMF conversion.

```
Default value = 4.0
Minimum value = 1.0
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetErrorMode

Syntax:

```
LBOOL pdfSetErrorMode(
    const PPDF* IPDF, // Instance pointer
    TErrMode ErrMode) // see below
typedef SI32 TErrMode;
#define emIgnoreAll       0x00000000 // Default
#define emSyntaxError     0x00000001
#define emValueError      0x00000002
#define emWarning         0x00000004
#define emFileError       0x00000008
#define emFontError       0x00000010
#define emAllErrors       0x0000FFFF
#define emNoFuncNames     0x10000000 // Do not output function names
#define emUseErrLog       0x20000000 // Redirect all messages to the error log
// Special flags for CheckConformance().
// These flags add info messages to the error log. The error callback function is not invoked.
#define emLogFonts        0x00010000 // If set, CheckConformance() logs which fonts were replaced
with
                                     // system fonts or converted to Type3
#define emLogFontsVerbose 0x00020000 // If set, the path to the font file is added to the message too
```

The error mode specifies which error types should be treated as fatal error and whether error messages should be added to the error log instead of calling the error callback function.

By default, DynaPDF ignores all non-fatal errors but calls the error callback function (if set) if an error occurs. The last error message is also internally stored so that GetErrorMessage() is able to return the last error message. Once a fatal error occurred, processing breaks immediately and no further error messages or warnings are returned.

The special flag *emNoFuncNames* names can be used to avoid the output of the function name in which the error occured. Error messages start normally with the function name that produced the

error, e.g. "SetFont: Font not found!". While this information is useful during development, it is usually not useful in an end user application.

If the flag *emUseErrLog* is set, DynaPDF redirects all error messages to the error log, see GetErrLogMessage() / GetErrLogMessageCount() / ClearErrorLog(). The error log is always cleared when CreateNewPDF() is called, but the error messages reside in memory when CloseFile() or FreePDF() is called.

The parameter *ErrMode* is a bit mask; multiple flags can be set with the bitwise or operator, e.g. (emSyntaxError | emWarning).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetExtColorSpace

Syntax:

```
LBOOL pdfSetExtColorSpace(
  const PPDF* IPDF, // Instance pointer
  UI32 Handle)      // Handle of an extended color space
```

The function activates an extended color space in the graphics state. An extended color spaces are non-device color spaces, such as ICCBased, Lab, Separation, DeviceN, and so on. The current fill and stroke color are initialized to black after the color space has been changed. Device color spaces can be set with SetColorSpace(). The fill and stroke color spaces can also be set separately with SetExtFillColorSpace() and SetExtStrokeColorSpace(). See also Color Spaces.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetExtFillColorSpace

Syntax:

```
LBOOL pdfSetExtFillColorSpace(
   const PPDF* IPDF, // Instance pointer
   UI32 Handle)      // Handle of an extended color space
```

The function activates an extended color space for fillings in the graphics state. The fill color is initialized to black after the color space has been changed. See also Color Spaces.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetExtGState

Syntax:

```
LBOOL pdfSetExtGState(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle)      // Handle of an extended graphics state object
```

The function activates an extended graphics state. Extended graphics states can be used to adjust certain settings of the graphics state. See CreateExtGState() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetExtStrokeColorSpace

Syntax:

```
LBOOL pdfSetExtStrokeColorSpace(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle)      // Handle of an extended color space
```

The function activates an extended color space for strokes in the graphics state. The stroke color is initialized to black after the color space has been changed. See also Color Spaces.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldBackColor

Syntax:

```
LBOOL pdfSetFieldBackColor(
    const PPDF* IPDF, // Instance pointer
    UI32 AColor)      // Background color
```

The function sets the background color used for newly created interactive form fields and annotations. Normal annotations support RGB colors only. Form fields support the color spaces DeviceRGB, DeviceGray, and DeviceCMYK. The color value must be defined in the current color space.

If the background should appear transparent set the color to NO_COLOR.

```
#define NO_COLOR 0xFFFFFFF1
Default value = NO_COLOR
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldBBox

Syntax:

```
LBOOL pdfSetFieldBBox(
    const PPDF* IPDF,      // Instance pointer
    UI32 AField,           // Field handle
    struct TPDFRect* BBox) // The new bounding box
```

The function changes the bounding box of a field. The bounding box must be defined in bottom up coordinates. The Top member of the structure *BBox* must be larger than Bottom as well as Right must be larger than the Left member.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldBorderColor

Syntax:

```
LBOOL pdfSetFieldBorderColor(
    const PPDF* IPDF, // Instance pointer
    UI32 AColor)      // Border color
```

The function sets the border color used for newly created interactive form fields and annotations. Normal annotations support RGB colors only. Form fields support the color spaces DeviceRGB, DeviceGray and DeviceCMYK. The color value must be defined in the current color space.

If the border should appear transparent set the color to NO_COLOR.

```
#define NO_COLOR 0xFFFFFFF1
Default value = 0 // Black
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldBorderStyle

Syntax:

```
LBOOL pdfSetFieldBorderStyle(
    const PPDF* IPDF,   // Instance pointer
    UI32 AField,        // Field handle
    TBorderStyle Style) // Border style

typedef enum
{
    bsSolid       = 0, // Solid border
    bsBevelled    = 1, // Bevelled border
    bsInset       = 2, // Inset border
    bsUnderline   = 3, // Underline only
    bsDashed      = 4, // Dashed border
    bsUserDefined = 5  // Not allowed
}TBorderStyle;
```

The function changes the border style of a specific Interactive Form field. The parameter *AField* must be a valid field handle.

Remarks:

It is not possible to change the border style of imported button fields. The global border style which is used for newly created fields can be set with the function SetBorderStyle().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldBorderWidth

Syntax:

```
LBOOL pdfSetFieldBorderWidth(
    const PPDF* IPDF, // Instance pointer
    UI32 AField,      // Field handle
    double LineWidth) // Line width of the border
```

This function can be used to change the border width of a field. The parameter *AField* must be a valid field handle. The parameter *LineWidth* must not be negative and it should either be 0, 1, 2, or 3 units (no border, thin, medium, or thick). Note that Adobe's Acrobat supports only these values. It is possible to draw a field with any border width, but if the field is repainted due to a reset action or due to other changes, the field appearance will be changed in Adobe's Acrobat and the border width is adjusted to the nearest supported value.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldCalcOrder

Syntax:

```
LBOOL pdfSetFieldCalcOrder(
    const PPDF* IPDF, // Instance pointer
    SI32 CurrIndex,   // Current calc order index
    UI32 NewIndex)    // New calc order index
```

The function moves a field in the global calc order array. This is the order in which field values will be recalculated when the value of any field changes.

The current calc order can be accessed with GetFieldCalcOrder(). The calc order array contains field handles. A field handle is a simple array index.

Example:

```
// Assume the current calc order array looks as follows:
calcOrder[0] = 5;
calcOrder[1] = 9;
calcOrder[2] = 45;
```

```
calcOrder[3] = 122;
// Move the field of index 1 to index 3
pdfSetFieldCalcOrder(pdf, 1, 3);
// Result:
calcOrder[0] = 5;
calcOrder[1] = 45;
calcOrder[2] = 122;
calcOrder[3] = 9;
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldColor

Syntax:

```
LBOOL pdfSetFieldColor(
    const PPDF* IPDF,         // Instance pointer
    UI32 AField,              // Field handle
    TFieldColor ColorType,    // Color type that should be changed
    TPDFColorSpace CS,        // Color space
    UI32 Color)               // Color value
typedef enum
{
    fcBackColor   = 0,
    fcBorderColor = 1,
    fcTextColor   = 2
}TFieldColor;
typedef enum
{
    csDeviceRGB  = 0,
    csDeviceCMYK = 1,
    csDeviceGray = 2
}TPDFColorSpace;
```

The function sets a specific color of an interactive form field. The parameter *AField* must be a valid field handle. The background and border color of a form field must be defined in the same color space. The color space for the text color can be defined in a separate color space (e.g. DeviceGray for the text and DeviceRGB for the background and border).

If the border or background should appear transparent set the value to NO_COLOR. The text color cannot be transparent.

```
#define NO_COLOR 0xFFFFFFF1
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldExpValue

Syntax:

```
LBOOL pdfSetFieldExpValue(
    const PPDF* IPDF,      // Instance pointer
    UI32 AField,           // Field handle
    SI32 ValIndex,         // Value index or -1
    const char* Value,     // New value
    const char* ExpValue,  // New export value
    LBOOL Selected)        // Select the value?
```

This function can be used to change the choice values of a combo or list box, or to change the state and export value of check box. In the latter case, the parameter *ValIndex* will be ignored. If the field handle refers to a radio button, *ValIndex* represents the index into the Kids array of the radio button.

To determine how many values are defined in a field call the function GetFieldExpValCount() or GetFieldEx().

The parameter *Value* is ignored if the field is a check box or radio button. However, the parameter *ExpValue* is required for these field types.

Notice:

If you don't need to change the value or export value then use the function SetFieldExpValueEx() instead.

If the field is either a combo box or list box, the parameter *ExpValue* is optional and can be NULL. If the parameter *Value* is set to NULL or to an empty string, the choice value will be deleted. Note that all choice values in a combo or list box must be unique. DynaPDF checks whether another choice value with the new name does already exist. If this is the case then the function will fail.

Combo boxes which have the field flag ffEdit set accept a value that was directly typed into the field in a viewer. Such a value may or may not exist the list of choice values. In any case, a special handling is required for this case because the value must be set as the field's value independent of whether it is already included in the list of choice value.

To achieve this, set the parameter *ValIndex* to PDF_MAX_INT (0x7FFFFFFF) or any index greater than the number of choice values in the field. The parameter *Value* can also be set to NULL or to an empty string to delete the field's value. If the string is set then it becomes the new field value but it is not added to the list of choice values if not included.

Note that the above handling is only valid for combo boxes which have the ffEdit field flag set.

Example (C++):

```cpp
...
// aField is a handle of a combo box in this example; we want to
// deselect the currently selected value, that's all.
char* value, *expValue;
LBOOL selected;
SI32 valCount = pdfGetFieldExpValCount(pdf, aField);
for (SI32 i = 0; i < valCount; i++)
{
   if (pdfGetFieldExpValueEx(pdf, aField, i, value, expValue, selected))
   {
      if (selected)
      {
         pdfSetFieldExpValue(pdf, aField, i, value, expValue, false);
         break;
      }
   }
}
```

Remarks:

To enumerate the choice values of a combo box, list box or radio button use the function GetFieldExpValueEx().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldExpValueEx

Syntax:

```cpp
LBOOL pdfSetFieldExpValueEx(
   const PPDF* IPDF,   // Instance pointer
   UI32 AField,        // Field handle
   UI32 ValIndex,      // Value index
   LBOOL Selected,     // New state
   LBOOL DefSelected)  // New default state
```

The function marks a choice value of a combo or list box as selected or unselected. It can also be used to change the state of check boxes or the children of a radio button. The parameter ValIndex will be ignored if the field is a normal check box (no child of a radio button or field group).

Remarks:

To enumerate the choice values of a combo box, list box or radio button use the function GetFieldExpValueEx().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldFlags

Syntax:

```
LBOOL pdfSetFieldFlags(
    const PPDF* IPDF,   // Instance pointer
    UI32 AField,        // Field handle
    TFieldFlags Flags,  // Field flags, see below
    LBOOL Reset)        // Reset the flags or add them?

typedef UI32 TFieldFlags;
// Basic flags supported by all field types except group fields and
// radio button fields.
#define ffReadOnly        0x00000001
#define ffRequired        0x00000002
#define ffNoExport        0x00000004
#define ffInvisible       0x00000008
#define ffHidden          0x00000010
#define ffPrint           0x00000020
#define ffNoZoom          0x00000040
#define ffNoRotate        0x00000080
#define ffNoView          0x00000100
// Special flags supported by specific fields only
#define ffMultiline       0x00001000 // Text fields only
#define ffPassword        0x00002000 // Text fields only
#define ffNoToggleToOff   0x00004000 // Radio buttons, check boxes
#define ffRadioIsUnion    0x04000000 // PDF 1.5 Radio buttons
#define ffCommitOnSelCh   0x08000000 // PDF 1.5 Combo and list boxes
#define ffEdit            0x00040000 // Combo boxes only
#define ffSorted          0x00080000 // Combo and list boxes
#define ffFileSelect      0x00100000 // PDF 1.4 Text fields only
#define ffMultiSelect     0x00200000 // PDF 1.4 List boxes only
#define ffDoNotSpellCheck 0x00400000 // PDF 1.4 Text fields, combo boxes
#define ffDoNotScroll     0x00800000 // PDF 1.4 Text fields only
#define ffComb            0x01000000 // PDF 1.5 Text fields only
```

The function sets the flags of a specific interactive form field. The parameter *AField* must be a valid field handle. The parameter *Flags* is a bit mask, multiple flags can be set by adding the values or use a bitwise Or operator, e.g. ffPrint | ffMultiline. If the parameter *Reset* is true the flags of the field are set to the ones specified. If *Reset* is false, the flags are added by using a bitwise Or operator.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Flag	Description
ffInvisible	If set, do not display the annotation if it does not belong to one of the standard annotation types and no annotation handler is available.
ffHidden	*(PDF 1.2)* If set, do not display or print the annotation or allow it to interact with the user, regardless of its annotation type or whether an annotation handler is available.
ffPrint	*(PDF 1.2)* If set, print the annotation when the page is printed. If clear, never print the annotation, regardless of whether it is displayed on the screen. This can be useful, for example, for annotations representing interactive pushbuttons, which would serve no meaningful purpose on the printed page.
ffNoZoom	*(PDF 1.3)* If set, do not scale the annotation's appearance to match the magnification of the page. The location of the annotation on the page (defined by the upper-left corner of its annotation rectangle) remains fixed, regardless of the page magnification.
ffNoRotate	*(PDF 1.3)* If set, do not rotate the annotation's appearance to match the rotation of the page. The upper-left corner of the annotation rectangle remains in a fixed location on the page, regardless of the page rotation.
ffNoView	*(PDF 1.3)* If set, do not display the annotation on the screen or allow it to interact with the user. The annotation may be printed (depending on the setting of the ffPrint flag), but should be considered hidden for purposes of on-screen display and user interaction.
ffReadOnly	*(PDF 1.3)* If set, do not allow the annotation to interact with the user. The annotation may be displayed or printed (depending on the settings of the afNoView and ffPrint flags), but should not respond to mouse clicks or change its appearance in response to mouse motions.
ffRequired	If set, the field must have a value at the time it is exported by a submit-form action (see CreateSubmitAction() for further information). Supported by all fields except group fields.
ffNoExport	If set, the field must not be exported by a submit-form action (see CtreateSubmitAction() for further information). Supported by all fields except group fields.
ffMultiline	If set, the field may contain multiple lines of text; if clear, the field's text is restricted to a single line. Supported by button fields, text fields.

Flag	Description
`ffPassword`	If set, the field is intended for entering a secure password that should not be echoed visibly to the screen. Characters typed from the keyboard should instead be echoed in some unreadable form, such as asterisks or bullet characters. To protect password confidentiality, the value of the text field is not stored in the PDF file if this flag is set. Supported by text fields only.
`ffNoToggleToOff`	If set, exactly one radio button must be selected at all times; clicking the currently selected button has no effect. If clear, clicking the selected button deselects it, leaving no button selected. Supported by radio button fields only.
`ffEdit`	If set, the combo box includes an editable text box as well as a drop list; if clear, it includes only a drop list. Supported by combo boxes only.
`ffSorted`	If set, the field values are sorted in ascending order. Supported by combo boxes and list boxes only.
`ffFileSelect`	*(PDF 1.4)* If set, the text entered in the field represents the pathname of a file whose contents are to be submitted as the value of the field. Supported by text fields only.
`ffMultiSelect`	*(PDF 1.4)* If set, more than one of the field 's option items may be selected simultaneously; if clear, no more than one item at a time may be selected. This flag is supported by list boxes only.
`ffDoNotSpellCheck`	*(PDF 1.4)* If set, the text entered to the field will not be spell-checked. Supported by text fields, combo boxes. If the field type is combo box, this flag is meaningful only if the flag ffEdit is also set.
`ffDoNotScroll`	*(PDF 1.4)* If set, the field will not scroll (horizontally for single-line fields, vertically for multiple-line fields) to accommodate more text than will fit within its annotation rectangle. Once the field is full, no further text will be accepted. Supported by text fields only.
`ffComb`	*(PDF 1.5)* Meaningful only if MaxLen is set (see CreateTextField()) and if the ffMultiline, ffPassword, and ffFileSelect flags are clear. If set, the field is automatically divided up into as many equally spaced positions, or combs, as the value of MaxLen, and the text is laid out into those combs. Supported by text fields only.

Flag	Description
ffCommitOnSelCh	*(PDF 1.5)* If set, the new value is committed as soon as a selection is made with the pointing device. This allows applications to perform an action once a selection is made, without requiring the user to exit the field. If clear, the new value is not committed until the user exits the field. Supported by combo boxes and list boxes only.
ffRadioIsUnion	*(PDF 1.5)* If set, a group of radio buttons within a radio button field that use the same export value for the on state will turn on and off in unison; that is, if one is checked, they are all checked. This flag requires Acrobat 6 or higher and is supported by check boxes only.

SetFieldFont

```
SI32 pdfSetFieldFont(
   const PPDF* IPDF, // Instance pointer
   UI32 Field,       // Field handle
   const char* Name, // Font name (depends on the font selection mode)
   TFStyle Style,    // Font style
   double Size,      // Font size or zero for auto size
   LBOOL Embed,      // If true, the font will be embedded
   TCodepage CP)     // Code page cp1252 or cpUnicode are recommended
```

The function sets or changes the font of a form field. Form fields support natively the code pages 1252 and MacRoman only.

If a form field should be used with another code page then DynaPDF must create an Unicode based font for the field.

Subject to change:

This is only possible if the following external cmaps are available:

Minimal requirement:

```
UniJIS-UTF16-H    // Encoding
Adobe-Japan1-UCS2 // Decoding
```

If OpenType fonts should be supported then these additional cmaps must be available:

```
UniGB-UTF16-H
Adobe-GB1-UCS2
UniKS-UTF16-H
Adobe-Korea1-UCS2
```

You find the above cmaps in the sub directory /Resource/CMap of the DynaPDF installation folder. The cmaps can be loaded delayed with SetCMapDir().

If no external cmaps are available then DynaPDF creates the appearance stream of the field with the specified font but the field font will be set to Helvetica! Although DynaPDF still creates a PDF file in this case do not assume that the resulting file is usable!

If you don't want to use external cmaps then don't use another code page than 1252 or MacRoman with form fields!

DynaPDF checks whether the field font supports all glyphs which are used in the field when the file is closed. If one or more glyphs are missing, then the font will be replaced if DynaPDF can find a better match that supports all glyphs.

If the font of several fields must be set to the same font, then store the returned field handle and set the font with SetFieldFontEx().

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Unicode version is preferred since font names are usually stored in Unicode format.

Field fonts and fonts which are used on a page are stored in different arrays. Therefore, a field font handle that was returned by this function cannot be used with ChangeFont() and a font handle that was returned by SetFont() cannot be used with SetFieldFontEx().

Return values:

If the function succeeds the return value is the font handle, a value greater zero. If the function fails the return value is a negative error code.

SetFieldFontEx

Syntax:

```
LBOOL pdfSetFieldFontEx(
   const PPDF* IPDF,
   UI32 Field,
   UI32 Handle,
   double FontSize)
```

The function sets or replaces the font of a field. The function requires a font handle that was returned by SetFieldFont().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldFontSize

Syntax:

```
LBOOL pdfSetFieldFontSize(
   const PPDF* IPDF, // Instance pointer
   UI32 AField,      // Field handle
   double FontSize)  // New font size
```

The function changes the font size of a specific field. A value of 0.0 is used as auto size. The optimal font size is then calculated by DynaPDF.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldHighlightMode

Syntax:

```
LBOOL pdfSetFieldHighlightMode(
   const PPDF* IPDF,    // Instance pointer
   UI32 AField,         // Field handle
   THighlightMode Mode) // New highlight mode
```

The function changes the highlight mode of a specific field. Supported field types are buttons, checkboxes, radio buttons, and signature fields. Other field types do not support this property.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldIndex

Syntax:

```
LBOOL pdfSetFieldIndex(
   const PPDF* IPDF, // Instance pointer
   UI32 AField,      // Field handle
   UI32 Index)       // New field index (this represents the tab order)
```

This function can be used to set the tab order of interactive form fields. All fields, independently whether they are created with DynaPDF or imported from external documents, holds in internal index which can be used to reorder or sort the fields with this index.

The start index of a new field is not zero, it is 1000 instead. This makes it easier to set a field in front of all other fields without changing all field indices. However, changing a field index changes not the tab order directly, the fields of a page must be sorted before the page will be closed with the function SortFieldByIndex().

The tab order of interactive form fields must be set for each page separately.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (C++):

In this example we create 50 text fields in two columns. The order in which the fields are created is line by line, not column by column. Execute the example and open the file with Acrobat or Reader. You can see now that the tab order is column by column, not line by line as the fields were created. Comment out the SortFieldsByIndex() function call and create the file again; the tab order is now line by line.

Take a look onto the for-statement; we set the field index of the text fields of the first column only. The other fields can be left unchanged because their index lies between 1001 and 1049.

```cpp
#include "dynapdf.h"
using namespace DynaPDF;
// First we declare our error callback function
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}
int main(int argc, char* argv[])
{
   char tmp[30]; double y; SI32 field, index;
   PPDF* pdf = pdfNewPDF(); // Create a PDF instance
   if (!pdf) return 2;      // Out of memory?
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfSetDocInfo(pdf, diCreator, "C++ sample project");
   pdfSetDocInfo(pdf, diSubject, "Tab order");
   pdfSetDocInfo(pdf, diCreator, "Tab order");

   pdfCreateNewPDF(pdf, "c:/cppout.pdf");
   pdfSetPageCoords(pdf, pcTopDown);
   pdfAppend(pdf);
   y = 50.0; index = 0;
   for (SI32 i = 0; i < 50; i++)
   {
      sprintf(tmp, "Field %d", i);
      if (i & 1)
      {
         // This is the second column, the field indices can be left
         // unchanged because they are above 1000.
         pdfCreateTextField(pdf, tmp, -1, false, 0, 210, y, 150, 20);
         y += 25.0; // goto the next line
      }else
      {
         field = pdfCreateTextField(pdf, tmp, -1, false,0,50,y,150,20);
```

```
        pdfSetFieldIndex(pdf, field, index++);
      }
   }
   pdfSortFieldsByIndex(pdf);
   pdfEndPage(pdf);
   pdfCloseFile(pdf);
   pdfDeletePDF(pdf); // Do not forget to delete the PDF instance
}
```

SetFieldMapName

Syntax:

```
LBOOL pdfSetFieldMapName(
   const PPDF* IPDF, // Instance pointer
   UI32 AField,      // Field handle
   const char* Name) // Mapping name (NULL to delete it)
```

The function sets or changes the mapping name of a field. The mapping is used when exporting interactive form field data from the document. The parameter *AField* must be a field handle. If the mapping name of the field should be deleted set the parameter *Name* to NULL. This function is also available in a Unicode compatible version. However, a mapping name should be defined as an Ansi string.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldName

Syntax:

```
LBOOL pdfSetFieldName(
   const PPDF* IPDF,     // Instance pointer
   UI32 AField,          // Field handle
   const char* NewName)  // New field name
```

The function changes the name of an interactive form field. Field names must sometimes be changed when multiple interactive forms with identical field names are imported. The resulting form will be damaged if the names of such fields are not changed.

The function does not check whether a field name is already in use. Such a check would not be useful, because the usage of the function would be much more complicated. Use the function CheckFieldNames() to check the integrity of the form after field names were changed.

Remarks:

Changing field names can cause problems if the field is used in a JavaScript action or global JavaScript. The JavaScript(s) must also be changed to avoid error messages in Adobe's Acrobat. Global JavaScripts can be accessed with the function GetJavaScript() and changed with the function

ChangeJavaScript(). A JavaScript Action can be accessed with the function GetJavaScriptAction() and changed with the function ChangeJavaScriptAction().

Remarks:

This function is available in an Ansi and Unicode compatible version. Unicode field names are supported since PDF 1.5 (Acrobat 6).

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldOrientation

Syntax:

```
LBOOL pdfSetFieldOrientation(
    const PPDF* IPDF, // Instance pointer
    UI32 AField,      // Field handle
    SI32 Value)       // Orientation in degrees (must be a multiple of 90)
```

The function sets or changes the orientation of a field. The parameter *AField* be a valid field handle. The parameter *Value* must a multiple of 90 or 0. Positive values rotate the field counter clockwise, negative values clockwise.

The function can also be used to rotate a *Caret* and *FreeText* annotation. The annotation handle must be added to the constant PDF_ANNOT_INDEX in this case because annotations are stored in a different array. Only caret and free text annotations can be rotated, all other annotation types do not support this feature.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldTextAlign

Syntax:

```
LBOOL pdfSetFieldTextAlign(
    const PPDF* IPDF, // Instance pointer
    UI32 AField,      // Text field or button field handle
    TTextAlign Align) // New alignment

typedef enum
{
    taLeft,
    taCenter,
    taRight,
    taJustify
}TTextAlign;
```

The function set or changes the text alignment of a text or button field. The parameter *AField* must be a valid handle of a text field or button field.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldTextColor

Syntax:

```
LBOOL pdfSetFieldTextColor(
   const PPDF* IPDF, // Instance pointer
   UI32 Color)       // Text color defined in the current color space
```

The function sets the text color which is used for newly created interactive form fields. The color value must be defined in the current color space. The color space must not be changed before the fields are created which should use this color.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFieldToolTip

Syntax:

```
LBOOL pdfSetFieldToolTip(
   const PPDF* IPDF,  // Instance pointer
   UI32 AField,       // Field handle
   const char* Value) // Tool tip
```

The function set or changes the tool tip or description string of an interactive form field. The parameter *AField* must be a valid field handle.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252 only. To create a description string in an arbitrary encoding convert the string to Unicode with the function ConvToIncode() first and use the Unicode version to apply the string.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFillColor

Syntax:

```
LBOOL pdfSetFillColor(
   const PPDF* IPDF, // Instance pointer
   UI32 Color)       // Color value defined in the current color space
```

The function sets the fill color. The parameter *Color* must be defined in the current color space. For example, if the current color space is DeviceGray the color value must be in the range 0 to 255.

CMYK colors can be constructed with the macro PDF_CMYK() or with the function CMYK() which is available in most programming languages. RGB colors can be constructed with the macro PDF_RGB() or with the function RGB() which is available in most programming languages.

If the corresponding color space contains more than four color components use SetFillColorEx() instead.

The function requires an open page, template, or pattern.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFillColorEx

Syntax:

```
LBOOL pdfSetFillColorEx(
   const PPDF* IPDF,    // Instance pointer
   const BYTE* Color,   // Color to be set
   UI32 NumComponents)  // Number of componnents
```

The function sets the fill color. The color must be defined as an array of bytes in the logical order of the color space. The number of components must match the number of components of the underlying color space.

Lab colors can be defined as signed char as usual. Make a typecast to BYTE* when passing the color to the function. See CreateCIEColorSpace() for further information.

Example (C/C++):

```
...
char labColor[3] = {50, -34, 77}; // L, *a, *b
pdfSetFillColor(pdf, (BYTE*)labColor, 3);
...
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFillColorF

Syntax:

```
LBOOL pdfSetFillColorF(
   const PPDF* IPDF,    // Instance pointer
   const float* Color,  // Array of float values
   UI32 NumComponents)  // Must match the underlying color space
```

The function sets the current fill color as an array of float values. The components of non-Lab color spaces must be in the range from 0 through 1. The *a and *b components of a Lab color space are typically in a range -128 though 127. The *L component ranges from 0 through 100.

The number of components must match the number of components of the underlying color space.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFillColorSpace

Syntax:

```
LBOOL pdfSetFillColorSpace(
   const PPDF* IPDF,  // Instance pointer
   TPDFColorSpace CS) // Color space
```

The function changes the fill color space. In PDF, fill and stroke colors use both their own color spaces. Although it is possible to use different color spaces for strokes and fillings it should be avoided if possible. The fill color space is the relevant color space when creating interactive objects such as form field or annotations. See also SetStrokeColorSpace().

Notice:

This function was added to DynaPDF primarily for testing purposes. The color space should be set with SetColorSpace() which sets always the same color space for fillings and strokes.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFloatPrecision

Syntax:

```
LBOOL pdfSetFloatPrecision(
   const PPDF* IPDF,        // Instance pointer
   UI32 NumTextDecDigits, // Number decimal digits for text object
   UI32 NumVectDecDigits) // Number of decimal digits for vector objects
```

The function changes the output precision of text and vector coordinates. The default precision for text and vector graphics is two decimal digits. This corresponds to 7200 DPI. The output precision of bezier curves is one higher as for vector graphics. The maximum output precision is 5 decimal digits.

Note that higher values increase the resulting file size.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFont

Syntax:

```
SI32 pdfSetFont(
    const PPDF* IPDF, // Instance pointer
    const char* Name, // Font name or NULL (see note below)
    TFStyle Style,    // Font style
    double Size,      // Font size
    LBOOL Embed,      // If true, the font is embedded
    TCodepage CP)     // Code page

typedef SI32 TFStyle;
#define fsNone          0x00000000 // Obsolete, use fsRegular instead
#define fsItalic        0x00000001
#define fsUnderlined    0x00000004
#define fsStriked       0x00000008
#define fsVerticalMode  0x00000010 // Not considered at this time
#define fsDblUnderline  0x00000020
// Width Class (defined for future use, ignored at this time)
#define fsUltraCondensed 0x00000100 // 1
#define fsExtraCondensed 0x00000200 // 2
#define fsCondensed      0x00000300 // 3
#define fsSemiCondensed  0x00000400 // 4
#define fsNormal         0x00000500 // 5
#define fsSemiExpanded   0x00000600 // 6
#define fsExpanded       0x00000700 // 7
#define fsExtraExpanded  0x00000800 // 8
#define fsUltraExpanded  0x00000900 // 9
// Weight Class
#define fsThin          0x06400000 // 100
#define fsExtraLight    0x0C800000 // 200
#define fsLight         0x12C00000 // 300
#define fsRegular       0x19000000 // 400
#define fsMedium        0x1F400000 // 500
#define fsDemiBold      0x25800000 // 600
#define fsBold          0x2BC00000 // 700 -> The old constant 2 is still supported to preserve
backward compatibility
#define fsExtraBold     0x32000000 // 800
#define fsBlack         0x38400000 // 900
#define fsUltraBlack    0x3E800000 // 1000

typedef enum
{
    cp1250,             // Eastern European
    cp1251,             // Cyrillic
    cp1252,             // Latin 1 Western European
    cp1253,             // Greek
    cp1254,             // Turkish
    cp1255,             // Hebrew
    cp1256,             // Arabic
    cp1257,             // Baltic
    cp1258,             // Vietnamese
    cp8859_2,           // Latin 2 Central Europe
    cp8859_3,           // Latin 3 Maltese
    cp8859_4,           // Baltic
    cp8859_5,           // Cyrillic
    cp8859_6,           // Arabic
    cp8859_7,           // Greek
    cp8859_8,           // Hebrew
    cp8859_9,           // Latin 5 Turkish
    cp8859_10,          // Latin 6 Nordic Area
    cp8859_13,          // Latin 7 Baltic Rim
    cp8859_14,          // Latin 8 Celtic
    cp8859_15,          // Latin 9 French
```

```
    cp8859_16,          // Latin 10 Hungarian
    cpSymbol,           // Symbol
    cp437,              // DOS USA
    cp737,              // DOS Greek
    cp775,              // DOS Baltic Rim
    cp850,              // DOS Multilingual
    cp852,              // DOS Slavic
    cp855,              // DOS Cyrillic
    cp857,              // DOS Turkish
    cp860,              // DOS Portuguese
    cp861,              // DOS Icelandic
    cp862,              // DOS Hebrew
    cp863,              // DOS French (Canada)
    cp864,              // DOS Arabic
    cp865,              // DOS Nordic
    cp866,              // DOS Russian
    cp869,              // DOS Modern Greek
    cp874,              // DOS Thai
    cpUnicode,          // Unicode without Uniocde algorithms
    cpCJK_Big5_Uni,     // Big5 plus HKSCS extension
    cpCJK_EUC_JP_Uni,   // EUC-JP
    cpCJK_EUC_KR_Uni,   // EUC-KR
    cpCJK_EUC_TW_Uni,   // CNS-11643-1992 (Planes 1-15)
    cpCJK_GBK_Uni,      // MS code page 936 (GB2312, EUC-CN plus GBK)
    cpCJK_GB12345_Uni,  // GB-12345-1990 (Trad. Chinese form of GB-2312)
    cpCJK_HZ_Uni,       // Mixed ASCII / GB-2312 encoding
    cpCJK_2022_CN_Uni,  // ISO-2022-CN-EXT (GB-2312 plus ISO-11643)
    cpCJK_2022_JP_Uni,  // ISO-2022-JP
    cpCJK_2022_KR_Uni,  // ISO-2022-KR
    cpCJK_646_CN_Uni,   // ISO-646-CN (GB-1988-80)
    cpCJK_646_JP_Uni,   // ISO-646-JP (JIS_C6220-1969-RO)
    cpCJK_IR_165_Uni,   // ISO-IR-165 (extended version of GB-2312)
    cpCJK_932_Uni,      // Microsoft extended version of SHIFT_JIS
    cpCJK_949_Uni,      // EUC-KR extended with UHC (Unified Hangul Codes)
    cpCJK_950_Uni,      // Microsoft extended version of Big5
    cpCJK_JOHAB_Uni,    // JOHAB
    cpShiftJIS,         // Native CJK character set requires a CJK font
    cpBig5,             // Native CJK character set requires a CJK font
    cpGB2312,           // Native CJK character set requires a CJK font
    cpWansung,          // Native CJK character set requires a CJK font
    cpJohab             // Native CJK character set requires a CJK font
    cpMacRoman,         // Mac Roman
    cpAdobeStd,         // Special encoding for Type1 fonts; should not used.
    cpInternal,         // Internal -> not usable
    cpGlyphIndexes,     // TrueType and OpenType fonts only
    cpPDFDocEnc,        // Internal -> not usable
    cpDingbats          // Internal -> not usable
}TCodepage;
```

The function loads a font that can be used for text output and interactive form fields. The parameter *Name* must be either the Family, Full, or PostScript Name depending on the current font selection mode (see SetFontSelMode() for further information).

It is also possible to set the font name to NULL to deactivate the active font. This can be useful when creating form fields. It means, use the default font of the global AcroForm object.

Fonts are looked up in the search directories which can be set with AddFontSearchPath(). On Windows and Mac OS X operating systems DynaPDF adds automatically the default font directories of the system to the list of font search paths. See SetUseSystemFonts() for further information.

Font enumeration

The available system fonts can be enumerated with GetSysFontInfo().

Font Search Order

DynaPDF searches for fonts in a specific order which can be changed with SetFontSearchOrder() or SetFontSearchOrderEx(). The default search order is:

- TrueType, TrueType Collection, OpenType fonts with TrueType outlines
- OpenType fonts with Postscript outlines
- Type1
- Standard PDF fonts

OpenType fonts with TrueType outline are treated like ordinary TrueType fonts. All supported font formats can be explicitly disabled if necessary. See SetFontSearchOrder() for further information.

Font names

All font types support different naming schemes: family and full names are normally used on Windows and the Postscript name is widely used on Linux and UNIX operating systems. Font names are case-sensitive, they must be specified exactly. The naming sheme that should be used for font selection can be set with SetFontSelMode() (the default is smFamilyName).

Family name

The family name is the typeface of a font, it refers to the member lfFaceName of the LOGFONT structure on Windows (see GetLogFont() for further information). Family names are stored in Unicode format in TrueType and OpenType fonts. However, most available fonts can be selected with the Ansi version of SetFont() too because a usual font name contains no special characters.

A family name is not unique; it specifies the font family and not a specific font of a given style.

The combination of the family name and the font style identifies a font. For example, the font Arial is available in several different styles such as Regular, Bold, Italic, or BoldItalic. The style information Bold, Italic, and so on is **not** part of the font name. That is the reason why the style is a separate parameter of SetFont().

Each font style is physically stored in a separate font file. However, if a requested style is not available, SetFont() emulates the missing style if a compatible variant can be found. The emulation of bold and italic font styles can be disabled with SetFontWeight() and SetItalicAngle().

Full name

The full name is a unique font name that identifies a font exactly. The full name is a combination of the family name plus the style name, e.g. "Arial Bold Italic". This is the name that Windows exposes to users. The parameter *Style* is ignored when selecting fonts via the full name.

TrueType and OpenType fonts can contain many localized full names. DynaPDF loads all these language variants so that they can be used independent of the current locale.

PostScript name

The postscript name should be a unique name that identifies a font exactly. However, it is known that certain font vendors deliver fonts with incorrectly named style variants, i.e. the same postscript name can be assigned to condensed and expanded versions. In such a case the postscript name is not unique and avoids correct font selection.

It is usually better to select fonts via the full name since this name seems to be unique in most fonts.

Font Styles

The parameter *Style* specifies the font style variant that should be loaded from a given font family. A font style is a 32 bit unsigned integer that is encoded as follows:

- `Bits  0..8  // Style bits fsItalic, fsUnderlined, fsStriked`
- `Bits  9..19 // Width class (1..9) -> Defined for future use`
- `Bits 20..31 // Font Weight (100..1000)`

The width class and font weight can be converted to a style constant with the macros WidthToStyle() and WeightToStyle():

```
#define WidthToStyle(w)  ((w) << 8)
#define WeightToStyle(w)  ((w) << 20)
// Corresponding macros to extract the values
#define WidthFromStyle(s) (((s) & 0x00000F00) >> 8)
#define WeightFromStyle(s)(((s) & 0xFFF00000) >> 20)
```

Example:

```
// Width class 5 (Normal), font weight 700 (bold), italic
TFStyle style = WidthToStyle(5) | WeightToStyle(700) | fsItalic;
// or use the defined constants instead
TFStyle style = fsNormal | fsBold | fsItalic;
```

A valid font weight is a multiple of 100 or 100. Values like 450 or 770 are not meaningful.

Supported font formats

The following font formats are supported by DynaPDF (on Mac OS X the same formats are also supported if they are stored in resource or data forks, as well as in .dfont or .suit files):

- TrueType fonts (*.ttf). MAC Standard, Unicode or CJK encoded fonts.
- TrueType Collection (*.ttc). These fonts contain multiple fonts into one.
- OpenType fonts (*.otf) with TrueType or PostScript outlines.
- PostScript Type 1 fonts (*.pfb or *.pfa). No metric files are required.

Font Subsetting

When a TrueType, OpenType or a font of a TrueType Collection will be embedded, DynaPDF creates a subset of this font which contains the used characters only. The function constructs a completely new font file that contains no unnecessary information.

Type 1 fonts are always stored unchanged in the PDF file. Font subsetting is not supported for this font format.

Font Embedding

All supported font formats can be embedded as long as font embedding is not restricted in the font file. Font embedding can be disabled when using the code page 1252 or MacRoman, but it is required if another code page is used; the parameter *Embed* is ignored in this case. If font embedding is required but not possible due to licensing restrictions the function will fail.

However, when creating Asian PDF files it is possible to use CID-keyed fonts instead. The CID-Keyed Font Architecture works with embedded and non-embedded OpenType and TrueType fonts. See SetCIDFont() for further information.

The 14 Standard Fonts

All versions of Adobe's Acrobat support 14 standard fonts. These fonts are always available independent whether they're embedded or not.

Family name	Full name	PostScript name	Style
Courier	Courier	Courier	fsRegular
Courier	Courier Bold	Courier-Bold	fsBold
Courier	Courier Oblique	Courier-Oblique	fsItalic
Courier	Courier Bold Oblique	Courier-BoldOblique	fsBold + fsItalic
Helvetica	Helvetica	Helvetica	fsRegular
Helvetica	Helvetica Bold	Helvetica-Bold	fsBold
Helvetica	Helvetica Oblique	Helvetica-Oblique	fsItalic
Helvetica	Helvetica Bold Oblique	Helvetica-BoldOblique	fsBold + fsItalic
Times	Times Roman	Times-Roman	fsRegular
Times	Times Bold	Times-Bold	fsBold
Times	Times Italic	Times-Italic	fsItalic
Times	Times Bold Italic	Times-BoldItalic	fsBold + fsItalic
Symbol	Symbol	Symbol	fsRegular
ZapfDingbats	ZapfDingbats	ZapfDingbats	fsRegular

DynaPDF includes the metrics of these fonts only. Please note that standard fonts have the lowest search priority. If a standard font is available in one of the font search paths then this version will be used. The font search order can be changed with SetFontSearchOrder().

Code pages versus character sets

The following table specifies which code page refers to which Windows character set.

Code page	Windows character set
cp1250	EASTEUROPE_CHARSET
cp1251	RUSSIAN_CHARSET
cp1252	Ansi_CHARSET
cp1253	GREEK_CHARSET
cp1254	TURKISH_CHARSET
cp1255	HEBREW_CHARSET
cp1256	ARABIC_CHARSET
cp1257	BALTIC_CHARSET
cp1258	VIETNAMESE_CHARSET
cp8859_2	EASTEUROPE_CHARSET
cp8859_3	TURKISH_CHARSET
cp8859_4	BALTIC_CHARSET
cp8859_5	RUSSIAN_CHARSET
cp8859_6	ARABIC_CHARSET
cp8859_7	GREEK_CHARSET
cp8859_8	HEBREW_CHARSET
cp8859_9	TURKISH_CHARSET
cp8859_10	BALTIC_CHARSET
cp8859_13	BALTIC_CHARSET
cp8859_14	Ansi_CHARSET
cp8859_15	Ansi_CHARSET
cp8859_16	EASTEUROPE_CHARSET
cpSymbol	SYMBOL_CHARSET
cp437	Ansi_CHARSET
cp737	GREEK_CHARSET
cp775	BALTIC_CHARSET
cp850	Ansi_CHARSET
cp852	EASTEUROPE_CHARSET
cp855	RUSSIAN_CHARSET
cp857	TURKISH_CHARSET
cp860	Ansi_CHARSET
cp861	Ansi_CHARSET
cp862	HEBREW_CHARSET
cp863	Ansi_CHARSET
cp864	ARABIC_CHARSET
cp865	Ansi_CHARSET
cp866	RUSSIAN_CHARSET
cp869	GREEK_CHARSET
cp874	VIETNAMESE_CHARSET
cpUnicode	DEFAULT_CHARSET
cpCJK_Big5_Uni	DEFAULT_CHARSET
cpCJK_EUC_JP_Uni	DEFAULT_CHARSET
cpCJK_EUC_KR_Uni	DEFAULT_CHARSET
cpCJK_EUC_TW_Uni	DEFAULT_CHARSET

```
cpCJK_GBK_Uni          DEFAULT_CHARSET
cpCJK_GB12345_Uni      DEFAULT_CHARSET
cpCJK_HZ_Uni           DEFAULT_CHARSET
cpCJK_2022_CN_Uni      DEFAULT_CHARSET
cpCJK_2022_JP_Uni      DEFAULT_CHARSET
cpCJK_2022_KR_Uni      DEFAULT_CHARSET
cpCJK_646_CN_Uni       DEFAULT_CHARSET
cpCJK_646_JP_Uni       DEFAULT_CHARSET
cpCJK_IR_165_Uni       DEFAULT_CHARSET
cpCJK_932_Uni          DEFAULT_CHARSET
cpCJK_949_Uni          DEFAULT_CHARSET
cpCJK_950_Uni          DEFAULT_CHARSET
cpCJK_JOHAB_Uni        DEFAULT_CHARSET
cpShiftJIS             SHIFTJIS_CHARSET
cpBig5                 CHINESEBIG5_CHARSET
cpGB2312               GB2312_CHARSET
cpWansung              HANGUL_CHARSET
cpJohab                HANGUL_CHARSET
cpMacRoman             MAC_CHARSET
```

How to use CJK encodings?

DynaPDF supports two types of CJK fonts: Native CJK fonts which support native CJK character sets such as cpShiftJIS, cpGB2312, cpWansung, and so on, and Unicode fonts which contains CJK characters.

Pure Unicode fonts support, as the name suggests, Unicode code points and no mixed 8/16 bit CJK encodings. Mixed 8/16 bit multi-byte strings must be converted to Unicode before they can be used with Unicode fonts. This conversion is automatically applied when a code page is used which ends with a "_Uni″, e.g. cpCJK_Big5_Uni or cpCJK_932_Uni. The "CJK" in the code page name indicates that mixed 8/16 bit CJK input character sequences must be used with this encoding.

The "_Uni" at the end of the name means that this code page can be used with a Unicode font. CJK to Unicode conversion algorithms are available in the Ansi versions of string methods only. This is normally no limitation because CJK strings are usually not defined as wide-strings.

The other CJK code pages which are available in DynaPDF are those without the "_Uni" at the end of the name, such as cpShiftJIS, cpGB2312, and so on. These code pages represent character sets and no code pages. The usage of a pure CJK encoding requires that the TrueType or OpenType font contains a CMap in format 2. This is mostly the case in pure CJK fonts (fonts which provide no Unicode based CMap). Unlike the CJK to Unicode code pages, these character sets can be used with the Ansi and wide string versions of a text method. Native character sets are faster than Unicode based versions because no string conversion is required.

In addition to the above code pages and character sets DynaPDF supports also the CID-Keyed font architecture which was specifically designed to process Asian languages. See SetCIDFont() for further information.

GDI Font selection in comparison to DynaPDF

The Windows GDI selects a font in another way than DynaPDF does. Let us see how a font can be selected with the GDI function CreateFont() and where are the differences in comparison to DynaPDF:

```
HFONT f = CreateFont(
   -240,                 // height of font
   0,                    // average character width
   0,                    // angle of escapement
   0,                    // base-line orientation angle
   500,                  // font weight
   1,                    // italic attribute option
   0,                    // underline attribute option
   0,                    // strikeout attribute option
   Ansi_CHARSET,         // character set identifier
   OUT_TT_ONLY_PRECIS,   // output precision
   CLIP_DEFAULT_PRECIS,  // clipping precision
   ANTIALIASED_QUALITY,  // output quality
   FF_DONTCARE,          // pitch and family
   "Arial")              // typeface name (Family Name)
```

nHeight

The GDI supports three modes how the font size or height can be specified. We want to discuss negative values only because the other modes are not of interest for high precision output (positive values specify the cell height of a font, this value must be converted by DynaPDF to determine the wished font size). The font height we used in the example CreateFont() function call is defined as a negative value. The absolute value of it is transformed to device units by the font mapper. We choose a size of 240 units; this is a normal font size for a 1440 DPI output.

A negative font height in a CreateFont() function call is exactly the same as a positive height or size in a SetFont() function call in DynaPDF (negative values are not supported by SetFont()).

However, GDI coordinates are normally upscaled because GDI functions support integer values only. For example, if we want to use a font size of 12.74 units, we can only upscale the coordinate system because it is impossible to define such a font size with a GDI function. That is the reason why GDI drawings are defined in a specific resolution. GDI coordinates are never really device independent.

However, to avoid issues with a fixed resolution the PDF format uses a completely different system to produce device independent output. PDF coordinates and font sizes are expressed in floating point units so that we can use our font size of 12.74 units directly. We do not need to upscale the coordinate system and we don't need to consider whether the PDF file is drawn on a monitor or printer. All the details about the output resolution are completely hidden for the user.

BTW - A PDF unit is 1/72 inch.

nWidth

The average character width (nWidth) is the same as horizontal text scaling in PDF (see SetTextScaling()), but the calculation is completely different. The GDI uses the average character width of a font to calculate the text scaling (the unscaled value is returned by the GDI function GetTextMetrics() if nWidth was set to zero during font selection). So, the font must be selected into device context twice to calculate a text scaling, one time to determine the unscaled value, and a second time to select the font with the wished value.

If a text should be scaled to 120% horizontally the calculation is as follows (C++):

```
TEXTMETRIC tm;
GetTextMetrics(dc, &tm);
int nWidth = tm.AveCharWidth * 120 / 100;
```

Note that text scaling is an output attribute; the value is not used during font selection.

In PDF, text scaling is a separate property of the current graphics state so that you don't need to determine the average character width of a font beforehand. Text scaling can be applied directly with the function SetTextScaling() without any further calculation. So, if you want to scale a text to 120% in PDF, you can simply call the function SetTextScaling() as follows:

```
pdfSetTextScaling(pdf, 120.0);
```

nEscapement

The angle of escapement specifies the output angle of a text string in tenths degrees. The output angle of a string has nothing to do with a font, it is an output attribute. Rotated text can be printed with the function WriteAngleText() with DynaPDF.

nOrientation

This parameter specifies the angle, between each character's base line and the x-axis of the device. DynaPDF does not support a function that enables printing of characters in this way. Each character must be printed separately with the function WriteAngleText() to get the same result.

nWeight

The font weight can be in the range 0 to 1000 (zero means don't know). A font can be installed in up to 10 different font weights and each weight refers to another font file. The weight is encoded in the parameter *Style* of a SetFont() call. The helper function WidthToStyle() converts a weight to a TFStyle constant. A valid font weight is dividable by 100. A value like 345 is no valid font weight while 300 or 400, for example, would be correct.

The style flags underlined and strikeout are supported by SetFont() too. The only difference is that each flag can be set with one parameter because the parameter Style of SetFont() is a bit-mask. See sesction Font Styles for further information.

fdwCharSet

A character set is not what a user types in with the keyboard; it specifies a character collection or an alphabet that the font must support. The charset has the highest search priority, followed by fdwPitchAndFamily. Font selection fails if no font can be found that supports the selected character set, but it does not fail if the font name is wrong.

This is the major difference in comparison to DynaPDF since the font name has the highest search priority in DynaPDF. The character set or code page parameter is not used for font selection.

The next difference is that the GDI does not automatically map code page code points to Unicode, this must be done manually. This step is not required when working with DynaPDF; the required mapping is automatically applied.

fdwPitchAndFamily

The pitch and family specifies whether a proportional or fixed pitch font should be selected. This parameter should be set to FF_DONTCARE because the pitch has a higher search priority as the font name. A wrong value of this parameter avoids the selection of the wished font by the GDI.

SetFont() supports no corresponding parameter or style flag.

lpszFace

This can be the family or full name of a font. The GDI does not support postscript names. The font name has the lowest search priority in the GDI while it has the highest in DynaPDF.

SetFont() needs to know what kind of name is supplied. The wished mode must be set beforehand with SetFontSelMode(). SetFont() can select fonts via family name, full name, or postscript name.

If all three names should tested then use SetFontEx() instead.

Remarks:

The function SetFont() is implemented in an Ansi and Unicode compatible version. Family and full names are originally defined in Unicode so that this version is preferred when the font selection mode is set to smFamilyName (default) or smFullName.

All string functions report a warning if one or more glyphs cannot be found. Call GetMissingGlyphs() to determine which characters could not be found.

Return values:

If the function succeeds the return value is the font handle, a value greater or equal zero. If the function fails the return value is a negative error code.

SetFontEx

Syntax:

```
SI32 pdfSetFontEx(
   const PPDF* IPDF, // Instance pointer
   const char* Name, // Font name or NULL (see note below)
   TFStyle Style,    // Font style
   double Size,      // Font size
   LBOOL Embed,      // If true, the font is embedded
   TCodepage CP)     // Code page
```

The function loads a font that can be used for text output and interactive form fields. The difference in comparison to SetFont() is the way how the function tries to find the font.

If the font style is <> fsNone the function searches the font first by family name, then by full name, and finally by postscript name if the font was not already found.

If the font style is set to fsNone the function searches the font first by full name, then by postscript name, and finally by family name if the font was not already found.

It is also possible to set the font name to NULL to deactivate the active font. This can be useful when creating form fields. It means: use the default font of the global AcroForm object for new fields.

The search runs are non-case-sensitive and spaces in the font name will be ignored.

This function can be used if a PDF font should be loaded or if it is not known whether the given font name represents a family, full, or postscript name. See also SetFont().

Return values:

If the function succeeds the return value is the font handle, a value greater or equal zero. If the function fails the return value is a negative error code.

SetFontOrigin

Syntax:

```
LBOOL pdfSetFontOrigin(
    const PPDF* IPDF, // Instance pointer
    TOrigin Origin)   // New origin

typedef enum
{
  orDownLeft = 0,
  orTopLeft  = 1
}TOrigin;
```

The function sets the current font origin that is used to position text strings. The font origin is automatically set to orTopLeft or orDownLeft if the coordinate system will be changed (see SetPageCoords() for further information).

```
Default value = orDownLeft
```

Font origin:

0|0

The origin is top left

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetFontSearchOrder

Syntax:

```
void pdfSetFontSearchOrder(
    const PPDF* IPDF,
    TFontBaseType Order[4]) // 4 element array or NULL

typedef enum
{
    fbtTrueType, // TrueType, TrueType Collections
    fbtType1,    // Type1 font
    fbtOpenType, // OpenType font with Postscript outlines
    fbtStdFont,  // PDF Standard font
    fbtDisabled  // Can be used to disable a font format
}TFontBaseType;
```

The function changes the font search order. The default search order is:

- TrueType
- OpenType (OpenType fonts with PostScript outlines)
- Type1
- PDF Standard Fonts

Every font type can be disabled if necessary with the value fbtDisabled. The array *Order* can be set to NULL to reset the search order to the default order as described above. If *Order* is set, the array must contain four elements and each value in the array must be properly initialized.

Example (C/C++):

```
TFontBaseType so[4] = {fbtStdFont, fbtTrueType, fbtOpenType, fbtType1};
// Change the search order
pdfSetFontSearchOrder(pdf, so);
...
// Reset the font search order to the default order
pdfSetFontSearchOrder(pdf, NULL);
```

Compatibility Note:

The default font search order in DynaPDF 2.0 was:

- Standard PDF Fonts
- TrueType or Type1

DynaPDF 2.0 selected simply the first font that matched the font name while the 14 PDF Standard fonts had the highest search priority. Applications which use the standard fonts can change the search order so that the standard fonts become a higher search priority than system fonts.

SetFontSearchOrderEx

Syntax:

```
void pdfSetFontSearchOrderEx(
    const PPDF* IPDF, // Instance pointer
    TFontBaseType S1, // First search format
    TFontBaseType S2, // Second search format
    TFontBaseType S3, // Third search format
    TFontBaseType S4) // Fourth search format

typedef enum
{
    fbtTrueType, // TrueType, TrueType Collections
    fbtType1,    // Type1 font
    fbtOpenType, // OpenType font with Postscript outlines
    fbtStdFont,  // PDF Standard font
    fbtDisabled  // Can be used to disable a font format
}TFontBaseType;
```

The function changes the font search order in the same way as SetFontSearchOrder() but it does not use an array to set the search order to improve the compatibility to programming languages with limited array support.

SetFontSelMode

Syntax:

```
LBOOL pdfSetFontSelMode(
    const PPDF* IPDF,  // Instance pointer
    TFontSelMode Mode) // Font selection mode

typedef enum
{
    smFamilyName     = 0, // Default
    smPostScriptName = 1,
    smFullName       = 2
}TFontSelMode;
```

The function changes the font selection mode. Note that the font names that will be passed to SetFont() must correspond with the current font selection mode. If you don't know which font name you have in memory then use SetFontEx() to select fonts. The font selection mode can be changed at runtime whenever necessary.

SetFontWeight

Syntax:

```
LBOOL pdfSetFontWeight(
    const PPDF* IPDF, // Instance pointer
    SI32 Weight)      // Font weight
```

The font weight specifies the thickness or boldness of a font. Not all fonts are available in a bold style but such a style can be emulated. The property *FontWeight* specifies the maximum weight that can be emulated.

A bold or bolder font style will only be emulated if the distance between the requested and the available weight is larger or equal 200 and if the requested weight is smaller or equal *FontWeight*.

Font weights:

- `100 – 300  // Ultra-light, Light`
- `400 – 500  // Standard`
- `600 – 1000 // Bold till UltraBlack`

Default value: **1000**

To disable the emulation of very thick font styles set the property to a lower value. To fully disable bold style emulation set the value to 100.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetGStateFlags

Syntax:

```
void pdfSetGStateFlags(
    const PPDF* IPDF,    // Instance pointer
    TGStateFlags Flags,  // See description
    LBOOL Reset)         // See below

typedef UI32 TGStateFlags;
#define gfCompatible          0x00000000 // Default
#define gfRestorePageCoords   0x00000001 // see below
#define gfRealTopDownCoords   0x00000002 // see below
#define gfNativeBlackWhite    0x00000004 // Do not convert RGB b/w to gray
#define gfUseImageColorSpace  0x00000008 // Use the image color space
#define gfIgnoreICCProfiles   0x00000010 // see description below
#define gfAnsiStringIsUTF8    0x00000020 // Ansi strings are UTF-8 encoded
#define gfRealPassThrough     0x00000040 // Insert JPEG images unchanged
#define gfNoBitmapAlpha       0x00000080 // Ignore the alpha channel in bitmaps
#define gfNoImageDuplCheck    0x00000100 // Disables duplicate check for images
#define gfNoObjCompression    0x00000200 // Disables object compression
#define gfComplexText         0x00000400 // If set, text is processed with Uniscribe on Windows.
#define gfDisableJavascript   0x00000800 // Ignore OnFormat scripts of text fields
#define gfDisableBidiCtrls    0x00001000 // Disable bidi control characters
#define gfDoNotComprMetadata  0x00002000 // Disable compression for metadata streams.
#define gfUpdatePDFVTModDate  0x00004000 // If set, the key GTS_PDFVTModDate will be set to the
                                         // file's modification date.
#define gfSkaleAnnotIcons     0x00008000 // If set, icons of text and file attach annotation are
                                         // scaled with the page like every other annotation.
```

The function sets optional flags affecting the graphics state, coordinate handling, as well as color and image conversion rules. If the parameter *Reset* is true, the new flags replace current flags. If set to false, the flags are combined with the current flags.

Flag	Description
gfCompatible	The graphics state is fully compatible to earlier versions of DynaPDF.
gfRestorePageCoords	If set, the current base coordinate system like bottom or top down is saved and restored with the graphics state.
gfRealTopDownCoords	This flag is reserved for future extensions. It is not implemented yet.
gfNativeBlackWhite	If set, RGB black or white is not converted to DeviceGray. This flag affects text and vector graphics but no images.
gfUseImageColorSpace	If set, the active color space is ignored when inserting an image. The color space is taken from the image file instead. See also "Color spaces and Images".

`gfIgnoreICCProfiles`	If set, embedded ICC profiles in image files are ignored when inserting an image. The image is inserted in the base color space instead. This flag is not meaningful if the flag gfUseImageColorSpace is absent. See also "Color spaces and Images".
`gfAnsiStringIsUTF8`	If set, all Ansi functions (functions which end with an uppercase 'A' or functions which contain only single byte string parameters (data type char*)) interpret string parameters as UTF-8 encoded Unicode strings. If a function does not support Unicode then the string will be converted to the encoding that the function supports, typically to the code page 1252.

This flag does **NOT** affect single byte strings in wide string functions which contain also single byte string prameters (functions which end with an uppercase 'W'), e.g. DecryptPDFW() or EncryptPDFW().

The flag is also considered in functions which use structures to pass string parameters to DynaPDF. For example, the function InitColorManagement() uses the structure TPDFColorProfiles. This structure supports Ansi and UTF-16 Unicode strings. If the flag gfAnsiStringIsUTF8 is set, Ansi strings are interpreted as UTF-8 encoded Unicode strings.

Special attention must be taken into account when encrypting or decrypting PDF files since most encryption handlers do not support Unicode passwords. See CloseFileEx() for further information.

`gfRealPassThrough`	If set, JPEG images are inserted as is. JPEG images are normally rebuild, also in pass-through mode, to avoid issues with certain malformed JPEG images which cannot be displayed in Adobes Acrobat or Reader. If you know that your JPEG images work then set this flag to avoid unnecessary processing time.
`gfNoBitmapAlpha`	If set, the alpha channel in bitmaps files will be ignored. This is sometimes useful since many 32 bit bitmaps contain an invalid alpha channel that makes the image fully transparent.
`gfNoImageDuplCheck`	If set, no duplicate check for images will be performed. This can significantly improve processing speed especially for memory

	based images.
gfComplexText	This flag enables complex text layout. See section "Complex Text Layout" for further information.
gfNoObjCompression	If set, object compression will be disabled. Object compression is enabled by default if the PDF version is set to PDF 1.5 or higher.
gfDisableJavascript	If set, Javascript actions associated with the *OnFormat* event of text fields are not executed to format the field value.
gfDisableBidiCtrls	Meaningful only if gfComplexText is set too. If set, bidi control characters are ignored. This flag can be useful if the result of web browsers should be emulated since web browsers do not support bidi control characters. This flag is also used internally to create the appearance stream of form fields since form fields do not support bidi control characters.
gfDoNotComprMetadata	If set, metadata streams assiciated with fonts, templates, or images, for example, will not be compressed. Metadata streams of PDF/X files are always left uncompressed.
gfUpdatePDFVTModDate	If set, the key GTS_PDFVTModDate will be set to the file's modification date. The key exists in PDF/VT files only.
gfSkaleAnnotIcons	If set, icons of text and file attach annotation are scaled with the page like every other annotation. These annotations have normally a fixed size that is indendent of the zoom or scaling factor.

SetIconColor

Syntax:

```
LBOOL pdfSetIconColor(
   const PPDF* IPDF, // Instance pointer
   UI32 Color)       // RGB color value
```

The icon color is used for the closed state of a text annotation. The annotation appears then as an icon. The color value must be defined in RGB color space.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

```
Default value = 0x0099FFFF
```

SetImportFlags

Syntax:

```
LBOOL pdfSetImportFlags(
    const PPDF* IPDF,   // Instance pointer
    TImportFlags Flags) // see below

typedef UI32 TImportFlags;
#define ifImportAll         0x0FFFFFFE // Default
#define ifContentOnly       0x00000000 // No interactive objects
#define ifNoContent         0x00000001 // Interactive objects only
#define ifImportAsPage      0x80000000 // No conversion to templates
#define ifCatalogAction     0x00000002 // Open action, Catalog actions
#define ifPageActions       0x00000004 // Page actions
#define ifBookmarks         0x00000008 // Bookmarks
#define ifArticles          0x00000010 // Articles
#define ifPageLabels        0x00000020 // Page labels
#define ifThumbs            0x00000040 // Thumbnails
#define ifTranspGroups      0x00000080 // Transparency groups
#define ifSeparationInfo    0x00000100 // Separation info
#define ifBoxColorInfo      0x00000200 // Box color info
#define ifStructureTree     0x00000400 // Structure tree (can be large)
#define ifTransition        0x00000800 // Presentation settings
#define ifSearchIndex       0x00001000 // External search index
#define ifJavaScript        0x00002000 // Global JavaScripts
#define ifJSActions         0x00004000 // JavaScript actions
#define ifDocInfo           0x00008000 // Document info entries
#define ifEmbeddedFiles     0x00200000 // File attachments
#define ifFileCollections   0x00400000 // File collections (PDF 1.7)
#define ifAllAnnots         0x009F0000 // All annotations
   #define ifFreeText       0x00010000 // FreeText annotations
   #define ifTextAnnot      0x00020000 // Text annotations
   #define ifLink           0x00040000 // Link annotations
   #define ifStamp          0x00080000 // Stamp annotations
   #define if3DAnnot        0x00100000 // 3D annotations
   #define ifOtherAnnots    0x00800000 // Other annotations
#define ifFormFields        0x01000000 // Interactive form fields
#define ifPieceInfo         0x02000000 // Private data. See description.
/* ------------------- Special flags ------------------- */

#define ifPrepareForPDFA    0x10000000 // See description
#define ifEnumFonts         0x20000000 // Import fonts for EnumDocFonts()
#define ifAllPageObjects    0x40000000 // Import links with ImportPageEx()
```

The function sets optional flags to control the import of external PDF files. The flags are ignored by the function ImportPage(). ImportPageEx() ignores the flags too if the page is imported outside of an open page (see ImportPageEx() for further information). The flags are defined as a bit-mask; multiple flags can be combined by using a bitwise or operator.

A second set of flags can be set with SetImportFlags2().

This function cannot fail; the return value is always 1.

Flag	Description
ifImportAll	If set, all objects of an external PDF file are imported.
ifContentOnly	If set, interactive objects are excluded from import.

`ifNoContent`	If set, the visual contents of external pages will not be imported. This flag can be used to copy interactive objects from one file to another.
`ifImportAsPage`	If set, pages are not converted to templates. This flag is useful if an imported page must not be used multiple times. Conversion to templates is recommended if multiple imported pages should appear on the same page since overlapping pages can cause resource conflicts.
`ifCatalogAction`	If set, catalog actions will be imported if any. The documents catalog is the root object of a PDF file, which can trigger actions when the file will be opened, closed, or printed. See AddActionToObj() for further information.
`ifPageActions`	If set, page actions are imported if any. A page action is an action that was added to the page itself, not an object of a page.
`ifBookmarks`	If set, bookmarks are imported if any.
`ifArticles`	If set, Articles are imported if any.
`ifPageLabels`	If set, page labels are imported if any. If multiple files are imported, DynaPDF imports the page labels of the first file only.
`ifThumbs`	If set, thumb nail images are imported if any. Thumb nails can be dynamically created by Acrobat since Acrobat 5. The thumb nail images can be removed to reduce disk space.
`ifTranspGroups`	If set, transparency groups are imported if any. If transparency groups are removed the objects appear then opaque.
`ifSeparationInfo`	If set, the separation information dictionary will be imported if any.
`ifBoxColorInfo`	If set, the bounding boxes artbox, bleedbox and trimbox are shown in user defined color. Note that the cropbox represents the current page; the page is already cropped if a cropbox was defined.
`ifStructureTree`	If set, structure trees are imported if any. A structure tree contains information about the logical structure of a document. This information can be used by screen readers and other accessibility plug-ins to enable certain features. The information of a structure tree is not required to view or print a PDF document.
`ifTransition`	If set, transition settings are import. A transition dictionary contains

	information about how a presentation should be displayed. These settings can be removed if the document should not be used in presentation mode.
ifSearchIndex	If set, the file path to an external search index will be imported if any.
ifJavaScript	If set, global JavaScripts are imported if any. Note that JavaScript actions can use the functions defined in global JavaScripts. If these scripts are removed, JavaScript actions must normally also removed to avoid errors. However, it is often useful to remove the scripts for certain debug operations.
ifJSActions	If set, JavaScript actions are imported if any. This flag is useful if an imported form contains JavaScripts which produces errors or warnings. For instance, a document can define certain scripts which may require specific settings such as a document info string and so on. It is then often difficult to determine which object produces the error or warning because it can be a JavaScript action or a global JavaScript. In such cases, JavaScript actions can be excluded temporarily to check whether the error still occurs.
ifDocInfo	If set, document info entries are imported if any. Already existing entries are not overridden.
ifEmbeddedFiles	If set, embedded files are imported.
ifFileCollections	If set, the global definition of a collection is imported. The file attachments are still imported if the flag ifEmbeddedFiles is set. The PDF file is simply not longer marked as portable collection, also known as PDF Package. See also CreateCollection().
ifAllAnnots	If set, all types of annotations are imported.
ifFreeText	This flag is meaningful only if the flag ifAllAnnots is clear. If set, FreeText annotations are imported if any.
ifTextAnnot	This flag is meaningful only if the flag ifAllAnnots is clear. If set, text annotations are imported if any.
ifLink	This flag is meaningful only if the flag ifAllAnnots is clear. If set, link annotations are imported if any. This annotation type is used for weblinks, page links, and file links.

`ifStamp`	This flag is meaningful only if the flag ifAllAnnots is clear. If set, stamp annotations are imported if any.
`if3DAnnot`	This flag is meaningful only if the flag ifAllAnnots is clear. If set, 3D annotations are imported if any.
`ifOtherAnnots`	This flag is meaningful only if the flag ifAllAnnots is clear. If set, other annotation types as the ones listed above are imported if any.
`ifFormFields`	If set, interactive form fields are imported if any (also called AcroForm). Form fields are also annotations, but this type is handled separately.
`ifPrepareForPDFA`	If set, LZW compressed streams are recompressed with Flate, the Interpolate key of images is deleted if set, embedded files are not imported.
`ifPieceInfo`	The PieceInfo dictionary contains arbitrary application defined data. The data in this dictionary is meaningful only for the application that created the data.
`ifEnumFonts`	If set, only font objects are imported to enable high performance enumeration of the fonts used in a PDF file. The PDF file must be deleted with FreePDF() after the fonts were enumerated. See EnumDocFonts() for further information.
`ifAllPageObjects`	If set, all types of link annotations are imported when importing a page within an open page with ImportPageEx(). The entire document should be imported in this case.

SetImportFlags2

Syntax:

```
LBOOL pdfSetImportFlags2(
  const PPDF* IPDF,     // Instance pointer
  TImportFlags2 Flags) // See below
typedef UI32 TImportFlags2;
#define if2MergeLayers     0x00000001 // See description
#define if2Normalize       0x00000002 // Replace LZW compression with Flate, apply limit
                                      // checks, repair errors if possible
#define if2UseProxy        0x00000004 // Load streams on demand
#define if2NoMetadata      0x00000008 // Ignore metadata streams attached on fonts, pages,
                                      // images, and so on.
#define if2DuplicateCheck  0x00000010 // Perform a duplicate check during PDF import. See
                                      // description below.
#define if2NoResNameCheck  0x00000020 // Useful in viewer applications
#define if2CopyEncryptDict 0x00000040 // If set, the encryption settings of an encrypted
                                      // PDF file are copied to the new PDF file. The flag
                                      // does nothing if the file is not encrypted.
#define if2SkipDPartData   0x00000080 // If set, DPart dictionaries are not imported.
```

The function sets optional flags to control the import of external PDF files. The flags are defined as a bit-mask; multiple flags can be combined by using a bitwise or operator.

This function cannot fail; the return value is always 1.

Flag	Description
if2MergeLayers	If set, layers of different PDF files with the same name are merged. That means the layer names in the same hierarchy will be unique. A merged layer controls the visibility of layers from different imported PDF files.
	If the flag is absent (default) it is possible that the same layer name occurs multiple times in the same hierarchy.
if2Normalize	This flag must be set if the file should be normalized with CheckConformance(). The flag enables additional limit checks, LZW compression will be replaced with Flate, and further repairs are applied which are not enabled by default.
If2UseProxy	If set, stream objects are not loaded into memory to reduce the memory usage. This makes it possible to import PDF files of almost arbitrary size with minimal memory usage. Nothing special must be considered if this flag is set. It is also allowed to call CloseImportFile() after the wished pages were imported. The file is automatically opened again when streams must be loaded.
	PDF files which were loaded in this way must not be deleted before the new PDF file in memory was closed.
if2NoMetadata	If set, metadata streams which are attached to fonts, images, pages, templates, and so on will not be imported.

`if2DuplicateCheck`	If set, the import algorithm performs a duplicate check on recourse objects like images, templates, fonts, and so on, as well as on many other objects. This flag can be useful if a PDF file must be split into separate files and if these files must be merged again into one PDF file. Such an action would normally lead to many double objects in the resulting PDF file.
	The duplicate check is computation intensive and must be applied for all PDF files which will be imported.
`if2NoResNameCheck`	Import resources as is. This flag can significantly imporove the loading time of pages with a huge resource tree. This flag should only be set in viewer applications to improve the loading time of pages.
`if2CopyEncryptDict`	If set, the encryption settings of an enrypted PDF file are copied to the document in memory when OpenImportFile() or OpenImportBuffer() is called. This makes it possible to edit and save a PDF file without loosing the encryption settings. The PDF file must be closed with CloseFile() or CloseAndSignFile(), for example since CloseFileEx() would override the encryption settings.
`if2SkipDPartData`	If set, DPart dictionaries are not imported. *DPart* stands for Document Parts, a feature of PDF/VT.

SetItalicAngle

Syntax:

```
LBOOL pdfSetItalicAngle(
   const PPDF* IPDF, // Instance pointer
   double Angle)     // Angle alpha in degrees
```

The property *ItalicAngle* is used to emulate an italic font if the font is not available in an italic style. To avoid emulation set the angle to zero.

The italic angle must not be smaller than -30 degrees and not larger than 30 degrees.

```
Default value = 14.0
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetJPEGQuality

Syntax:

```
LBOOL pdfSetJPEGQuality(
   const PPDF* IPDF, // Instance pointer
   UI32 Value)       // Quality (0..100 or 0..-1000)
```

The function sets the quality for JPEG and JPEG 2000 compressed images. Lower values cause higher compression rates, however, worse image quality.

The JPEG encoder supports a range from 0 through 100. The value can be set as positve or negative number. The JPEG 2000 encoder supports a positive range from 0 through 1000, and from 0 through -1000. Values outside the allowed range are adjusted to next nearest valid value. The larger range is supported to achieve better backwards compatibility to older DynaPDF versions.

A negative value deactivates pass-through mode of the JPEG and JPEG 2000 decoders. That means, an image will be re-compressed, also if not necessary.

The JPEG 2000 encoder divides a value greater 100 by 10 and uses finally the absolute value.

The values 0, 100, or -1000 activate loss-less encoding. Loss-less encoding is supported by JPEG 2000 only.

```
Default value = 70
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetLanguage

Syntax:

```
LBOOL pdfSetLanguage(
   const PPDF* IPDF,   // Instance pointer
   const char* ISOTag) // Language identifier see below
```

The property specifies the language of the document. The document's language should always be set when creating Tagged PDF files or Interactive Forms. The spell check feature in Adobe's Acrobat depends on the properly defined document language.

Language Identifiers

The value of the language property is a string that specifies the language with a language identifier having the syntax, defined in *Internet RFC 3066, Tags for the Identification of Languages.*

This syntax, which is summarized below, is also used to identify languages in XML, according to the World Wide Web Consortium document *Extensible Markup Language* (XML) 1.0.

Language identifiers can be based on codes defined by the International Organization for Standardization in ISO 639 and ISO 3166 or registered with the Internet Assigned Numbers

Authoring (IANA, whose Web site is located at http://www.iana.org), or they can include codes created for private use. A language identifier consists of a primary code optionally followed by one or more sub codes (each preceded by a hyphen). The primary code can be any of the following:

- A 2-character ISO 639 language code - for example, *en* for English or *es* for Spanish.
- The letter i, designating an IANA-registered identifier
- The letter x for private use

The first sub code can be a 2-character ISO 3166 country code, as in *en-US*, or a 3- to 8-character sub code registered with IANA, as in *en-cockney* or *i-cherokee* (except in private identifiers, for which sub codes are not registered). Sub codes beyond the first can be any that have been registered at IANA.

Language identifiers are treated case-sensitive.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetLeading

Syntax:

```
LBOOL pdfSetLeading(
   const PPDF* IPDF,  // Instance pointer
   double Value)      // New leading or 0
```

The function sets the leading (also called line height or line spacing) that is used by the functions AddContinueText() and WriteFText() to calculate the distance between two text lines.

The default leading in PDF is the font size. However, this value is often too small and can cause overlapping text lines. Most modern TrueType and OpenType fonts contain metrics of the typographic leading or line height that was intended for the particular font. This value considers the particular properties of the font and is the recommended way to calculate the line height.

The typographic line height or leading is returned by the functions GetTypoLeading() and GetFontMetrics().

A value of 0 determines that the font size should be used as leading, this is the default behaviour.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetLicenseKey

Syntax:

```
LBOOL pdfSetLicenseKey(
    const PPDF* IPDF,   // Instance pointer
    const char* Value)  // License key as string
```

The function sets the license key and deactivates the demo mode if the key was valid. If the value represents a correct license key the demo string does not longer appear on PDF pages. Once a correct key was set, it will be active until the library will be unloaded or the current PDF instance is deleted.

Note that the key must be set for each PDF instance or process of a multi-threading application.

To determine whether a specific function is available in DynaPDF Starter or Lite, pass the string "Starter" or "Lite" to the function; all functions which are disabled in these versions produce then an error. It is a good idea to set an error callback function so that you can see all errors…

Notice: The license key must **not** be stored in the registry or other files in an unencrypted form.

Return values:

- The function returns **true** whether or not the key was valid!
- If the key is valid but expired, then the return value is **false**.

The recommended way to apply the license key is as follows:

```
// Check first whether the right version of the dynapdf.dll was loaded and
// then set the license key.
char* ver = pdfGetDynaPDFVersionInt();
if (ver[0] <= 40450131 || pdfSetLicenseKey(pdf, "...") == false)
{
    throw("Wrong dynapdf.dll version loaded!");
}
```

SetLineAnnotParms

Syntax:

```
LBOOL pdfSetLineAnnotParms(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle,      // Handle of a line annotation
    SI32 FontHandle,  // Font handle or -1 for Helvetica
    double FontSize,  // Font size of the caption or zero
    struct TLineAnnotParms* Parms) // Can be NULL to delete all measure
                                   // line specific values

typedef enum
{
    cpInline, // The caption is centered inside the line
    cpTop     // The caption is drawn on top of the line
}TLineCaptionPos;
```

```
struct TLineAnnotParms
{
    UI32  StructSize;          // Must be set to sizeof(TLineAnnotParms)
    LBOOL Caption;             // See description
    float CaptionOffsetX;      // Horizontal offset of the caption
    float CaptionOffsetY;      // Vertical offset of the caption
    TLineCaptionPos CaptionPos; // See enum TLineCaptionPos
    float LeaderLineLen;       // Length of the leader lines
    float LeaderLineExtend;    // Leader line extend (must be positive or 0)
    float LeaderLineOffset;    // Amount of space between the endpoints of the
                               // annotation and the leader lines (must be a
                               // positive value or zero)
};
```

The function sets or changes the properties of a line annotation relating to measure lines. The parameter *Parms* can be set to NULL to delete all measure line specific values. The parameters *FontHandle* and *FontSize* will be ignored in this case.

The member *Caption* specifies whether the parameter *Content* of the function LineAnnot() should be used as caption of the measure line. Although a measure line can display the string in a PopUp annotation like ordinary line annotations, this is not recommended and not fully supported in Adobe's Acrobat or Reader.

If *Caption* is true (recommended), the caption is drawn horizontally centered either on top or inside the measure line. The text position can be changed from its normal position with the members *CaptionOffsetX* and *CaptionOffsetY*.

The parameter *FontHandle* can be used to specify an arbitrary font that should be used to draw the caption. Although it is possible to use any font or font size greater 0 with DynaPDF, a PDF viewer will change the font and font size to it's default values when the annotation will be edited since line annotations support no property to specify the font or font size.

The default font to draw the caption is Helvetica and the default font size is 9 units. In order to use the default font set the parameter *FontHandle* to -1. The default font size is used if the parameter *FontSize* is set to to zero.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetLineCapStyle

Syntax:

```
LBOOL pdfSetLineCapStyle(
    const PPDF* IPDF,     // Instance pointer
    TLineCapStyle Style) // see below

typedef enum
{
    csButtCap   = 0,
    csRoundCap  = 1,
    csSquareCap = 2
}TLineCapStyle;
```

The function sets the line cap style that specifies how the endpoint of a line will be drawn. The property must be applied inside an open page, template, or pattern.

Style	Appearance	Description
csButtCap		The stroke is squared off at the endpoint of the path. There is no projection beyond the end of the path.
csRoundCap		A semicircular arc with a diameter equal to the line width is drawn around the endpoint and filled in.
csSquareCap		The stroke continues beyond the endpoint of the path for a distance equal to half the line width and is then squared off.

```
Default value = csButtCap
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetLineDashPattern (obsolete)

Syntax:

```
LBOOL pdfSetLineDashPattern(
    const PPDF* IPDF, // Instance pointer
    const char* Dash, // Dash array defined as string
    SI32 Phase)       // The pattern's phase
```

This function is obsolete, please use SetLineDashPattern2() instead.

Return values:
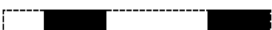
If the function succeeds the return value is 1. If the function fails the return value is 0.

SetLineDashPattern2

Syntax:

```
LBOOL pdfSetLineDashPattern2(
    const PPDF* IPDF,   // Instance pointer
    const float* Dash,  // Dash array defined
    UI32 NumValues,     // Number of values in the dash array
    float Phase)        // The pattern's phase
```

The line dash pattern controls the pattern of dashes and gaps used to stroke paths. It is specified by a dash array and a dash phase. The dash array's elements are numbers that specify the lengths of alternating dashes and gaps; the dash phase specifies the distance into the dash pattern at which to start the dash. The elements of both the dash array and the dash phase are expressed in user space units. Before beginning to stroke a path, the dash array is cycled through, adding up the lengths of dashes and gaps. When the accumulated length equals the value specified by the dash phase, stroking of the path begins, using the dash array cyclically from that point onward. The table shows examples of line dash patterns. As can be seen from the table, an empty dash array and zero phase can be used to restore the dash pattern to a solid line (you can also use ResetLineDashPattern()).

Dash & Phase	Appearance	Description
(NULL, 0)		No dash; solid, unbroken lines
("3", 0)		3 units on, 3 units off…
("2", 1)		1 on, 2 off, 2 on, 2 off, …
("2 1", 0)		2 on, 1 off, 2 on, 1 off, …
("3 5", 6)		2 off, 3 on, 5 off, 3 on, 5 off, ...
("2 3", 11)		1 on, 3 off, 2 on, 3 off, 2 on, ...

Dashed lines wrap around curves and corners just as solid stroked lines do. The ends of each dash are treated with the current line cap style, and corners within dashes are treated with the current line join style. A stroking operation takes no measures to coordinate the dash pattern with features of the path; it simply dispenses dashes and gaps along the path in the pattern defined by the dash array. When a path consisting of several sub paths is stroked, each sub path is treated independently - that is, the dash pattern is restarted and the dash phase is reapplied to it at the beginning of each sub path.

How to create a dotted line?

As described above, a line dash pattern is drawn by applying the current line cap style and join style. A round cap is painted as semicircular arc with a diameter equal to the line width. If we set the length of the first on-state to 0 then we get a circle, or dot. The second number of the dash array specifies the length of the off-state measured from the centers of our circles.

It is quite easy to draw the line now:

```
...
float dash[2] = {0.0f, 60.0f};
pdfSetLineWidth(pdf, 30);
pdfSetLineCapStyle(pdf, csRoundCap);
pdfSetLineDashPattern2(pdf, dash, 0.0f);
pdfMoveTo(pdf, 50, 50);
pdfLineTo(pdf, 550, 50);
pdfStrokePath(pdf);
...
```

Output:
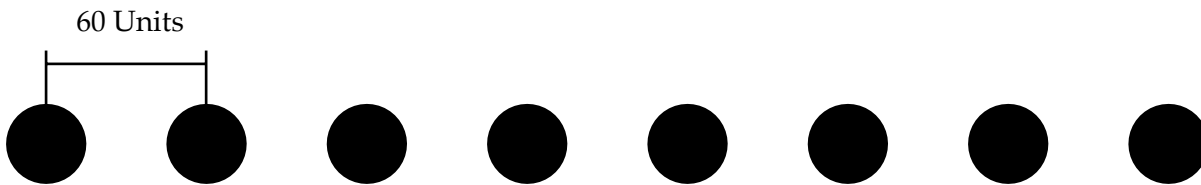
60 Units

● ● ● ● ● ● ● ●

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetLineDashPatternEx (obsolete)

Syntax:

```
LBOOL pdfSetLineDashPatternEx(
    const PPDF* IPDF,    // Instance pointer
    const double* Dash,  // Array of doubles representing the pattern
    UI32 NumValues,      // Array length
    SI32 Phase)          // Dash phase
```

This function is obsolete, please use SetLineDashPattern2() instead.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetLineJoinStyle

Syntax:

```
LBOOL pdfSetLineJoinStyle(
    const PPDF* IPDF,    // Instance pointer
    TLineJoinStyle Style) // see below

typedef enum
{
  jsMiterJoin = 0,
  jsRoundJoin = 1,
  jsBevelJoin = 2
}TLineJoinStyle;
```

The function sets the line join style which specifies how two line segments are connected. The property must be applied inside an open page, template or pattern.

Style	Appearance	Description
jsMiterJoin		The outer edges of the strokes for the two segments are extended until they meet at an angle, as in a picture frame. If the segments meet at too sharp an angle (as defined by the miter limit parameter — see SetMiterLimit(), a bevel join is used instead.
jsRoundJoin		An arc of a circle with a diameter equal to the line width is drawn around the point where the two segments meet, connecting the outer edges of the strokes for the two segments. This pie slice-shaped figure is filled in, producing a rounded corner.
jsBevelJoin		The two segments are finished with butt caps (see SetLineCapStyle()) and the resulting notch beyond the ends of the segments is filled with a triangle.

Default value = jsMiterJoin

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetLineWidth

Syntax:

```
LBOOL pdfSetLineWidth(
    const PPDF* IPDF, // Instance pointer
    double Value)     // Line width
```

The function sets the line width used to stroke paths. The line width is also used to specify the thickness of the border of annotations and interactive form fields. In the latter case, the line width should be a multiple of one (or zero) and not exceed 3 units. Larger values are still correctly rendered by Adobe's Acrobat but the appearance can be changed when a Reset Form Action is executed or when other changes are made to the document.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetLinkHighlightMode

Syntax:

```
LBOOL pdfSetLinkHighlightMode(
    const PPDF* IPDF,    // Instance pointer
    THighlightMode Mode) // see below
typedef enum
{
    hmNone,    // Default
    hmInvert,  // Invert the contents of the annotation's bounding box
    hmOutline, // Invert the annotations border
    hmPush,    // Simulate a push button effect
    hmPushUpd  // Update the appearance stream on changes
}THighlightMode;
```

The function sets the highlight mode that is used by link annotations. The highlight mode applies a visual effect when moving the mouse over a link annotation.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetListFont

Syntax:

```
LBOOL pdfSetListFont(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle)      // Font handle
```

The function marks an arbitrary font as list font. The font is used in WriteFText() when a list operator was found. The symbol to be used as list symbol can be set with the list operator. See WriteFText() for further information. The parameter *Handle* must be a valid font handle that was returned by SetFont(), SetFontEx(), or LoadFont().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetMatrix

Syntax:

```
LBOOL pdfSetMatrix(
    const PPDF* IPDF,    // Instance pointer
    struct TCTM* Matrix) // Transformation matrix

struct TCTM
{
    double a;
    double b;
    double c;
    double d;
    double x;
    double y;
};
```

The function left multiplies the current transformation matrix with the new one. The transformation matrix is part of the graphics state. When changing the coordinate system it is usually best to save the graphics state beforehand with SaveGraphicState(). This makes it possible to restore the coordinate system with RestoreGraphicState().

Notice:

The native coordinate system in PDF is bottom up. Changing the base coordinate system with SetPageCoords() to top down results in a mirrored coordinate system that is applied with the current transformation matrix. DynaPDF makes sure that text functions and so on produce correct results in the mirrored coordinate system.

However, when multiplying a new matrix with the already changed transformation matrix DynaPDF must set the base coordinate system back to bottom up after multiplying the matrices. This is required because multiplying the transformation matrix again with another one would otherwise produce incorrect results.

Matrix multiplications and every other transformation of the coordinate system should always be applied in bottom up coordinates. It is the only way to avoid side effects when changing the coordinate system arbitrary often.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetMaxErrLogMsgCount

Syntax:

```
void pdfSetMaxErrLogMsgCount(
   const PPDF* IPDF, // Instance pointer
   UI32 Value)       // New limit
```

The function can be used the change the maximum number of error messages which can be stored in the internal error log. It is normally best to restrict the number of error messages especially when importing PDF files because damaged PDF files can produce hundreds or thousands of warnings.

See also GetErrLogMessage().

The default value is 100.

SetMaxFieldLen

Syntax:

```
SI32 pdfSetMaxFieldLen(
   const PPDF* IPDF, // Instance pointer
   UI32 TxtField,    // Text field handle
   SI32 MaxLen)      // Maximum string length or zero if not restricted
```

The function changes the maximum count of characters which can be entered into a text field. A value of zero determines that the string length should not be restricted.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetMetaConvFlags

Syntax:

```
SI32 pdfSetMetaConvFlags(
    const PPDF* IPDF, // Instance pointer
    TMetaFlags Flags) // see below

typedef UI32 TMetaFlags;
#define mfDefault          0x00000000 // No flags
#define mfDebug            0x00000001 // Insert debug comments
#define mfShowBounds       0x00000002 // Show the bounding boxes of text
#define mfNoTextScaling    0x00000004 // Do not scale text records
#define mfClipView         0x00000008 // Clip the output rectangle
#define mfUseRclBounds     0x00000010 // Use the raw bounding box rclBounds
#define mfNoClippingRgn    0x00000040 // Ignore clipping regions
#define mfNoFontEmbedding  0x00000080 // Don't embed fonts used by the EMF
#define mfNoImages         0x00000100 // Ignore image records
#define mfNoStdPatterns    0x00000200 // Ignore standard hatch patterns
#define mfNoBmpPatterns    0x00000400 // Ignore bitmap patterns
#define mfNoText           0x00000800 // Ignore text records
#define mfUseUnicode       0x00001000 // Use always Unicode to print text
#define mfUseTextScaling   0x00004000 // Scale text (see description)
#define mfNoUnicode        0x00008000 // Avoid the usage of Unicode fonts
#define mfFullScale        0x00010000 // Scale coordinates to Windows size
#define mfUseRclFrame      0x00020000 // See description
#define mfDefBkModeTransp  0x00040000 // Initial backg. mode is transparent
#define mfApplyBidiAlgo    0x00080000 // Apply the bidirectional algorithm
#define mfGDIFontSelection 0x00100000 // Use the GDI to select fonts
#define mfRclFrameEx       0x00200000 // See description
#define mfSrcCopy_Only     0x00800000 // See description
#define mfClipRclBounds    0x01000000 // See description
#define mfDisableRasterEMF 0x02000000 // Disable the EMF rasterizer
#define mfNoBBoxCheck      0x04000000 // Disable the bbox check
#define mfDontSkipROP_DPa  0x10000000 // If set, the ROP code 0x00A000C9 (DPa) is considered in
                                      // BitBlt, StretchBlt, TranparentBlt, and AlphaBlend records.
                                      // This kind of ROP code combines a pattern brush with the
                                      // backdrop. If the flag mfDisableRasterEMF is absent, the
                                      // usage of such a ROP code causes that the file is rendered
                                      // to an image.
#define mfCurrComprFilter  0x20000000 // If set, the currently active compression filter is used to
                                      // compress the image (see SetCompressionFilter()).
                                      // This flag is used only if the file is rastered to an image.

// Obsolete flags -> These flags are ignored, do not longer use them!
#define mfUseSpacingArray   0x0020 // Enabled by default
#define mfIntersectClipRect 0x2000 // Enabled by default
```

The function sets specific flags to control the conversion of metafiles. The flags are described in detail on the next page. This function cannot fail the return value is always 1.

Flag	Description
mfApplyBidiAlgo	If set, the bidirectional algorithm is applied on Unicode strings. This flag must be set to process Hebrew text correctly.
mfClipRclBounds	If set, the EMF file is drawn into a clipping rectangle in the size of the rclBounds rectangle. This flag can be useful if the EMF file contains objects outside the rclBounds rectangle.
mfClipView	If set, the metafile is drawn into a clipping rectangle in the size of the

	metafile or currently defined view. This flag should always be set if a user defined view is used. See InsertMetafileExt() for further information.
`mfCurrComprFilter`	If set, the currently active compression filter is used to compress the image (see SetCompressionFilter()). This flag is used only if the file is rastered to an image.
`mfDebug`	If set, the EMF record names are printed to the content stream which produces a specific output. Open the PDF file in a good text editor such as Textpad to view the output. The PDF file must not be compressed if this flag is used, otherwise you can't see the debug strings. Compression can be disabled with the function SetCompressionLevel().
`mfDefault`	This is the default behaviour. No specific parameters are used for metafile conversion.
`mfDefBkModeTransp`	If set, the initial background mode is set to transparent. SetBkMode records still override this state. The default background mode is opaque in the GDI. This state causes that a rectangle is printed in background of any text string, also if the rectangle is not required. Especially if text strings are printed as single characters the opaque background can drastically increase the resulting file size due to the many rectangles. This flag should always be set if the EMF or WMF files do not initialize the background mode to the required value.
`mfDisableRasterEMF`	If set, EMF files are no longer rastered to an image if the file uses ROP codes which combine background and foregound colors. Many EMF files use such ROP codes but it is often not required to raster the EMF file because many ROP codes which combine background and foreground colors have no effect if no other object was drawn in background.
`mfDontSkipROP_DPa`	If set, the ROP code 0x00A000C9 (DPa) is considered in BitBlt, StretchBlt, TranparentBlt, and AlphaBlend records. This kind of ROP code combines a pattern brush with the backdrop. If the flag mfDisableRasterEMF is absent, the usage of such a ROP code causes that the file is rendered to an image.
`mfFullScale`	If set, all coordinates are scaled to the output window size. Set this flag if the EMF file uses 32 bit coordinates, e.g. large CAD drawings. Full scaling avoids floating point overflows in PDF viewer applications because all coordinate transformations are already applied. The resulting file size is also smaller due to the smaller coordinate values which must be stored in the PDF file.

`mfGDIFontSelection`	If set, DynaPDF uses a GDI device context to select fonts. This flag can be set to make sure that DynaPDF selects exactly the fonts which the GDI uses to render to EMF file.
`mfNoBBoxCheck`	If set, the rclBounds and rclFrame rectangles are used as is to calculate the picture size.
	DynaPDF tries to identify invalid bounding boxes by default so that EMF files with invalid bounding boxes can be converted. The EMF converter uses the rclBounds rectangle to calculate the picture size if the resolution of the EMF file seems to be larger as 1800 DPI which is quite unusual.
	This is mostly an indication that the rclFrame rectangle was incorrectly calculated since EMF files are typcially created in a much lower resolution.
	If you need to process such high resolution EMF files then disable the bounding box check.
`mfNoBmpPatterns`	If set, bitmap patterns are ignored.
`mfNoClippingRgn`	If set, clipping regions will be ignored.
`mfNoFontEmbedding`	If set, fonts used by an EMF file are not embedded.
`mfNoImages`	If set, image records are ignored.
`mfNoStdPatterns`	If set, standard hatch patterns are not applied.
`mfNoText`	If set, text records are ignored.
`mfNoTextScaling`	If set, text strings are not scaled and no kerning space will be applied. EMF files contain sometimes an invalid spacing array especially when the original string was substituted from an Arabic code page. In such cases, characters do overlap. To avoid this effect disable the usage of the intercharacter spacing array; all strings are then printed without scaling or kerning space.
`mfNoUnicode`	If set, all strings are converted to the code page 1252 and no Unicode font will be embedded in the PDF file during EMF conversion. This flag should be set if PDF-1.2 compatibility is recommended because PDF-1.2 does not support Unicode fonts. However, note that you get invalid results if the EMF file contains characters outside the code page 1252.
`mfRclFrameEx`	If set, and if the rclBounds rectangle is larger than rclFrame, InsertMetafile() extends the output rectangle according to rclBounds and uses the resulting bounding box to calculate the image size (rclBounds represents the unscaled image size).
	This is probably the correct way to calculate the image size. However,

	to preserve backward compatibility the default calculation cannot be changed.
mfShowBounds	If set, the bounding boxes of text strings are shown by inserting a stroked rectangle. Use this flag if text strings appear misplaced.
mfSrcCopy_Only	If set, images which use a ROP code other than SRCCOPY are ignored. This can be useful when processing Excel 2007 spool files.
mfUseRclBounds	If set, the bounding box of the EMF picture is used to calculate the position and picture size. This can be useful if the rclFrame rectangle contains invalid values which avoid proper positioning and scaling of the graphic.
mfUseRclFrame	If set, the rectangle rclFrame of the EMF header is used to calculate the picture size. This flag is primarily used to convert EMF files which were originally created from non-portable WMF files. Set this flag if the EMF picture appears wrongly scaled.
mfUseTextScaling	If set, strings are scaled instead of applying kerning space to get the correct string width. Text scaling produces often better results due to limited precision of the integer values of the intercharacter spacing array. However, text scaling cannot always be used because characters can be placed individually on the x-axis by applying kerning space. In the latter case, strings can overlap or single characters appear on a wrong x-coordinate; do not set this flag in this case.
mfUseUnicode	If set, the character set within CreateFont() records will be ignored and all strings are printed in Unicode mode (EMF files contain Unicode strings only). This flag can be used to avoid the conversion of strings to the Ansi character set if Ansi_CHARSET was used in the CreateFont() record. The character set is often wrongly defined in EMF files so that characters outside of the Ansi_CHARSET are replaced by question marks due to the default conversion to Ansi if the character set Ansi_CHARSET is used.

SetMetadata

Syntax:

```
LBOOL pdfSetMetadata(
    const PPDF* IPDF,      // Instance pointer
    TMetadataObj ObjType,  // See below
    SI32 Handle,           // Object handle or -1 for the catalog object
    const void* Buffer,    // Pointer of the stream buffer
    UI32 BufSize)          // Buffer size in bytes

typedef enum
{
    mdoCatalog, // The global XMP stream of the document (no handle needed)
    mdoFont,    // Not yet supported
    mdoImage,   // Parameter Handle must be an image handle
    mdoPage,    // Parameter Handle must be a page number
    mdoTemplate // Parameter Handle must be a template handle
}TMetadataObj;
```

The function sets or replaces the XMP metadata stream of a specific object. The function deletes the XMP stream if no buffer will be provided.

If the global XMP stream of the Catalog object should be replaced then proceed as follows:

- Set the wished output PDF version with SetPDFVersion().
- Get a preview of the XMP stream with GetMetadata().
- Modify the returned stream as needed and save it with SetMetadata(), finished!

The above steps make sure that the XMP metadata and document info contain the same values. This is especially important for PDF standards like PDF/A or PDF/X. DynaPDF makes sure that the creation and modification date will not be changed when closing the file.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetMinLineWidth2 (Rendering Engine)

Syntax:

```
void PDF_CALL rasSetMinLineWidth2(
    IRAS* RasPtr, // Pointer of the rasterizer
    float Value)  // Minimum line width (must be positive)
```

The function sets the minimum line width in which thin lines are rendered. The value should be in the range 0.1 through 1.0.

Default value: 0.5
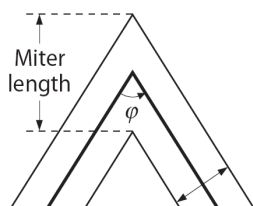
SetMiterLimit

Syntax:

```
SI32 pdfSetMiterLimit(
   const PPDF* IPDF, // Instance pointer
   double Value)     // New miter limit
```

When two line segments meet at a sharp angle and mitered joins have been specified as the line join style (see SetLineJoinStyle()), it is possible for the miter to extend far beyond the thickness of the line stroking the path. The miter limit imposes a maximum on the ratio of the miter length to the line width (see Figure below). When the limit is exceeded, the join is converted from a miter to a bevel.

The ratio of miter length to line width is directly related to the angle φ between the segments in user space by the formula:

$$\frac{\textbf{miterLength}}{\textbf{lineWidth}} = \frac{1}{\textbf{sin(φ/2)}}$$

For example, a miter limit of 1.414 convert miters to bevels for φ less than 90 degrees, a limit of 2.0 converts them for φ less than 60 degrees, and a limit of 10.0 converts them for φ less than approximately 11.5 degrees.

```
Default value = 10.0
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetNeedAppearance

Syntax:

```
LBOOL pdfSetNeedAppearance(
   const PPDF* IPDF, // Instance pointer
   LBOOL Value)      // Value -> true or false
```

The global NeedAppearance flag of an Interactive Form specifies whether a PDF viewer should rebuild the field appearances when opening the file or whether the existing definitions should be used. DynaPDF creates always appearance streams for all field types. However, in certain cases it can be useful to let the viewer render fields with their own algorithms because the exact way how Adobe's Acrobat builds the field appearances is not documented.

For example, when editing the contents of a text field in Adobe's Acrobat the viewer rebuilds first the field appearance before placing the editing cursor into the field. The new appearance created from Adobe's Acrobat can be slightly different in comparison to the one that was created by DynaPDF. The visible contents, especially of text fields, is sometimes not absolutely stable.

If the NeedAppearance flag is set, the viewer uses already its own algorithms to build the field appearances when opening the file. This avoids visible changes when editing a field. However, the NeedAppearance flag must not be set to true if a form contains page templates.

Return values:

If the function succeeds the return value is 1. If the function failse the return value is 0.

SetNumberFormat

Syntax:

```
SI32 pdfSetNumberFormat(
    const PPDF* IPDF,          // Instance pointer
    UI32 TxtField,             // Text field handle
    TDecSeparator Sep,         // Decimal separator
    UI32 DecPlaces,            // Number of decimal places
    TNegativeStyle NegStyle,   // Negative number format
    const char* CurrStr,       // Currency string or NULL
    LBOOL Prepend)             // Position of Currency string

// Thousand separator, decimal separator
typedef enum
{
    dsCommaDot,
    dsNoneDot,
    dsDotComma,
    dsNoneComma,
    dsQuoteDot
}TDecSeparator;

typedef enum
{
    nsMinusBlack,
    nsRed,
    nsParensBlack,
    nsParensRed
}TNegativeStyle;
```

The function restricts the allowed input characters of a text field to numbers and formats the resulting string as specified. A number format is applied via two separate JavaScript actions in PDF which are automatically created and added to the text field.

The same formats can also be applied manually by creating two JavaScript actions: one for the *OnKeyStroke* event, and one for the *OnFormat* event of the text field. See AddActionToObj() for a description of the events.

DynaPDF uses the JavaScript functions *AFNumber_Keystroke() / AFNumber_Format()* to apply a number format. The functions are described in the JavaScript scripting reference that is available at https://www.adobe.com.

Currency String

The optional currency string is interpreted in the Windows code page 1252 but it is possible to add arbitrary Unicode code points to the string in form of escape codes. Escape codes have the syntax backslash + "u" + four hexadecimal digits:

Example:

```
"\u20ac" = €
"€"      = € // Ok, this symbol is available in the code page
"\u0029" = $
```

Note that the backslash is an escape character in C/C++. The backslash itself must be preceded with another backslash if the string is defined as string literal. Note also that the lower case 'u' is part of the syntax and must not be omitted.

If the currency smybol should be separated with a space then add a space character before or after the symbol depending on *Prepend*.

The maximum length of the raw currency string including escape characters is 16 bytes. Longer strings are truncated.

Field value

The value of a text field with a number format must be a pure number with no formatting characters, e.g. "3445" or "2346.78". The decimal separator (if any) must be a period (.), independent of the current locale. Note that the Javascript code will fail if the field value contains non-numerical characters.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetOCGContUsage

Syntax:

```
LBOOL pdfSetOCGContUsage(
    const PPDF* IPDF,                   // Instance pointer
    UI32 Handle,                        // OCG handle
    struct TPDFOCGContUsage* Value) // See below

struct TPDFOCGContUsage
{
    UI32            StructSize;     // Must be set to sizeof(TPDFOCGContUsage)
    UI32            ExportState;    // 0 = Off, 1 = On, PDF_MAX_INT = not set
    const char*     InfoCreatorA;   // CreatorInfo -> The application that created the group
    const UI16*     InfoCreatorW;   // CreatorInfo -> The application that created the group
    const char*     InfoSubtype;    // CreatorInfo -> A name defining the type of content, e.g.
                                    // Artwork, Technical etc.
    const char*     LanguageA;      // A language code as described at SetLanguage()
    UI16*           LanguageW;      // A language code as described at SetLanguage()
    UI32            LangPreferred;  // 0 = Off, 1 = On, PDF_MAX_INT = not set. The preffered state if
                                    // there is a partial but no exact match of the language id.
    TOCPageElement  PageElement;    // If the group contains a pagination artefact
    UI32            PrintState;     // 0 = Off, 1 = On, PDF_MAX_INT = not set
    const char*     PrintSubtype;   // The type of content that is controlled by the OCG, e.g.
                                    // Trapping, PrintersMarks or Watermark.
    UI32            UserNamesCount; // The user names can be accessed with GetOCGUsageUserName()
    TOCUserType     UserType;       // The user for whom the OCG is primarily intendet
    UI32            ViewState;      // 0 = Off, 1 = On, PDF_MAX_INT = not set
    float           ZoomMin;        // Minimum zoom factor at which the OCG should be On. -1 = not set
    float           ZoomMax;        // Maximum zoom factor at which the OCG should be On. -1 = not set
};

typedef enum
{
   utIndividual,
   utOrganization,
   utTitle,
   utNotSet
}TOCUserType;
```

The function creates or changes the Content Usage dictionary of an OCG. The structure must be initialzed with InitOCGContUsage() before certain settings of it can be changed. Do not initialize the structure with memset() since this would cause unexpected results!

Notice:

> If no members of the structure were changed after initialization then the Content Usage dictionary of the OCG will be deleted, if it contained already one. Since a Content Usage dictionary is required for OCGs which are included in an application event, the OCG will also be deleted from all application events in which it was included.

The information in this dictionary serves as pure information as long as the OCG is not added to an application event (see AddOCGToAppEvent() for further information).

PDF supports the following application events which can be used to control the visibility of an OCG or layer:

- Export
- Print
- View

Each event has corresponding categories or members in the Content Usage dictionary as well as additional categories, which can be combined with three events, e.g. the viewer language or the current user.

A viewer, that supports these application events, will change the visibility state according to the settings in the Content Usage dictionary.

Remarks:

Note that application events are not supported in all viewer applications. If a watermark should appear when printing, for example, then it is usually best to set the state of such a layer to visible, and to make it invisible in the View event. A viewer that supports application events will hide the layer at viewing time but still print it. A viewer that does not support application events will always display the layer.

If the initial state of the layer would be invisible, then it would also stay invisible at printing time if the viewer does not support application events but layers.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetOCGState

Syntax:

```
LBOOL pdfSetOCGState(
   const PPDF* IPDF, // Instance pointer
   UI32 Handle,      // OCG handle
   LBOOL On,         // Wished state: true = visible, false = invisible
   LBOOL SaveState)  // Save the state in the PDF file?
```

The function changes the visibility state of an OCG or layer. If the parameter *SaveState* is true, the new state is also saved in the PDF file. Otherwise, the state is only changed temporarily so that the wished state can be rendered with RenderPage() or RenderPageToImage() for example.

Note that only the new state of the current OCG will be be saved. If the state of other OCGs were changed beforehand, while *SaveState* was set to false, then these changes will not be considered.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetOnErrorProc

Syntax:

```
LBOOL pdfSetOnErrorProc(
   const PPDF* IPDF,        // Instance pointer
   const void* Data,        // User defined pointer or NULL
   const void* ErrProc)     // Callback function, see below

typedef SI32 PDF_CALL TErrorProc(
   const void* Data,        // User defined pointer
   SI32 ErrCode,            // Error code starting at zero
   const char* ErrMessage,  // Null-terminated error string
   SI32 ErrType);           // Error types, see below

#define PDF_CALL __stdcall // Windows only, otherwise empty
```

The function sets an error callback function which is called by DynaPDF to output error messages and warnings. The parameter *Data* holds a user defined pointer which is passed unchanged to the callback function. If this pointer is not required set it to NULL.

The calling convention of the callback function is standard call under Windows. Note that a wrongly defined calling convention causes an access violation.

The Visual Basic 6 interface uses events instead of a callback function, see Language Binding Visual Basic). In VB .Net you can use events or a an error callback function.

ErrCode is a positive error number starting at zero; it is an index into the array of error messages. The error messages are defined in the file main/drv_base_err_msg.h (this file is included in DynaPDF Enterprise only). *ErrType* is a bitmask to determine what kind of error occurred. The following constants are defined:

```
#define E_WARNING      0x02000000
#define E_SYNTAX_ERROR 0x04000000
```

```
#define E_VALUE_ERROR  0x08000000
#define E_FONT_ERROR   0x10000000
#define E_FATAL_ERROR  0x20000000
#define E_FILE_ERROR   0x40000000
```

At time of publication only one flag is set at any one time. Future versions may be set multiple flags, e.g. E_SYNTAX_ERROR and E_WARNING. Because of this, it is recommended to mask out the error type with a bitwise and operator.

If the callback function returns a value other than zero (0), processing stops immediately.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetOnPageBreakProc

Syntax:

```
LBOOL pdfSetOnPageBreakProc(
    const PPDF* IPDF,        // Instance pointer
    const void* Data,        // User defined pointer or NULL
    const void* OnBreakProc) // Callback function, see below

#define PDF_CALL __stdcall  // Windows only, otherwise empty

typedef SI32 PDF_CALL TOnPageBreakProc(
    const void* Data,        // User defined pointer
    double LastPosX,         // Last x-coordinate of the string
    double LastPosY,         // Last y-coordinate of the string
    LBOOL PageBreak);        // True, if a manual page break occurred
```

The function sets a callback function which is called by WriteFText() when a page break occurred. The parameter *Data* holds a user defined pointer which is passed unchanged to the callback function. If this pointer is not required set it to NULL. The calling convention of the callback function is standard call on Windows.

The Visual Basic and VB .Net interfaces use events instead of a callback function, see Language Binding Visual Basic or VB .Net for further information). In VB .Net you can use a callback function or the event. The OnPageBreakProc event must be connected manually in these interfaces to avoid different runtime behaviour of the function WriteFText(). To enable the event call the function ConnectPageBreakEvent() in VB or VB .Net.

The parameter *PageBreak* of the callback function specifies why the callback function was called:

- If true, a page break tag in the string was found (see WriteFText() for further information).
- If false, the output rectangle was filled with text entirely.

The usage of the callback function is described in detail under WriteFText(). However, the following return values of the callback function can be used the change the text alignment:

```
#define NEW_ALIGN_LEFT    1
#define NEW_ALIGN_RIGHT   2
#define NEW_ALIGN_CENTER  3
#define NEW_ALIGN_JUSTIFY 4
```

A return value of zero indicates that the alignment must not be changed. Negative return values break processing immediately. WriteFText() returns then with a warning.

Remarks:

When the callback function is called, the coordinate system is always bottom-up, it can be changed to top-down coordinates inside the callback function if necessary. The current page is still open, the page must be closed with EndPage() before a new page can be added or opened with Append() or EditPage(). To determine the current page number, call GetPageNum().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetOpacity

Syntax:

```
LBOOL pdfSetOpacity(
   const PPDF* IPDF, // Instance pointer
   double Value)     // 1.0 = opaque, 0.0 = invisible
```

The function sets the opacity value which is used to render the appearance of a text annotation. The parameter Value must be in the range 0.0 to 1.0. If *Value* is zero the annotation is invisible, if *Value* is 1.0, the annotation is completely opaque.

At time of publication the opacity property is used for text annotations only. Later versions of DynaPDF use this property may be for other PDF objects too.

```
Default value = 1.0
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetOrientation

Syntax:

```
LBOOL pdfSetOrientation(
   const PPDF* IPDF, // Instance pointer
   SI32 Value)       // Orientation of the page
```

The function sets the orientation of a page. The parameter *Value* must be a multiple of 90 or 0. Positive values rotate the page clockwise, negative counter counter clockwise.

The page orientation is a property that is applied dynamically at viewing or printing time. It does not change the page's coordinate system or any object in the page description.

To create a true landscape paper format the page's coordinate system must be rotated into the opposite direction. This can be done with SetOrientationEx(). Another way to create a landscape paper format is to change the bounding box of the page, see SetBBox().

```
Default value = 0 // Portrait
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetOrientationEx

Syntax:

```
LBOOL pdfSetOrientationEx(
  const PPDF* IPDF, // Instance pointer
  SI32 Value)       // Orientation of the page
```

The function changes the orientation of a page, incl. important page properties such as the width and height, and the coordinate system. The parameter *Value* must be a multiple of 90 or 0. Positive values rotate the page clockwise, negative values counter clockwise.

The zero point of the coordinate system is set to the upper left or down left point depending on the current page coordinate system (top down or bottom up, see SetPageCoords()).

When using a landscape paper format the functions GetPageWidth() and GetPageHeight() return the logical width and height of the page. While the real paper format is not changed the functions return the paper format as if the page would have a landscape paper format (the width and height are exchanged).

The coordinate system is also changed so that you can work with the page as if it were not rotated. Form fields and annotations are automatically rotated with the page; there is no need to rotate them manually.

Pages which are rotated with this function appear in a viewer application always in the right orientation. Also if the page is rotated, e.g. by 180 degrees the page is shown as if it were not rotated.

Notice:

Since landscape or rotated paper formats require a changed coordinate system, care must be taken into account if further coordinate transformations must be applied. DynaPDF makes sure that one transformation can be applied without causing errors, e.g. rotating or translating the coordinate system, but coordinate transformations must not be nested.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPageBBox

Syntax:

```
LBOOL pdfSetPageBBox(
   IPGE* PagePtr,          // Page pointer return by GetPageObject()
   TPageBoundary Boundary, // The bounding box to set
   struct TFltRect* BBox)  // Required -> The new bounding box
```

The function sets or changes the bounding box of a page. This function is intended to be fast as possible. It requires no open page like SetBBox(), it does not change the default media or crop box, and it does not adjust the current coordinate system if the media box will be changed.

The function is useful if only one or more bounding boxes must be changed, without editing a page, or when a specific area of a page should be rendered. In the latter case change the crop box and restore it to the previous value when finish.

To delete a bounding box set the bounding box to 0, 0, 0, 0. The parameter *BBox* is required to be present, it cannot be NULL.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPageCoords

Syntax:

```
LBOOL pdfSetPageCoords(
   const PPDF* IPDF,       // Instance pointer
   TPageCoord PageCoords) // see below
typedef enum
{
    pcBottomUp = 0, // Bottom-up coordinates
    pcTopDown  = 1  // Top-down coordinates
}TPageCoord;
```

The native coordinate system of the Portable Document Format is bottom-up. However, DynaPDF supports also top-down coordinates to make the usage of the library easier. Top-down coordinates are then converted to bottom-up coordinates by DynaPDF, the coordinate system in the PDF file will not be changed by this function.

When changing the coordinate system to top-down, the coordinate origin of text (font origin) will also be changed to the top-left corner (see also SetFontOrigin()). If the coordinate system will be changed to bottom-up, the font origin is also changed to the down-left corner too.

```
Default value = pcBottomUp
```

Remarks:

The property *PageCoords* is a global property which can be changed at any time. When transforming the coordinate system, bottom-up coordinates must be used.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPageFormat

Syntax:

```
LBOOL pdfSetPageFormat(
    const PPDF* IPDF,  // Instance pointer
    TPageFormat Value) // see below
typedef enum
{
    pfDIN_A3     = 0,
    pfDIN_A4     = 1,
    pfDIN_A5     = 2,
    pfDIN_B4     = 3,
    pfDIN_B5     = 4,
    pfDIN_B6     = 5,
    pfDIN_C3     = 6,
    pfDIN_C4     = 7,
    pfDIN_C5     = 8,
    pfDIN_C6     = 9,
    pfDIN_C65    = 10,
    pfDIN_DL     = 11,
    pfDIN_E4     = 12,
    pfDIN_E5     = 13,
    pfDIN_E6     = 14,
    pfDIN_E65    = 15,
    pfDIN_M5     = 16,
    pfDIN_M65    = 17,
    pfUS_Legal   = 18,
    pfUS_Letter  = 19,
    pfDIN_A0     = 20,
    pfDIN_A1     = 21,
    pfDIN_A2     = 22,
    pfDIN_A6     = 23,
    pfDIN_A7     = 24,
    pfDIN_A8     = 25,
    pfDIN_A9     = 26,
    pfDIN_A10    = 27
}TPageFormat;
```

The function sets a predefined page or paper format. If an open page can be detected the format of the page will be changed, the new format will also be used for newly created pages. If no open page is detected, the default format for new pages will be set only. This function changes the media box of a page (see SetBBox() for further information).

Paper formats:

Format	Size in PDF Units (Width x Height) -> 1 PDF Unit = 1/72 inch
pfDIN_A0	2384.0 x 3370.0
pfDIN_A1	1684.0 x 2384.0
pfDIN_A2	1191.0 x 1684.0
pfDIN_A3	842.0 x 1191.0
pfDIN_A4	595.0 x 842.0
pfDIN_A5	419.0 x 595.0
pfDIN_A6	298.0 x 420.0

```
pfDIN_A7          |  210.0 x  298.0
pfDIN_A8          |  147.0 x  210.0
pfDIN_A9          |  105.0 x  147.0
pfDIN_A10         |   74.0 x  105.0
pfDIN_B4          |  709.0 x 1001.0
pfDIN_B5          |  499.0 x  709.0
pfDIN_B6          |  354.0 x  499.0
pfDIN_C3          |  918.0 x 1298.0
pfDIN_C4          |  649.0 x  918.0
pfDIN_C5          |  459.0 x  649.0
pfDIN_C6          |  323.0 x  459.0
pfDIN_C65         |  323.0 x  649.0
pfDIN_DL          |  312.0 x  624.0
pfDIN_E4          |  623.0 x  879.0
pfDIN_E5          |  439.0 x  624.0
pfDIN_E6          |  312.0 x  439.0
pfDIN_E65         |  312.0 x  624.0
pfDIN_M5          |  439.0 x  632.0
pfDIN_M65         |  317.0 x  632.0
pfUS_Legal        |  612.0 x 1008.0
pfUS_Letter       |  612.0 x  792.0
Default value = pfDIN_A4
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPageHeight

Syntax:

```
LBOOL pdfSetPageHeight(
    const PPDF* IPDF, // Instance pointer
    double Value)     // New height
```

The function changes the height of the currently open page if any, or the default height for newly created pages. This function changes the media box of a page (see also SetBBox()). The page height must be greater zero.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPageLayout

Syntax:

```
LBOOL pdfSetPageLayout(
    const PPDF* IPDF,   // Instance pointer
    TPageLayout Layout) // see below

typedef enum
{
    plSinglePage     = 0, // Show one page at time
    plOneColumn      = 1, // Show the pages continous
    plTwoColumnLeft  = 2, // Two columns, start with left column
    plTwoColumnRight = 3, // Two columns, start with right column
    plTwoPageLeft    = 4, // PDF 1.5
    plTwoPageRight   = 5, // PDF 1.5
    plDefault        = 6  // Use viewer's default settings
}TPageLayout;
```

The function sets the page layout that is used when opening the document with Adobe's Acrobat.

```
Default value = plOneColumn
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPageMode

Syntax:

```
LBOOL pdfSetPageMode(
    const PPDF* IPDF, // Instance pointer
    TPageMode Mode)   // see below

typedef enum
{
    pmUseNone        = 0, // Default
    pmUseOutlines    = 1, // Show the outline tree
    pmUseThumbs      = 2, // Show the thumb nails
    pmFullScreen     = 3  // Open the document in full-screen mode
    pmUseOC          = 4, // PDF 1.5
    pmUseAttachments = 5  // PDF 1.6
}TPageMode;
```

The function sets the page mode that is used when opening the document with Adobe's Acrobat.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPageOrientation

Syntax:

```
LBOOL pdfSetPageOrientation(
   const PPDF* IPDF, // Instance pointer
   IPGE* PagePtr,    // Page pointer returned by GetPageObject()
   SI32 Value);      // New value
```

The function changes the orientation of page. This function is intended to be fast as possible. It requires no open page like SetOrientation() and accesses the property directly.

The parameter *PagePtr* must be a valid pointer returned by GetPageObject().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPageWidth

Syntax:

```
LBOOL pdfSetPageWidth(
   const PPDF* IPDF, // Instance pointer
   double Value)     // New page width
```

The function changes the width of the currently open page if any, or the default width for newly created pages. This function changes the media box of a page (see also SetBBox()). The page width must be greater zero.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPDFVersion

Syntax:

```
LBOOL pdfSetPDFVersion(
    const PPDF* IPDF,     // Instance pointer
    TPDFVersion Version) // Output PDF version

typedef enum TPDFVersion
{
    pvPDF_1_0     = 0,   // PDF 1.0
    pvPDF_1_1     = 1,   // PDF 1.1
    pvPDF_1_2     = 2,   // PDF 1.2
    pvPDF_1_3     = 3,   // PDF 1.3
    pvPDF_1_4     = 4,   // PDF 1.4
    pvPDF_1_5     = 5,   // PDF 1.5 -> default
    pvPDF_1_6     = 6,   // PDF 1.6
    pvPDF_1_7     = 7,   // PDF 1.7
    pvPDF_2_0     = 8,   // PDF 2.0
    pvReserved    = 9,   // Reserved for future use
    pvPDFX1a_2001 = 10,  // PDF/X-1a:2001
    pvPDFX1a_2003 = 11,  // PDF/X-1a:2003
    pvPDFX3_2002  = 12,  // PDF/X-3:2002
    pvPDFX3_2003  = 13,  // PDF/X-3:2003
    pvPDFA_2005   = 14,  // PDF/A-1b
    pvPDFX_4      = 15,  // PDF/X-4
    pvPDFA_1a     = 16,  // PDF/A 1a
    pvPDFA_2a     = 17,  // PDF/A 2a
    pvPDFA_2b     = 18,  // PDF/A 2b
    pvPDFA_2u     = 19,  // PDF/A 2u
    pvPDFA_3a     = 20,  // PDF/A 3a
    pvPDFA_3b     = 21,  // PDF/A 3b
    pvPDFA_3u     = 22,  // PDF/A 3u
    // The following constants are flags which can be combined with pvPDFA_3a, pvPDFA_3b, and
    // pvPDFA_3u. If used stand alone PDF/A 3b with the correspondig ZUGFeRD metadata will be
    // created.
    pvZUGFeRD_Basic    = 0x00010000,
    pvZUGFeRD_Comfort  = 0x00020000,
    pvZUGFeRD_Extended = 0x00040000,
    // Factur-X / ZUGFeRD 2.1, and ZUGFeRD 2.0 profiles. The usage is identically in comparision
    // to the older ZUGFeRD 1.0 flags. That means if used alone, a PDF/A 3b file will be created.
    // ZUGFeRD 2.1 and Factur-X are identically defined in PDF. In order to create a ZUGFeRD 2.1
    // invoice, set one of the Factur-X constants but don't set the flag pvZUGFeRD2_Flag.
    pvFacturX_Minimum  = 0x00080000, // Minimum profile
    pvFacturX_Basic    = 0x00100000, // Basic profile
    pvFacturX_Basic_WL = 0x00200000, // Basic WL profile
    pvFacturX_Comfort  = 0x00400000, // EN 16931 profile
    pvFacturX_Extended = 0x00800000, // Extended profile
    pvZUGFeRD2_Flag    = 0x01000000, // If set, a ZUGFeRD 2.0 will be created.
    pvPDFX5g           = 23,         // PDF/X-5g
    pvPDFX5n           = 24,         // PDF/X-5n
    pvPDFX5pg          = 25,         // PDF/X-5pg
    pvPDFX6            = 26,         // PDF/X 6
    pvPDFX6n           = 27,         // PDF/X 6n
    pvPDFX6p           = 28,         // PDF/X 6p
    pvPDFUA1           = 29,         // PDF/UA-1 (ISO 14289-1)
    pvPDFX4p           = 30,         // PDF/X-4p
    pvPDFVT1           = 0x02000000, // PDF/VT 1 flag. The base version must be a PDF/X version.
    pvPDFVT2           = 0x04000000, // PDF/VT 2 flag. The base version must be a PDF/X version.
    pvPDFVT3           = 0x08000000, // PDF/VT 3 flag. The base version must be a PDF/X version.
    pvPDFVTMask        = pvPDFVT1 | pvPDFVT2 | pvPDFVT3,
    pvPDFUAFlag        = 0x20000000, // PDF/UA flag. Can be used in addition with other standards.
                                     // File metadata will contain PDF/A and PDF/UA entries.
    pvPDFA_4           = 31,         // PDF/A 4.
    pvPDFA_4e          = 32,         // PDF/A 4e allows 3D contents in rich media annotations.
    pvPDFA_4f          = 33          // PDF/A 4f allows file attach annotations.
}TPDFVersion;
```

The function changes the output file version. The default output version is PDF 1.4. If a PDF file with a higher version as curently set will be imported, then the file version is adjusted to the version of the imported file.

Note that not all available versions can be created by DynaPDF. However, DynaPDF is able to identify all versions listed above when importing a PDF file. A known version will not be stripped out when saving a PDF file with DynaPDF.

To improve processing speed, the output version is **not** checked at runtime and no features are disabled which are maybe not supported by the current PDF version.

However, if a PDF file must be compatible to a specific Acrobat version, change the version and open the file in Adobe's Acrobat. If error messages or other warnings appear, the file contains unsupported features. For example, Unicode output is supported since PDF 1.3. If the output version will be changed to PDF 1.2, you will get an error message in Adobe's Acrobat if the file contains CID fonts (Unicode fonts). There is no other known feature that causes errors when opening a PDF file with Acrobat or Reader.

To create PDF/X or PDF/A compatible files we recommended to use a preflight tool to check whether unsupported features were used or whether additional settings are required to meet the requirements of the standard. The PDF version should be set at the end of processing, directly before CloseFile() or CloseFileEx() is called. See also PDF/X and PDF/X Compatibility.

ZUGFeRD, ZUGFeRD 2.0, Factur-X / ZUGFeRD 2.1/2.2

ZUGFeRD 2.0 and Factur-X are almost identically defined. Both standards support the very same profiles. To create a ZUGFeRD 2.0 invoice with the profile Basic WL, for example, set the PDF version as follows:

```
pdfSetPDFVersion(pdf, TPDFVersion(pvFacturX_Basic_WL | pvZUGFeRD2_Flag));
```

The PDF version is set to PDF/A 3b in the above example. If you want to create the very same invoice but as PDF/A 3u file, for example, then set also the whished output PDF version:

```
pdfSetPDFVersion(pdf, TPDFVersion(pvPDFA_3u | pvFacturX_Basic_WL | pvZUGFeRD2_Flag));
```

Note that this function just sets the PDF version. It does not check whether the resulting file is compatible to the whished output version. Use CheckConformance() to convert arbitrary PDF files to PDF/A, ZUGFeRD, or Factur-X files.

ZUGFeRD 2.1 or higher

ZUGFeRD is an ever growing standard that is updated every few months. Important to know is that beginning with ZUGFeRD 2.1 Factur-X and ZUGFeRD are identically defined in PDF. Therefore, the Factur-X constants can be used to create Factur-X and ZUGFeRD 2.1 or higher files. The PDF container cannot distiguish between these formats.

ZUGFeRD and Factur-X invoices require an embedded XML invoice. The XML file can be attached with AttachFile() or AttachFileEx().

The XML invoice must be named as follows (case-sensitive and without quotes):

- `ZUGFeRD 1.0:`   `"ZUGFeRD-invoice.xml"`
- `ZUGFeRD 2.0:`   `"zugferd-invoice.xml"`
- `ZUGFeRD 2.1 or higher:`   `"factur-x.xml"`
- `Factur-X:`   `"factur-x.xml"`

The embedded invoice must also be associated with the *Catalog* object. This can be done with AssociateEmbFile(). The parameter *RelationShip* must be set to *arAlternate*, *arData*, or *arSource* depending on the used Factur-X or ZUGFeRD profile. See CheckConformance() for further information.

Examples:

```
pdfSetPDFVersion(pdf, pvZUGFeRD_Basic);                                    // ZUGFeRD 1.0
pdfSetPDFVersion(pdf, TPDFVersion(pvFacturX_Basic_WL | pvZUGFeRD2_Flag)); // ZUGFeRD 2.0
pdfSetPDFVersion(pdf, pvFacturX_Basic_WL);                                 // ZUGFeRD 2.1 or Factur-X
pdfSetPDFVersion(pdf, pvFacturX_XRechnung);                                // XRechnung
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetPrintSettings

Syntax:

```
LBOOL pdfSetPrintSettings(
    const PPDF* IPDF,             // Instance pointer
    TDuplexMode Mode,            // See below
    SI32 PickTrayByPDFSize,      // 1 = true, 0 = false, -1 = app default
    UI32 NumCopies,              // 0 = app default, max = 5
    TPrintScaling PrintScaling,  // See below
    UI32* PrintRanges,           // Optional start/end page number pairs
    UI32 NumRanges)              // Number of ranges

typedef enum
{
    dpmNone, // Default
    dpmSimplex,
    dpmFlipShortEdge,
    dpmFlipLongEdge
}TDuplexMode;

typedef enum
{
    psAppDefault, // Default
    psNone
}TPrintScaling;
```

The function adds preferred print settings to the PDF file. Viewer applications use the print settings to initialize the print dialog. Print ranges, if set, consist of two numbers which define the start and end page number that should be printed.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetProgressProc

Syntax:

```
LBOOL pdfSetProgressProc(
    const PPDF* IPDF,            // Instance pointer
    const void* Data,           // User defined pointer
    const void* InitProgress,   // Initialization callback function
    const void* Progress)       // Progress callback function

typedef SI32 TProgType;
#define ptImportPage  0 // Start page import
#define ptWritePage   1 // Start writing a page to file or buffer
#define ptPrintPage   2 // Start printing the PDF file
#define ptConvertPage 3 // Start converting the PDF file (Optimize(), CheckConformance())

#define PDF_CALL __stdcall   // Windows only, otherwise empty

typedef void PDF_CALL TInitProgressProc(
    const void* Data,       // User defined pointer
    TProgType ProgType,     // Current process, see above
    SI32 MaxCount);         // The number of callback function calls

typedef SI32 PDF_CALL TProgressProc(
    const void* Data,        // User defined pointer
    SI32 ActivePage);        // Current page number
```

The function sets two callback functions which can be used to control a progress bar. This function is absent in the Visual Basic interface because this interface uses events instead of callback functions.

The progress callback functions are called by these functions:

- CheckConformance()
- ImportPDFFile()
- Optimize()
- CloseFile(), CloseFileEx(), and so on

The parameter *Data* is a user defined pointer that is passed unchanged to the callback functions.

The parameter *InitProgress* defines a callback function that is called before the progress callback function is called the first time. The parameters of this callback function can be used to initialize a progress bar. The parameter *ProgType* of the initialization callback function defines the current process; this can be either importing, writing, or converting a page.

Other operations do not call the progress callback functions because most operations are too fast as if they could be used to display the current progress.

The parameter *MaxCount* defines how often the progress callback function will be called. To disable the progress callback functions set the parameters *InitProgress* and *Progress* to NULL.

Return values of the progress callback function:

The return value of the progress callback function must be 0, any other return value breaks processing. All functions, except ImportPDFFile(), release the PDF file in memory if a non-zero value will be returned.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetRenderingIntent

Syntax:

```
LBOOL pdfSetRenderingIntent(
    const PPDF* IPDF,        // Instance pointer
    SI32 ImgHandle,          // Image handle
    TRenderingIntent Value)  // Rendering intent
```

Set function sets the rendering intent of an image. The rendering intent describes how colors should be interpreted.

The parameter ImgHandle must be a valid image handle that was returned by InsertImage() or InsertImageEx(). Imported images returned by ParseContent(), for example, can be changed too.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetResolution

Syntax:

```
LBOOL pdfSetResolution(
   const PPDF* IPDF, // Instance pointer
   UI32 Value)       // Image resolution in DPI
```

The function sets the resolution in DPI (Dots per Inch), in which images are stored by DynaPDF. The property will be ignored if the property *SaveNewImageFormat* was set to false. 1 bit images are always stored in the original resolution. The image resolution must be in the range 1 to 12000 DPI.

```
Default value = 150 (DPI)
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetSaveNewImageFormat

Syntax:

```
LBOOL pdfSetSaveNewImageFormat(
   const PPDF* IPDF, // Instance pointer
   LBOOL Value)      // If true, images are downscaled if necessary
```

The property *SaveNewImageFormat* specifies whether images should be downscaled if the original resolution of the image is higher than the value of the property resolution (see also SetResolution()). If the property SaveNewImageFormat is false, JPEG compressed images will be inserted in pass-through mode as long as the image is not a CMYK JPEG (such images must be converted because Acrobat viewer require another byte order).

```
Default value = true
```

SetScreenRes (Rendering Engine)

Syntax:

```
void rasSetScreenRes(
   IRAS* RasPtr, // Pointer of the rasterizer
   UI32 Value)   // Screen resolution in DPI
```

The function sets the screen resolution. The screen resolution is sometimes required, e.g. to render Text annotations correctly. This type of annotation has a fixed size that is independent of the current scaling factor. However, the correct size depends on the screen resolution on which the image is rendered.

SetSeparationInfo

Syntax:

```
LBOOL pdfSetSeparationInfo(
    const PPDF* IPDF, // Instance pointer
    UI32 Handle)      // Separation color space handle
```

The function sets the separation info of the current open page. The parameter *Handle* must be a valid handle of a Separation color space.

Remarks:

In high-end printing workflows, pages are ultimately produced as sets of separations, one per colorant. Ordinarily, each page in a PDF file is treated as a composite page that paints graphics objects using all the process colorants and perhaps some spot colorants as well. In other words, all separations for a page are generated from a single PDF description of that page.

In some workflows, however, pages are pre-separated before generating the PDF file. In a pre-separated PDF file, the separations for a page are described as separate page objects, each painting only a single colorant (usually specified in the DeviceGray color space). When this is done, additional information is needed to identify the actual colorant associated with each separation and to group together the page objects representing all the separations for a given page. This information is contained in the separation info of each page object.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetSpaceWidthFactor

Syntax:

```
LBOOL pdfSetSpaceWidthFactor(
    const PPDF* IPDF,
    double Value)
```

The function sets the space width factor that is used by ExtractText() to determine whether the distance between two text records or glyphs should be interpreted as space character.

Background:

PDF files do often not contain space characters. Depending on the font type, i.e. fixed pitch or proportional fonts, different default widths are used during text extraction. However, the full space width is mostly too large to find spaces. Therefore, the text extraction algorithm multiplies the default space width with the space width factor to find suitable values.

The default value is 0.7. Increase the value if too many spaces occur in extracted text and decrease the value if too few spaces were found.

The value must be large zero. Although the upper bound is not restricted, values larger than 2 are mostly not meaningful.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetStrokeColor

Syntax:

```
LBOOL pdfSetStrokeColor(
   const PPDF* IPDF, // Instance pointer
   UI32 Color)       // Color value defined in the current color space
```

The function sets the stroke color. The stroke color is also used by text objects if the text rendering mode is dmStroke or dmFillStroke (see SetTextDrawMode() for further information).

The color value must be defined in the current color space. CMYK colors can be constructed with the macro PDF_CMYK() or with the function CMYK() which is available in most programming languages. RGB colors can be constructed with the macro PDF_RGB() or with the function RGB() which is available in most programming languages.

If the color space contains more than 4 color components use SetStrokeColorEx() instead.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetStrokeColorEx

Syntax:

```
LBOOL pdfSetStrokeColorEx(
  const PPDF* IPDF,   // Instance pointer
  const BYTE* Color,  // Color to be set
  UI32 NumComponents) // Number of componnents
```

The function sets the stroke color. The color must be defined as an array of bytes in the logical order of the color space. For example, if the color space is DeviceRGB the array must specify the color values of the red, green, and blue components in that order. The number of components must be equal to the one of the corresponding color space.

Lab colors can be defined as signed char as usual. Make a typecast to BYTE* when passing the color to the function. See CreateCIEColorSpace() for further information.

Example (C/C++):

```
...
char labColor[3] = {50, -34, 77}; // L, *a, *b
pdfSetStrokeColorEx(pdf, (BYTE*)labColor, 3);
...
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetStrokeColorF

Syntax:

```
LBOOL pdfSetStrokeColorF(
    const PPDF* IPDF,    // Instance pointer
    const float* Color, // Array of float values
    UI32 NumComponents) // Must be equal to the underlying color space
```

The function sets the current stroke color as an array of float values. The components of non-Lab color spaces must be in the range from 0 through 1. The *a and *b components of a Lab color space are typically in a range from -128 though 127. The *L component ranges from 0 through 100.

The number of components must match the number of components of the underlying color space.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetStrokeColorSpace

Syntax:

```
LBOOL pdfSetStrokeColorSpace(
   const PPDF* IPDF,  // Instance pointer
   TPDFColorSpace CS) // Color space
```

The function changes the stroke color space. In PDF, fill and stroke colors use both their own color spaces. Although it is possible to use different color spaces for strokes and fillings it should be avoided if possible. The fill color space is the relevant color space when creating interactive objects such as form field or annotations. See SetFillColorSpace().

Notice:

This function was added to DynaPDF primarily for testing purposes. The color space should be set with SetColorSpace() which sets always the same color space for fillings and strokes.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetTabLen

Syntax:

```
LBOOL pdfSetTabLen(
   const PPDF* IPDF, // Instance pointer
   SI32 TabLen)       // New tabulator length in number of spaces
```

The function sets the tabulator length, specified in number of spaces, which will be used to emulate tabulators during text formatting (see WriteFText() for further information).

Because tabulators are emulated with spaces they have no fixed with. The width of a tabulator depends on the width of the space character of the active font.

```
Default value = 3 (min = 1, max = 256)
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetTemplBBox

Syntax:

```
LBOOL pdfSetTemplBBox(
    const PPDF* IPDF,        // Instance pointer
    UI32 Handle,             // Template handle
    TPageBoundary Boundary,  // pbMediaBox or pbCropBox
    struct TPDFRect* BBox)   // New bounding box or NULL to delete it
```

The function sets or changes a bounding box of a template. Templates support the bounding boxes pbMediaBox and pbCropBox only. The latter one is considered by PlaceTemplateEx() only.

A crop box can be set to clip the contents of a template without changing its bounding box. This can be useful when placing imported pages on another page.

If *BBox* is set to NULL, the bounding box is deleted. Note that only the crop box can be deleted.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetTextDrawMode

Syntax:

```
LBOOL pdfSetTextDrawMode(
    const PPDF* IPDF, // Instance pointer
    TDrawMode Mode)   // see below
typedef enum
{
    dmNormal        = 0, // Default
    dmStroke        = 1, // Stroke the text (outlines only)
    dmFillStroke    = 2, // Fill and stroke the text
    dmInvisible     = 3, // Neither fill nor stroke text (invisible)
    dmFillClip      = 4, // Fill text and add to path for clipping
    dmStrokeClip    = 5, // Stroke text and add to path for clipping
    dmFillStrokeClip = 6, // Fill & stroke text and add to path for clipping
    dmClipping      = 7  // Add the text to path for clipping
}TDrawMode;
```

The text draw mode specifies how text should be rendered. Texts can be used as clipping paths such as normal vector graphics; however, the usage is not the same. To use a text as clipping path save the graphics state, set the text draw mode to a clipping mode, paint the objects which should be clipped into the path and restore the graphics state.

Note that the functions BeginClipPath() and ClipPath() can **not** be used to define a text as clipping path (see the example below).

```
Default value = dmNormal
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Draw Mode	Description
dmNormal	Default, render the text in the usual way. The fill color is used as the text's color.
dmStroke	Tread the glyph outlines as path to be stroked. The current line width and stroke color are used to paint the path.
dmFillStroke	Tread the glyph outlines as path to be stroked and filled. The current line width, fill color, and stroke color are used to paint the path.
dmInvisible	Neither fill nor stroke the text, the text is invisible.
dmFillClip	Tread the glyph outlines as path to be filled and add this path to the current clipping path. The settings of the current line width and fill color are used to paint the path.
dmStrokeClip	Tread the glyph outlines as path to be stroked and add this path to the current clipping path. The settings of the current line width and stroke color are used to paint the path.
dmFillStrokeClip	Tread the glyph outlines as path to be stroked and filled and add this path to the current clipping path. The settings of the current line width, fill color and stroke color are used to paint the path.
dmClipping	Tread the glyph outlines as path and add this path to the current clipping path.

How to use text as clipping path?

As mentioned earlier the usage of text as clipping path is not the same as a normal clipping path. The following example shows how a text can be used for clipping:

Example (C++):

```cpp
#include "dynapdf.h"
using namespace DynaPDF;
// First we declare an error callback function.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0; // We ignore non-fatal errors
}

int main(int argc, char* argv[])
```

```cpp
{
   void* pdf = pdfNewPDF();
   if (!pdf) return 2; // Out of memory?
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   // The document info entries are changed by the function if set
   pdfSetDocInfo(pdf, diSubject, "Text as clipping path");
   pdfSetDocInfo(pdf, diCreator, "C++ example test project");
   pdfSetDocInfo(pdf, diTitle, "Text as clipping path");
   pdfSetPageCoords(pdf, pcTopDown);

   pdfCreateNewPDF(pdf, "c:/cppout.pdf");

   pdfAppend(pdf);
   pdfSetFont(pdf, "Bookman Old Style", fsBold, 80, true, cp1252);
   // We want to draw an axial shading into the clipping path
   SI32 sh = pdfCreateAxialShading(pdf, 0, 50, 0, 130, 1, PDF_BLUE,
                     PDF_YELLOW, false, false);
   // Do not forget to save the graphics state
   pdfSaveGraphicState(pdf);
   pdfSetTextDrawMode(pdf, dmClipping); // Use the text as clipping path
   pdfWriteText(pdf, 50, 50 - (80 - pdfGetCapHeight()), "Clipping");
   pdfApplyShading(pdf, sh);
   // Restore the graphics state now to disable the clipping path
   pdfRestoreGraphicState(pdf);
   EndPage(pdf);

   pdfCloseFile(pdf);
   pdfDeletePDF(pdf); // Do not forget to delete the PDF instance
   return 0;
}
```

Output:

SetTextFieldValue

Sytnax:

```
SI32 pdfSetTextFieldValue(
   const PPDF* IPDF,       // Instance pointer
   UI32 Field,             // Text field handle
   const char* Value,      // Field's value or NULL
   const char* DefValue,   // Field's default value or NULL
   TTextAlign Align)       // Text alignment
```

The function sets or changes the value and default value of a text field. The parameters *Value* and *DefValue* can both be NULL, if both values should be deleted. The parameter *DefValue* defines the default value of the field, this value is shown in Adobe's Acrobat as long as the field has no value; it will be hidden when entering the field.

The strings are drawn with the font that is associated with the field. The font size is taken from the field settings. If set to "auto", the font size is computed from the field's bounding box.

Field format

Text fields can be formatted to certain number formats with Javascript actions that are executed in the *OnFormat* event. DynaPDF has limited support for the Javascript functions *AFNumber_Format()*, *AFPercent_Format()*, and *AFSpecial_Format()*.

However, the results of number formats of Adobes Acrobat and DynaPDF are sometimes slightly different. It is not known whether this is a Javascript or Acrobat issue, but the results shown below can be reproduced with all known Acrobat versions.

Example number format with two fractional digits:

Value	Acrobat	DynaPDF
1.45499	1.45	1.45
1.454991	1.46	1.45
1.4549901	1.46	1.45
1.45499001	1.46	1.45
5.0049	5.00	5.00
5.00499	5.01	5.00
5.004989	5.00	5.00

It is possible to deactivate all kinds of number formats by setting the flag *gfDisableJavascript* with SetGStateFlags(). The appearance stream of text fields is then created with the raw field value.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetTextFieldValueEx

Syntax:

```
LBOOL pdfSetTextFieldValueEx(
    const PPDF* IPDF,  // Instance pointer
    UI32 Field,        // Text field handle
    const char* Value) // New value or NULL to delete it
```

The function changes or sets the value of a text field. If the value is set to NULL, the field's value will be deleted.

The strings are drawn with the font that is associated with the field. The font size is taken from the field settings. If set to "auto", the font size is computed from the field's bounding box.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

SetTextRect

Syntax:

```
LBOOL pdfSetTextRect(
    const PPDF* IPDF, // Instance pointer
    double PosX,      // X-Coordinate of output rectangle
    double PosY,      // Y-Coordinate of output rectangle
    double Width,     // Width of output rectangle
    double Height)    // Height of output rectangle or -1
```

The function defines the output rectangle that is used to output formatted text by the function WriteFText(). The text is not clipped into the rectangle. WriteFText() executes a callback if defined (see SetOnPageBreakProc()), when the rectangle was filled with text and more text is available for output. If no callback function is used, the text continues on the next page by using the same output rectangle. The usage of the function is described in detail under WriteFText().

The parameter *Height* can be set to **-1** to determine that no page break should occur.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetTextRise

Syntax:

```
LBOOL pdfSetTextRise(
    const PPDF* IPDF, // Instance pointer
    double Value)     // Text rise in units
```

Text rise specifies the distance, to move the baseline up or down from its default location. Positive values of text rise move the baseline up. Adjustments to the baseline are useful for drawing superscripts or subscripts. The default location of the baseline can be restored by setting the text rise

to 0. The figure below illustrates the effect of the text rise. Text rise always applies to the vertical coordinate in text space.

The text moves around the baseline.

```
Default value = 0
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetTextScaling

Syntax:

```
LBOOL pdfSetTextScaling(
   const PPDF* IPDF, // Instance pointer
   double Value)     // Horizontal text scaling
```

The scaling value adjusts the width of glyphs by stretching or compressing them in the horizontal direction. Its value is specified as a percentage of the normal width of the glyphs, with 100 being the normal width.

```
Default value = 100
```

Value = 100	Word
Value = 50	WordWord

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetTransparentColor

Syntax:

```
LBOOL pdfSetTransparentColor(
   const PPDF* IPDF, // Instance pointer
   UI32 AColor)      // Color value defined in current color space
```

The function sets the transparent color which is used for newly inserted images. Due to color interpolation during decompression of JPEG images, it not always possible to mask JPEG compressed images correctly.

```
Default value = 0x00FFFFFF (RGB White)
```

SetTrapped

Syntax:

```
void pdfSetTrapped(
  const PPDF* IPDF, // Instance pointer
  LBOOL Value)      // True or false
```

The function sets the trapped key of the document. Trapping is a technique to avoid unwanted visual artifacts such as brightly colored gaps or bands around the edges of printed objects. In high-quality reproduction of color documents, such artifacts are commonly avoided by creating an overlap, called a trap, between areas of adjacent color.

The trapped key determines whether the document was trapped. The default value is unknown that means no value is written to the file. The trapped key should be set if possible.

SetUseExactPwd

Syntax:

```
LBOOL pdfSetUseExactPwd(
   const PPDF* IPDF, // Instance pointer
   LBOOL Value)      // True or false
```

If the property *UseExactPwd* is false, an encrypted PDF file can always be decrypted, if either the open or owner password in the file is an empty string. If true, the open or owner password must be known to open the PDF file.

```
Default value = 1 (true)
```

Remarks:

If your application should allow the modification of encrypted PDF files, you may check the access permissions to grant user rights, if the file was opened with the open password instead of the owner password (see also GetUserRights()).

Due to the license agreement of Adobe, all manufacturers of applications which make the treatment of encrypted PDF files possible, must respect the access permissions of a PDF file, if the file was opened with the open password.

Only if the file was opened with the owner password, all rights should be granted. See *PDF Reference 1.5* for further information. This document is available at www.adobe.com.

The property *UseExactPwd* should be true, if the application is a commercial product.

SetUseGlobalImpFiles

Syntax:

```
LBOOL pdfSetUseGlobalImpFiles(
   const PPDF* IPDF, // Instance pointer
   LBOOL Value)      // True or false
```

The property can be used to load one or more external PDF files permanent into memory, e.g. to split a large PDF file into smaller pieces. The next PDF file that will be opened with the function OpenImportFile() or OpenImportBuffer() will not be closed when CloseFile() or FreePDF() is called. When creating a new PDF file the previously opened import file is still open and it is possible to import pages from this file without loading the file again.

If more than one file should be loaded permanent into memory then set also the flag if2UseProxy with SetImportFlags2(). PDF files which are no longer needed can be closed with CloseImportFile() or CloseImportFileEx().

Return values:

This function cannot fail, the return value is always 1.

Example (Delphi):

In this example we have a large PDF file that should be splitted into separate files which should contain one page each.

If we would not use the property *UseGlobalImpFiles* we must read the entire file structure each time a page was extracted because the import file would be closed when CloseFile() is called. Splitting a large file in this way would be extremely slow. To improve processing speed we set the property *UseGlobalImpFiles* to true so that we must open the import file only one time.

```
// This is our error callback function
function ErrProc(const Data: Pointer; ErrCode: Integer; const ErrMessage:
PAnsiChar; ErrType: Integer): Integer; stdcall;
var s: String;
begin
   s := ErrMessage + #13#13'Abort processing?';
   if MessageDlg(s, mtError, [mbYes, mbNo], 0) = mrYes then
      Result := 1
   else
      Result := 0;
end;

procedure TForm1.Button2Click(Sender: TObject);
var pdf: TPDF; i: Integer;
begin
   pdf := nil;
   try
      pdf := TPDF.Create;
      pdf.SetOnErrorProc(nil, @ErrProc);
```

```
      pdf.SetUseGlobalImpFiles(true);
      // No conversion to templates is required. In addition, we need the
      // page contents only.
      pdf.SetImportFlags(ifImportAsPage);
      pdf.OpenImportFileA('c:/test.pdf', ptOpen, '');
      // In the following loop we create a new PDF file, import one page
      // and close the new file. We continue until all pages are stored
      // in a new PDF file each.
      // We store the files in the directory c:/SplitOut. Note that the
      // size of all single page files is many times larger as the size
      // of the original import file because used fonts and other
      // resources must be exported to each file, they cannot be shared.
      for i := 1 to pdf.GetInPageCount do begin
         pdf.CreateNewPDFA(Format('c:/SplitOut/split%.4d.pdf', [i]));
         pdf.Append;
         pdf.ImportPageEx(i, 1.0, 1.0); // Import a page
         pdf.EndPage;
         pdf.CloseFile;
      end;
   except
      on E: Exception do MessageDlg(E.Message, mtError, [mbOK], 0);
   end;
   if pdf <> nil then pdf.Free;
end;
```

SetUseImageInterpolation

Syntax:

```
LBOOL pdfSetUseImageInterpolation(
   const PPDF* IPDF, // Instance pointer
   UI32 Handle,      // Image handle
   LBOOL Value)      // Value
```

The function enables or disables image interpolation explictely for a given image. If nothing was specified (this is the default), a viewer application can only use a heuristic to determine whether image interpolation should be enabled or not. The result is then of course application specific.

The parameter *Handle* must be a valid image handle that was returned by a DynaPDF function like InsertImage(), InsertImageEx(), or InsertImageFromBuffer(), for example.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetUseImageInterpolationEx

Syntax:
```
LBOOL pdfSetUseImageInterpolationEx(
    PIMG* Image, // Image pointer
    LBOOL Value) // Value
```

The function enables or disables image interpolation explictely for a given image exactly like SetUseImageInterpolation() but accepts an image pointer as input. The pointer of an image object is provided in the TInsertImage callback function of ParseContent().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetUserUnit

Syntax:
```
LBOOL pdfSetUserUnit(
    const PPDF* IPDF, // Instance pointer
    float Value)      // Must be in the range 0.01..75.0
```

The function sets the user unit of the current open page. A user unit is a scaling factor. The page format and all page coordinates are multiplied with this factor in a viewer application. The default size of a PDF unit is 1/72 inch. User units can be useful if the page format would be too large to be expressed in standard PDF units. The largest page format in PDF is limited to 14,400 units or 200 inches. This limit can be extended with the user unit.

The largest value that is supported is 75.0 which results in a maximum page format of 15,000 x 15,000 inches or 1,800,000 units. Note that all functions which return page coordinates or page properties do **not** consider the user unit.

The page format must still be in the range 3..14400 units. It is also strongly recommended to set the user unit only if necessary. This is only the case if the required page format is larger 14400 units.

```
Default value = 1.0
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetUseStdFonts

Syntax:

```
LBOOL pdfSetUseStdFonts(
   const PPDF* IPDF, // Instance pointer
   LBOOL Value)      // If false, the 14 standard fonts are disabled
```

The function can be used to disable the 14 standard fonts temporarily. PDF viewer applications support 14 standard fonts, these fonts are not embedded by default, also if the parameter Embed of the function SetFont() is true.

If certain standard fonts are available on the system, then they have typically a higher search priority but it is still possible to select a standard font. This can occur, for example, if the font weight of the system font does not exactly match the requested weight. If a standard font represents a better match, then this font would be selected.

To avoid unwanted font selections it is possible to disable the standard fonts from font selection. The value can be changed whenever necessary.

Default value = 1 (true)

SetUseSwapFile (obsolete)

Syntax:

```
LBOOL pdfSetUseSwapFile(
   const PPDF* IPDF,   // Instance pointer
   LBOOL SwapContents, // If true, content streams are paged out too
   UI32 SwapLimit)     // Memory limit in KB
```

This function is obsolete and should no longer be used. Beginning with DynaPDF 3.0, more efficient features are available to restrict the memory usage.

When importing large PDF files, the flag if2UseProxy should be set to drastically reduce the memory usage (see SetImportFlags2() for furhter information). When this flag is set, PDF files of arbitrary size can be imported with minimal memory usage. The only restriction is that the imported PDF file(s) cannot be deleted before CloseFile() or CloseFileEx() was called.

To enable the creation of large PDF files (up to 4 GB), it is possible to write finished pages into the output file with FlushPages(). This is much more efficient and reduces the memory usage drastically.

Beginning with DynaPDF 3.0, this function writes only images into the temp file. The parameter *SwapContents* is no longer considered.

Remarks:

The function tmpfile() creates the temp file usually in the system's root directory. If you want to use your own directory into which the temp file can be created then use the function SetUseSwapFileEx() instead.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetUseSwapFileEx (obsolete)

Syntax:

```
LBOOL pdfSetUseSwapFileEx(
   const PPDF* IPDF,      // Instance pointer
   LBOOL SwapContents,    // If true, content streams are paged out too
   UI32 SwapLimit,        // Memory limit in KB
   const char* SwapDir)   // Destination directory of the temp file
```

This function is obsolete and should no longer be used. Beginning with DynaPDF 3.0, more efficient features are available to restrict the memory usage. See SetUseSwapFile() for further information.

Beginning with DynaPDF 3.0, the function writes only images into the temp file. The parameter *SwapContents* is no longer considered.

The temp file will be created in the directory *SwapDir*. This directory must exist when the function is called.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetUseSystemFonts

Syntax:

```
LBOOL pdfSetUseSystemFonts(
   const PPDF* IPDF, // Instance pointer
   LBOOL Value)      // see description
```

The property specifies whether the system fonts should be automatically loaded. On Windows all fonts in the Windows/Fonts directory will be loaded as well as shared fonts listed in the Registry. On Mac OS X the following font paths will be added to the list of font search paths:

```
 /System/Library/Fonts
 /Library/Fonts
~/Library/Fonts
```

Additional font paths can be added with AddFontSearchPath().

```
Default value = 1 (true)
```

The fonts are loaded when SetFont() is called the first time. The property is ignored on Linux or UNIX.

Remarks:

Please note that the internal font list is not automatically updated when the system's font pool changes. Windows operating systems send a WM_FONTCHANGE message if the font pool was

modified. In this case, the application should call ClearHostFonts() before calling SetFont() the next time so that the list of available system fonts can be updated.

On Mac OS X there is no notification when the font database changes. Mac OS X applications need to use functions of the font manager to determine whether font database has been modified.

SetUseTransparency

Syntax:

```
LBOOL pdfSetUseTransparency(
    const PPDF* IPDF, // Instance pointer
    LBOOL Value)      // If false, images appears opaque
```

The property specifies whether images should be masked with the current transparent color to make this color transparent, see also SetTransparentColor(). This technique is known as color key masking.

The default transparent color is white.

1 bit images are inserted as an image mask if this property is true. Non-zero pixels of an image mask are rendered with the current fill color. See InsertImageEx() for further information.

Default value = 1 (true)

Remarks:

Color key masking is never applied on images with an alpha channel.

SetUseVisibleCoords

Syntax:

```
LBOOL pdfSetUseVisibleCoords(
    const PPDF* IPDF, // Instance pointer
    LBOOL Value)      // See description
```

The property specifies whether DynaPDF should consider the coordinate origin and the crop box of a page to calculate the position of an object. The crop box crops a page, but the paper format (media box) is left unchanged. The media box represents the coordinate system of the page. The bounding boxes are described in detail at SetBBox().

The coordinate origin of the media box or the crop box (if specified) must normally be considered when drawing contents on a page. SetUseVisibleCoords() simplifies page editing since no further considerations are required to place arbitrary contents on a page.

The functions GetPageWidth() and GetPageHeight() return the logical width and height if SetUseVisibleCoords() is true. However, only these two functions consider this property. Functions like GetBBox() or GetInBBox() return still the original values.

```
Default value = 0 (false)
```

SetViewerPreferences

Syntax:

```
LBOOL pdfSetViewerPreferences(
    const PPDF* IPDF,         // Instance pointer
    TViewerPreference Value, // Preference
    TViewPrefAddVal AddVal)  // Parameter of the preference if any

typedef SI32 TViewerPreference;
#define vpUseNone              0x00000000 // No preference is set
#define vpHideToolBar          0x00000001 // No parameter
#define vpHideMenuBar          0x00000002 // No parameter
#define vpHideWindowUI         0x00000004 // No parameter
#define vpFitWindow            0x00000008 // No parameter
#define vpCenterWindow         0x00000010 // No parameter
#define vpDisplayDocTitle      0x00000020 // (PDF 1.4) No parameter
#define vpNonFullScrPageMode   0x00000040 // Key, values see below
#define vpDirection            0x00000080 // (PDF 1.3)
#define vpViewArea             0x00000100 // (PDF 1.4)
#define vpViewClip             0x00000200 // (PDF 1.4)
#define vpPrintArea            0x00000400 // (PDF 1.4)
#define vpPrintClip            0x00000800 // (PDF 1.4)

typedef SI32 TViewPrefAddVal;
#define avNone                 0x00000000
#define avNonFullScrUseNone    0x00000001
#define avNonFullScrUseOutlines 0x00000002
#define avNonFullScrUseThumbs  0x00000004
#define avNonFullScrUseOC      0x00000400 // PDF 1.6
#define avDirectionL2R         0x00000008
#define avDirectionR2L         0x00000010
#define avViewPrintArtBox      0x00000020
#define avViewPrintBleedBox    0x00000040
#define avViewPrintCropBox     0x00000080
#define avViewPrintMediaBox    0x00000100
#define avViewPrintTrimBox     0x00000200
```

The function sets the viewer preferences which control certain features in PDF viewers, such as hiding the toolbar or menu bar. The parameters *Value* and *AddVal* are both bitmasks. Multiple values can be set by using a bitwise or operator. It is also possible to call the function multiple times to set each preference separately; the flags are combined with already existing one in this case. Already defined flags can be deleted with the flag vpUseNone. A few flags require an additional parameter that must be added to *AddVal*.

The flags are described in detail on the next page.

```
Default value = vpUseNone
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Preference	Description
vpUseNone	This is the default behaviour; the flag can also be used to delete all flags which were set beforehand.
vpHideToolBar	If set, the toolbar is hidden in Acrobat when the document is active.
vpHideMenuBar	If set, the menubar is hidden in Acrobat when the document is active.
vpHideWindowUI	If set, scrollbars and navigation controls are hidden when the document is active.
vpDisplayDocTitle	*(PDF 1.4)* If set, Acrobat displays the document's title in the window's title bar.
vpNonFullScrPageMode	*(PDF 1.4)* The parameter *AddVal* specifies how to display the document in non-fullscreen mode. This flag is meaningful only if the page mode was set to pmFullscreen (see SetPageMode()).
vpDirection	*(PDF 1.4)* The parameter *AddVal* defines the natural reading order of the document. This entry has no direct effect on the document's contents or page numbering, but can be used to determine the relative positioning of pages when displayed side by side or printed n-up. Default value: L2R.
vpViewArea	*(PDF 1.4)* The parameter *AddVal* specifies the name of the page boundary representing the area of a page to be displayed when viewing the document on the screen. The value is the key designating the relevant page boundary in the page object (see SetBBox() for further information). Default value: CropBox.
vpViewClip	*(PDF 1.4)* The parameter *AddVal* specifies the name of the page boundary to which the contents of a page are to be clipped when viewing the document on the screen. The value is the key designating the relevant page boundary in the page object (see SetBBox() for further information). Default value: CropBox.
vpPrintArea	*(PDF 1.4)* The parameter *AddVal* specifies the name of the page boundary representing the area of a page to be rendered when printing the document. The value is the key designating the relevant page boundary in the page object (see SetBBox() for further information). Default value: CropBox.

vpPrintClip	*(PDF 1.4)* The parameter *AddVal* specifies name of the page boundary to which the contents of a page are to be clipped when printing the document. The value is the key designating the relevant page boundary in the page object (see SetBBox() for further information). Default value: CropBox.

SetWMFDefExtent

Syntax:

```
LBOOL pdfSetWMFDefExtent(
   const PPDF* IPDF, // Instance pointer
   UI32 Width,       // Width in 0.01 millimetres
   UI32 Height)      // Height in 0.01 millimetres
```

The function sets the default size which is used to convert non-portable WMF files to EMF. See InsertMetafile() for further information.

```
Default value:
Width  = 0;
Height = 0;
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetWMFPixelPerInch

Syntax:

```
LBOOL pdfSetWMFPixelPerInch(
   const PPDF* IPDF, // Instance pointer
   UI32 Value)       // Pixels per inch of the y-axis see description
```

The function sets the default pixels per inch of the y-axis which are used to convert portable WMF files to EMF. DynaPDF uses a corrected value to get exact proportions. See InsertMetafile() for further information. The value must be in the range 2000 to 3000.

```
Default value = 2400
```

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetWordSpacing

Syntax:

```
double pdfGetWordSpacing(
   const PPDF* IPDF, // Instance pointer
   double Value)     // Word spacing
```

The function sets the word spacing.

```
Default value = 0
```

Value = 0	Word Space
Value = 80	Word Space

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SetXFAStream

Syntax:

```
LBOOL pdfSetXFAStream(
   const PPDF* IPDF,   // Instance pointer
   UI32 Index,         // Stream index
   const void* Buffer, // XFA buffer
   UI32 BufSize)       // Buffer size in bytes
```

The function replaces the contents of an XFA stream with new contents. *Index* must be a valid XFA resource index. XFA streams can be enumerated with GetXFAStream() / GetXFAStreamCount().

The function does not check whether the provided XFA buffer is valid. The buffer is written to the stream as is.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SkewCoords

Syntax:

```
LBOOL pdfSkewCoords(
    const PPDF* IPDF, // Instance pointer
    double alpha,     // Angle alpha in degrees
    double beta,      // Angle beta in degrees
    double OriginX,   // Origin of the x-axis
    double OriginY)   // Origin of the y-axis
```

The function skews the coordinate system and sets the coordinate origin to the point *OriginX*, *OriginY*. It is highly recommended to save the graphics state beforehand, otherwise it is very difficult or impossible to restore the coordinate system later.

After the coordinate system was changed by the function, bottom-up coordinates are active. It is not possible to use top-down coordinates with a skewed coordinate system.

Skewing is internally calculated as follows:

```
TCTM M; // Transformation matrix, the data type is declared in dynapdf.h
M.a = 1.0;
M.b = tan(alpha / 180 * PI);
M.c = tan(beta  / 180 * PI);
M.d = 1.0;
M.x = OriginX;
M.y = OriginY;
```

Because the tangent of the angles alpha and beta represent the skewed coordinate system, the angles must not be 90, 180, 270, or 360 degrees.

Remarks:

If the graphics state was not saved beforehand the function sets a warning but the transformation is applied.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0. Use an error callback function or check whether an error occurred with the function GetErrorMessage(). See Exception handling for further information.

Skewing:
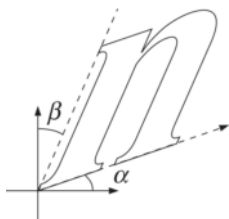
SortFieldsByIndex

Syntax:

```
LBOOL pdfSortFieldsByIndex(
    const PPDF* IPDF) // Instance pointer
```

The function sorts the interactive form fields of a page by comparing the internal indices which can be set for each field separately (see SetFieldIndex() for further information). The field's index represents the tab order of a field.

If the field indices were not changed beforehand, sorting makes no sense.

To sort the fields of a page, open the page for editing beforehand with EditPage().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SortFieldsByName

Syntax:

```
LBOOL pdfSortFieldsByName(
    const PPDF* IPDF) // Instance pointer
```

The function sorts the interactive form fields of a page in ascending order by field name. To sort the fields of a page, open the page for editing beforehand with EditPage(). The order in which fields appear in the page's field array specify the tab order of the fields.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

SquareAnnot

Syntax:

```
SI32 pdfSquareAnnot(
    const PPDF* IPDF,     // Instance pointer
    double PosX,          // y-coordinate of the annotation
    double PosY,          // x-coordinate of the annotation
    double Width,         // Width of the annotation
    double Height,        // Height of the annotation
    double LineWidth,     // Line width of the circle or ellipse
    UI32 FillColor,       // Fill color or NO_COLOR. See description
    UI32 StrokeColor,     // Stroke color or NO_COLOR. See description
    TPDFColorSpace CS,    // Color space of the fill and stroke colors
    const char* Author,   // Optional author
    const char* Subject,  // Optional subject
    const char* Comment)  // Optional comment
```

The function draws a square annotation on the current open page. If the parameters *Width* and *Height* are equal the function draws a square, a rectangle otherwise. If the annotation should be drawn without a border, set the parameter *LineWidth* zo zero or *StrokeColor* to the special constant NO_COLOR.

If the interior should be transparent set *FillColor* to the special constant NO_COLOR.

Although the line width can be set to any positive floating point value, Adobe's Acrobat or Reader restrict the line width to 0 through 12 units. The line width should be restricted in the same way to avoid issues in Adobe viewer products.

Remarks:

This function is implemented in an Ansi and Unicode compatible variant. Ansi strings are interpreted in the Windows code page 1252.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails, the return value is a negative error code.

StampAnnot

Syntax:

```
LBOOL pdfStampAnnot(
    const PPDF* IPDF,      // Instance pointer
    TRubberStamp SubType, // Stamp type
    double PosX,          // x-coordinate
    double PosY,          // y-coordinate
    double Width,         // Stamp width
    double Height,        // Stamp height
    const char* Author,   // Optional
    const char* Subject,  // Optional
    const char* Comment)  // Optional

typedef enum
{
    rsApproved,
    rsAsIs,
    rsConfidential,
    rsDepartmental,
    rsDraft,
    rsExperimental,
    rsExpired,
    rsFinal,
    rsForComment,
    rsForPublicRelease,
    rsNotApproved,
    rsNotForPublicRelease,
    rsSold,
    rsTopSecret,
    rsUserDefined // See below
}TRubberStamp;
```

The function creates a stamp annotation. The function is able to create appearance streams for the predefined stamps in English, German, and France. The default language is English.

In order to create a stamp in a different language than English, set the language identifier with SetLanguage() before creating the annotation. StampAnnot() supports only the notation as country key, e.g. "EN", "FR", "DE-de", and so on. Only the first two characters are used to identify the wished language. The comparison is non-case-sensitive.

The default stamps are drawn in the following default colors:

Stamp	Default color
rsApproved	PDF_RGB(62, 172, 72)
rsAsIs	PDF_RGB(240, 64, 36)
rsConfidential	PDF_RGB(240, 64, 36)
rsDepartmental	PDF_RGB(0, 112, 186)
rsDraft	PDF_RGB(240, 64, 36)
rsExperimental	PDF_RGB(0, 112, 186)
rsExpired	PDF_RGB(240, 64, 36)
rsFinal	PDF_RGB(240, 64, 36)
rsForComment	PDF_RGB(62, 172, 72)
rsForPublicRelease	PDF_RGB(62, 172, 72)
rsNotApproved	PDF_RGB(240, 64, 36)
rsNotForPublicRelease	PDF_RGB(240, 64, 36)
rsSold	PDF_RGB(0, 112, 186)
rsTopSecret	PDF_RGB(240, 64, 36)

The default color can be changed if necessary with SetAnnotColor(). The color type must be set to fcBorderColor.

A user defined stamp can be created as follows:

- Create the stamp with *SubType* set to rsUserDefined.
- Call CreateAnnotAP() with the handle of the stamp annotation to create a user defined appearance stream.
- Draw arbitrary contents and close the template finally with EndTemplate(), finished!

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

StrokePath

Syntax:

```
LBOOL pdfStrokePath(
   const PPDF* IPDF) // Instance pointer
```

The function strokes the current path without closing it. If no open path can be detected the function returns with a warning.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

TestGlyphs

Syntax:

```
SI32 pdfTestGlyphs(
    const PPDF* IPDF, // Instance pointer
    SI32 FontHandle,  // Font handle
    const char* Text) // Text that should be tested
```

The function checks whether all glyphs of the text are available in the specified font. The return value is the position of the first missing glyph, or -1 if all glyphs are available.

If the font uses a mixed 8/16 bit CJK code page that requires a conversion to Unicode (a code page that ends with "_Uni"), the return value corresponds to the converted Unicode string and not to the CJK input string.

TestGlyphsEx

Syntax:

```
SI32 pdfTestGlyphsEx(
    const PPDF* IPDF, // Instance pointer
    SI32 FontHandle,  // Font handle
    const char* Text, // Text that should be tested
    UI32 Len)         // Text length in characters
```

The function checks whether all glyphs of the text are available in the specified font like TestGlyphs() but it accepts a non-null-terminated string as input. The return value is the position of the first missing glyph, or -1 if all glyphs are available.

If the font uses a mixed 8/16 bit CJK code page that requires a conversion to Unicode (a code page that ends with "_Uni"), the return value corresponds to the converted Unicode string and not to the CJK input string.

TextAnnot

Syntax:

```
SI32 pdfTextAnnot(
    const PPDF* IPDF,    // Instance pointer
    double PosX,         // X-Coordinate of the annotation
    double PosY,         // Y-Coordinate of the annotation
    double Width,        // Width of the window in open state
    double Height,       // Height of the window in open state
    const char* Author,  // Author of the annotation or NULL
    const char* Text,    // The annotation's Text or NULL
    TAnnotIcon Icon,     // Annotation icon, see below
    LBOOL Open)          // Should the annotation appear open or closed?
typedef enum
{
    aiComment,
    aiHelp,
```

```
    aiInsert,
    aiKey,
    aiNewParagraph,
    aiNote,
    aiParagraph,
    aiUserDefined // Internal, not usable!
}TAnnotIcon;
```

The function creates a text annotation. The parameters *Width* and *Height* define the size of the annotation's window. The window is shown in the open state of the annotation.

PosX / *PosY* represents always the upper left corner of the annotation, independent of the used coordinate system. This is different in comparison to all other annotations because text annotations are never scaled or rotated with a page. The upper left corner of the scaled bounding box is the reference point on which the icon is rendered. Therefore, this is the only stable point at different zoom factors.

If the parameter *Open* is true, the annotation appears in its open state when opening the document.

The parameters *Author* and *Text* are optional, they can be NULL.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. The Ansi Version supports Ansi strings of the code page 1252 only. To create a text annotation in an arbitrary encoding convert the string to Unicode with the function ConvToIncode()and use the Unicode version to create the annotation.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

TranslateCoords

Syntax:

```
LBOOL pdfTranslateCoords(
   const PPDF* IPDF, // Instance pointer
   double OriginX,   // New coordinate origin of the x-axis
   double OriginY)   // New coordinate origin of the y-axis
```

The function translates the coordinate system to the new origin *OriginX*, *OriginY*. Save the graphics state beforehand and restore it when finish.

After the coordinate system was changed by the function, bottom-up coordinates are active. It is not possible to use top-down coordinates with a transformed coordinate system.

Remarks:

If the graphics state was not saved beforehand the function set a warning but the transformation will be applied.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0. Use an error callback function or check whether an error occurred with the function GetErrorMessage(). See Exception handling for further information.

TranslateRawCode (Font API)

Syntax:

```
UI32 fntTranslateRawCode(
   const void* IFont,  // Pointer of the active font
   const BYTE* Text,   // Raw text to be converted
   UI32 Len,           // Text length in bytes
   double ADDR Width,  // The width of the character
   UI16 OutText[32],   // Static Unicode output buffer
   SI32 ADDR OutLen,   // Number of characters copied to OutText
   LBOOL ADDR Decoded, // If false, Width, OutText, OutLen must be ignored
   float CharSpacing,  // Current character spacing
   float WordSpacing,  // Current word spacing
   float TextScale)    // Current text scaling
```

The function converts a source character to Unicode. The code length of a character depends on the font's encoding. PDF supports encodings with fixed and variable code lengths from one through four bytes per character. The return value is the number of bytes which were consumed to convert the character to Unicode; this value must be used to increment the text pointer and to decrement the remaining text length.

The resulting Unicode character or sequence is stored in the parameter *OutText,* as well as the corresponding output text length and the character width.

The parameter *Decoded* is set to true if the source character could be sucsessfully converted to Unicode. If *Decode* is false the output string and character width contain no meaningful values.

The function can only fail when processing strings of CID fonts, e.g. if the font depends on an external CMap that could not be loaded, or if the font uses a damaged CMap. The search path(s) to external CMaps should always be set before extracting text from PDF files with SetCMapDir().

Passing an invalid or undefined code sequence to the function does not result in an error. In this case the notdef character is added to the output string (this is usually 0 or 0xFFFD if the font contains a ToUnicode CMap).

Notice:

To improve processing speed the function does not check whether the parameters are valid.

Remarks:

This function was designed to convert raw strings on a per character basis to Unicode which were returned by ParseContent() or GetPageText(). The parameters *CharSpacing*, *WordSpacing*, and *TextScale* must be taken from the current graphics state or from the structure TPDFStack.

Return values:

The return value of this function is always greater or equal to one. Check the parameter *Decoded* to determine whether the function succeed.

TranslateString (obsolete)

Syntax:

```
SI32 fntTranslateString(
  struct TPDFStack* Stack, // Structure which holds the current text
stack
  UI16* OutText,           // Output Unicode buffer
  UI32 Size,               // Length of the Unicode buffer in characters
  UI32 Flags)              // No flags are defined at this time
```

The function converts a binary string to Unicode that was returned by GetPageText(). This function is marked as obsolete because the member Kerning of the structure TPDFStack contains already the converted Unicode string. See GetPageText() for further information.

The converted string is copied to the parameter *OutText*. The output buffer must be allocated by the caller. Note that the string buffer is a Unicode string in UTF-16 format. The parameter *Size* represents the length of the buffer in characters, it must be long enough to hold the entire string.

The output buffer should be at least *stack.TextLen* * 16 / 10 + 32 characters long. However, to avoid unnecessary memory allocation calls it is usually best to use a static conversion buffer with a length of about 2048 or 4096 characters. Longer strings can normally not occur in well formatted PDF files because a huge paper format would be required to display such a long string.

The parameter *Flags* was defined for future use. No flags are defined at this time, the parameter is ignored.

Remarks:

The output string buffer is automatically allocated in VB .Net and C#. The parameter *OutText* should be initialized with null or Nothing when using these programming languages.

This function should not be used to convert strings of CID fonts. To process strings of CID fonts use the function TranslateRawCode().

Notice:

To improve processing speed the function does not check whether the parameters are valid.

In C/C++ and Delphi the output string buffer is not null-terminated and the function does not use the exception handling of DynaPDF. In VB .Net and C# the function returns a native .Net string. No error message is returned on failure. However, the only possible error is out of memory.

Return values:

If the function succeeds the return value is the number of Unicode characters copied into the buffer. If the function fails the return value is zero.

TranslateString2 (Font API)

Syntax:

```
SI32 fntTranslateString2(
  const void* IFont, // Pointer of the active font
  const BYTE* Text,  // Raw text to be converted
  UI32 Len,          // Text length in bytes
  UI16* OutText,     // Output Unicode buffer
  UI32 Size,         // Length of the Unicode buffer in characters
  UI32 Flags);       // No flags are defined at this time
```

The function converts a binary string to Unicode that was returned by ParseContent() or GetPageText(). The converted string is copied to the parameter *OutText*. The output buffer must be allocated by the caller. Note that the string buffer is a Unicode string in UTF-16 format. The parameter *Size* represents the length of the buffer in characters, it must be long enough to hold the entire string.

The output buffer should be at least *Len* * 16 / 10 + 32 characters long. However, to avoid unnecessary memory allocation calls it is usually best to use a static conversion buffer with a length of about 2048 or 4096 characters. Longer strings can normally not occur in well formatted PDF files because a huge paper format would be required to display such a long string. The maximum allowed string length in PDF is 32767 characters.

Remarks:

The output string buffer is automatically allocated in VB .Net and C#. The parameter *OutText* should be initialized with null or Nothing when using these programming languages.

This function should not be used to convert strings of CID fonts. To process strings of CID fonts use the function TranslateRawCode().

Notice:

To improve processing speed the function does not check whether the parameters are valid.

In C/C++ and Delphi the output string buffer is not null-terminated and the function does not use the exception handling of DynaPDF. In VB .Net and C# the function returns a native .Net string. No error message is returned on failure. However, the only possible error is out of memory.

Return values:

If the function succeeds the return value is the number of Unicode characters copied into the buffer. If the function fails the return value is zero.

Triangle

Syntax:

```
LBOOL pdfTriangle(
    const PPDF* IPDF,      // Instance pointer
    double x1,             // X-Coordinate of the first point
    double y1,             // Y-Coordinate of the first point
    double x2,             // X-Coordinate of the second point
    double y2,             // y-Coordinate of the second point
    double x3,             // X-Coordinate of the third point
    double y3,             // Y-Coordinate of the third point
    TPathFillMode FillMode) // Fill mode
```

The function draws a triangle.

The draw direction can be changed with the function SetDrawDirection().

A triangle is a closed path that can be filled, stroked or both. It is also possible to draw a triangle invisible to apply the filling rules nonzero winding number or even-odd. The filling rules are described under ClipPath(). The parameter *FillMode* is ignored if the triangle is drawn inside a clipping path. The fill modes are described under ClosePath().

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

UnLockLayer

Syntax:

```
LBOOL pdfUnLockLayer(
    const PPDF* IPDF, // Instance pointer
    UI32 Layer)       // Handle of an OCG
```

The function removes a layer from the list of locked layers. See also LockLayer() for further information.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

UTF16ToUTF32

Syntax:

```
UI32* pdfUTF16ToUTF32(
    const PPDF* IPDF,   // Instance pointer
    const UI16* Source) // Null-terminated UTF-16 Unicode string
```

The function converts a UTF-16 Unicode string to UTF-32. UTF-32 is the default Unicode format on Linux and UNIX operating systems. The input string must be defined in the CPU's byte ordering (little-endian on a little-endian machine and big-endian on a big-endian machine), the output string is also returned in the CPU's byte ordering.

The return value is a pointer to the original conversion buffer; it must not be freed or changed. The function can be called multiple times without causing a memory leak. However, if the conversion buffer is no longer needed it can be released with the function FreeUniBuf(). The buffer is automatically released when CloseFile() or FreePDF() is called.

The function requires no open PDF file; it can be used at any time. Invalid character sequences are skipped by the algorithm, they do not break conversion.

Return values:

If the function succeeds the return value is a pointer to the converted UTF-32 Unicode string. If the function fails the return value is NULL. The only possible error during conversion is out of memory.

UTF16ToUTF32Ex

```
UI32* pdfUTF16ToUTF32Ex(
    const PPDF* IPDF,   // Instance pointer
    const UI16* Source, // UTF-16 Unicode string
    UI32 ADDR Len)      // String length in characters
```

The function converts a UTF-16 Unicode string to UTF-32. UTF-32 is the default Unicode format on Linux and UNIX operating systems. The input string must be defined in the CPU's byte ordering (little-endian on a little-endian machine and big-endian on a big-endian machine), the output string is also returned in the CPU's byte ordering.

The used conversion algorithm is binary save. The parameter *Len* holds the string length in characters before and after conversion. The new string length is normally not longer then original one, but it is possible that the new string length will be shorter because surrogates are expressed as two character sequence in UTF-16, but as one character in UTF-32.

The return value is a pointer to the original conversion buffer; it must not be freed or changed. The function can be called multiple times without causing a memory leak. However, if the conversion buffer is no longer needed it can be released with the function FreeUniBuf(). The buffer is automatically released when CloseFile() or FreePDF() is called.

The function requires no open PDF file; it can be used at any time. Invalid character sequences are skipped by the algorithm, they do not break conversion.

Return values:

If the function succeeds the return value is a pointer to the converted UTF-32 Unicode string. If the function fails the return value is NULL. The only possible error during conversion is out of memory.

UTF32ToUTF16

Syntax:

```
UI16* pdfUTF32ToUTF16(
    const PPDF* IPDF,   // Instance pointer
    const UI32* Source) // Null-terminated UTF-32 Unicode string
```

The function converts a UTF-32 Unicode string to UTF-16. UTF-32 is the default Unicode format on Linux and UNIX operating systems. This function uses the macro ToUTF16() which can be used to convert UTF-32 strings to UTF-16.

The macro is mostly used to convert literal strings in parameterized function calls which can contain more than one Unicode parameter. To enable the usage in functions with more than one string parameter, the function holds an array of **six** separate conversion buffers so that a second or third call of the macro does not override or free the buffer of a previous call.

Example:

```
// Linux uses UTF-32 as Unicode string format
Anyfunction(pdf, ToUTF16(pdf, L"First"), ToUTF16(pdf, L"Second"),
ToUTF16(pdf, L"Third!"));
```

The code above produces no access violation because each call of the macro ToUTF16() uses its own conversion buffer. DynaPDF contains no function that supports more than three Unicode string parameters so that the number of six separate conversion buffers is large enough for all functions which are currently defined in DynaPDF.

The number of conversion buffers is defined by the constant PDF_UTF16_BUF_COUNT in the header file pdf_main.h; it can be changed if necessary (you need the source code of DynaPDF and the library must be recompiled when changing the value).

The input string must be defined in the CPU's byte ordering (little-endian on a little-endian machine and big-endian on a big-endian machine), the output string is also returned in the CPU's byte ordering.

The return value is a pointer to the original conversion buffer; it must not be freed or changed. The function can be called multiple times without causing a memory leak. However, if the conversion buffers are no longer needed they can be released with FreeUniBuf() (this will free all four buffers). The buffers are automatically released when CloseFile() or FreePDF() is called.

The function requires no open PDF file; it can be used at any time. Invalid character sequences are skipped by the algorithm, they do not break conversion.

Return values:

If the function succeeds the return value is a pointer to the converted UTF-16 Unicode string. If the function fails the return value is NULL. The only possible error during conversion is out of memory.

UTF32ToUTF16Ex

Syntax:

```
UI16* pdfUTF32ToUTF16Ex(
   const PPDF* IPDF,   // Instance pointer
   const UI32* Source, // UTF-32 Unicode string
   UI32 ADDR Len)      // String length in characters
```

The function converts a UTF-32 Unicode string to UTF-16. UTF-32 is the default Unicode format on Linux and UNIX operating systems. The input string must be defined in the CPU's byte ordering (little-endian on a little-endian machine and big-endian on a big-endian machine), the output string is also returned in the CPU's byte ordering.

The used conversion algorithm is binary save. The parameter *Len* holds the string length in characters before and after conversion. The new string length can be longer then original one because surrogates are expressed as two character sequence in UTF-16, but as one character in UTF-32.

The return value is a pointer to the original conversion buffer; it must not be freed or changed. The function can be called multiple times without causing a memory leak. However, if the conversion buffer is no longer needed it can be released with the function FreeUniBuf(). The buffer is automatically released when CloseFile() or FreePDF() is called.

The function requires no open PDF file; it can be used at any time. Invalid character sequences are skipped by the algorithm, they do not break conversion.

Return values:

If the function succeeds the return value is a pointer to the converted UTF-32 Unicode string. If the function fails the return value is NULL. The only possible error during conversion is out of memory.

WatermarkAnnot

Syntax:

```
SI32 pdfWatermarkAnnot(
    const PPDF* IPDF, // Instance pointer
    double PosX,      // x-coordinate
    double PosY,      // y-coordinate
    double Width,     // Annotation width
    double Height)    // Annotation height
```

The function creates a Watermark annotation. Watermark annotations have no interactive elements like other annotation types. This property is useful in many cases because it is usually not possible to select or change the contents of a watermark annotation in viewer applications.

The annotation has no appearance after it was created. Call CreateAnnotAP() to create an appearance template for the annotation. After the appearance template was created you can draw text, images or vector graphics into it. The template must be closed with EndTemplate() when finished. It is also possible to import an external page or EMF contents into the template.

Watermark annotations can be shared on multiple pages. To place the annotation on other pages call AddAnnotToPage().

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

WebLink

Syntax:

```
SI32 pdfWebLink(
    const PPDF* IPDF, // Instance pointer
    double PosX,      // X-Coordinate of bounding rectangle
    double PosY,      // Y-Coordinate of bounding rectangle
    double Width,     // Width of bounding rectangle
    double Height,    // Height of bounding rectangle
    const char* URL)  // URL defined as null-terminated 7 bit ASCII string
```

The function inserts a web link onto the current open page. The parameter *URL* holds the URL defined as 7 bit ASCII string. A uniform resource locator (URL) is a string that resolves to a resource on the internet - typically a file that is the destination of a hypertext link, although it can also resolve to a query or other entity. A web link opens the standard browser to view the resource by default. The full version of Adobes Acrobat enables you also to embed a reverenced html resource in your document.

If the coordinate system is bottom-up the point *PosX, PosY* defines the lower left corner of the bounding rectangle. If the coordinate system is top-down it defines the upper left corner.

The border of the link annotation is drawn by using the current line width, stroke color and line dash pattern. If the link should appear without a border set the line width to zero beforehand.

When clicking on a link annotation the rectangle is highlighted, that is a simple visual effect. Several highlight modes are supported, see SetLinkHighlightMode() for further information.

Remarks:

This function is implemented in an Ansi and Unicode compatible version. Because the URL must be defined as 7 bit ASCII string, a Unicode URL must not contain characters outside of 7 bit ASCII.

Return values:

If the function succeeds the return value is the annotation handle, a value greater or equal zero. If the function fails the return value is a negative error code.

WriteAngleText

Syntax:

```
LBOOL pdfWriteAngleText(
   const PPDF* IPDF,  // Instance pointer
   const char* AText, // Text to be printed
   double Angle,      // Angle alpha in degrees
   double PosX,       // X-Coordinate of the text
   double PosY,       // Y-Coordinate of the text
   double Radius,     // Output radius or 0, see below
   double YOrigin)    // Origin of the y-axis, see below
```

The function draws text in a user defined angle around a radius. If the coordinate system is bottom-up the point *PosX, PosY* defines the lower-left corner of the text, otherwise the top-left corner. The parameter *Radius* moves the text in the rotated coordinate system on the x-axis so that the string will be printed on a radius. The parameter *YOrigin* moves the string also in the rotated coordinate system, but on the y-axis; the origin is measured in bottom-up coordinates independently of the current coordinate system. The y-origin is required when the text should be centered on the y-axis.

For instance, if the coordinate system is bottom-up, the point *PosX, PosY* defines the baseline of the text that is the center of the rotation. If the text should be centred on the y-axis the origin of the y-axis must be moved downward by the half cap height of the current font (see example below).

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

Example (C++):

```cpp
#include "dynapdf.h"
using namespace DynaPDF;
// First we declare an error callback function
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMessage, SI32 ErrType)
{
   printf("%s\n", ErrMessage);
   return 0;
}

int main(int argc, char* argv[])
{
   double alpha = 0.0;
   double y, pageCenter;
   SI32 i;
   void* pdf = pdfNewPDF();
   if (!pdf) return 2; // Out of memory?
   pdfSetOnErrorProc(pdf, NULL, PDFError);
   pdfSetPageCoords(pdf, pcTopDown);
   pdfCreateNewPDF(pdf, "c:/cppout.pdf");
   pdfAppend(pdf);
   pdfSetFont(pdf, "Arial", fsNone, 20.0, true, cp1252);
   pageCenter = pdfGetPageWidth(pdf) / 2.0;
   y = -(20.0 + pdfGetDescent(PDF)) / 2.0; // Font size + descent
   for (i = 0; i < 12; i++)
   {
      pdfSetFillColor(pdf, (0x00C08080 | ((SI32)alpha) << 6));
      pdfWriteAngleText(pdf,"Circular Text",alpha,pageCenter,300,24,y);
      alpha += 30.0;
   }
   pdfEndPage(pdf);
   pdfCloseFile(pdf);
}
```

Output:
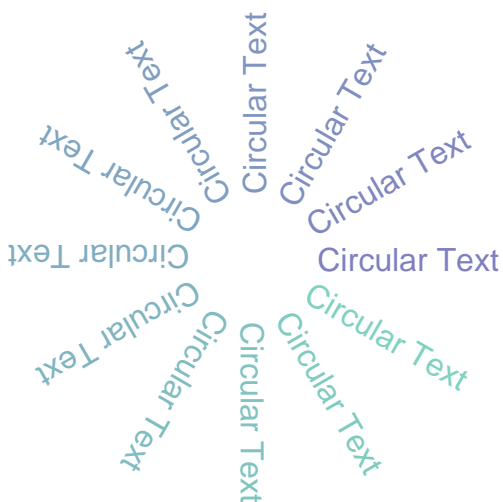
WriteFText

Syntax:

```
LBOOL pdfWriteFText(
    const PPDF* IPDF,  // Instance pointer
    TTextAlign Align,  // Base alignment
    const char* AText) // Null-terminated string to be printed

typedef enum
{
    taLeft      = 0,          // Left aligned text
    taCenter    = 1,          // Centered text
    taRight     = 2,          // Right aligned text
    taJustify   = 3,          // Justified text
    taPlainText = 0x10000000 // Process the text as is
}TTextAlign;
```

Instead of drawing text on a line by line basis it is also possible to output text into a rectangle by applying a formatting algorithm. The parameter *Align* defines the base alignment of the text to be drawn. The alignment can be changed inside the text by using an alignment tag.

The parameter *Text* holds a null-terminated string which can contain optional format tags to enable more complex formattings.

WriteFText() supports a callback function as well as several format tags which can be used to apply more complex formattings. The following sections describe the formatting options and the usage of the function in detail, please read these sections carefully.

Output rectangle

The output rectangle that is used by the function must be defined with the function SetTextRect(). If no output rectangle was set beforehand the function uses a default rectangle that is defined as follows:

- If the page contains a crop box the output rectangle becomes the dimension of the **crop box** with a border of 50 units (width - 100, height -100).

- If no crop box is present the output rectangle becomes the dimension of the **media box** with a border of 50 units (width - 100, height -100).

The bounding boxes are described in detail at SetBBox().

WirteFText() supports a callback function which is called when the output rectangle was filled with text and if more text is available for output (see SetOnPageBreakProc() for further information). If no callback was set, the text continues on the next page by using the same output rectangle.

To avoid page breaks set the height of the output rectangle to -1. Note that the Visual Basic and VB .Net interfaces use events instead of callback functions. The OnPageBreak event must be manually connected with the function ConnectPageBreakEvent().

Text formatting

WriteFText() can be used to output simple formatted text such as right aligned or justified text, but it is also possible to use certain format tags to create a list or other specific formattings. It is also possible to use a callback function to create multi-column text or other more complex formattings. First, we want to see how the text alignment can be changed inside a text block. WriteFText() supports 4 format tags which can be used to do this, we call these tags alignment tags.

Alignment tags

An alignment tag changes the text alignment in a text block. The base alignment that is used by the function was already defined by the parameter *Align*. The base alignment can also be changed inside the text by inserting an alignment tag. The following tags are available:

- `\le#` `// Left aligned text`
- `\re#` `// Right aligned text`
- `\ce#` `// Centred text`
- `\ju#` `// Justified text`

Like all format tags, an alignment tag begins with a backslash (character code 92) and ends with a numbersign (character code 35).

It is not possible to use multiple alignments in one text line. When an alignment tag is found the current line is closed and the cursor moves to the next line and continues with the new alignment.

Example:

```
char fText[] = "Left aligned text, this is left aligned text, this is "
"left aligned text...\\ju#\\FC[255]This is justified text, this is "
"justified text, this is justified text, this is justified text, this "
"is justified text, this is justified text, this is justified text, "
"this is justified text...\\re#\\FC[165536]This is right aligned text, "
"this is right aligned text, this is right aligned text..."
"\\le#\\FC[233512]Back to left aligned text, we go back to left "
"aligned text...";

pdfSetTextRect(pdf, 50, 50, 150, -1);
pdfWriteFText(pdf, taLeft, fText);
double height = pdfGetPageHeight(pdf) - pdfGetLastTextPosY(pdf) - 50;
pdfRectangle(pdf, 50, 50, 150, height, fmStroke);
...
```

Note that the above string is a literal string defined in C/C++. Because the backslash is an escape character in C/C++ it must be added twice. "\ \le#" is resolved to "\le#". The latter string is what the compiler passes to WriteFText(). The same string read from a file requires no escaping with additional backslashes!

Output:

Left aligned text, this is left
aligned text, this is left
aligned text...
This is justified text, this is
justified text, this is justified
text, this is justified text, this
is justified text, this is
justified text, this is justified
text, this is justified text...
This is right aligned text,
this is right aligned text, this
is right aligned text...
Back to left aligned text, we
go back to left aligned text...

As you can see above, the string is not delimited, each alignment tag follows just a string without a carriage return or line feed. We use the command tag `"\FC[]"` to change the text color so that we can better distinguish between the alignments. However, after changing the alignment the cursor moves always to the next line and the text is printed in the new alignment.

You can also see how the rectangle was filled with text. The function GetLastTextPosY() returns the baseline of the last text line in bottom-up coordinates . We use this value to calculate the height of the rectangle. However, the last baseline is often required to place other objects onto the page after the text was printed.

The x-coordinate can also be determined by the function GetLastTextPosX() so that it is quite easy to get the exact position of the end of the string.

The text alignment is active until it will be changed by an alignment tag. If the string contains no alignment tag, the entire text is printed in the same alignment. Now we want to see what kind of command tags are available. You have already seen a command tag (`\FC[]`), let's take a look on the other tags.

Command tags

A command tag is a tag that causes the execution of a function. This function can change a property, set a font, create a bookmark, or other things.

```
• \ul#                    // Underline
• \dl#                    // Double underline
• \st#                    // Strikeout
• \us#                    // Underline + strikeout
• \np#                    // New page
• \BM[char*]              // Add a bookmark
• \CS[float]              // Char spacing
• \EH#                    // End highlight text
• \EK#                    // End link
```

- `\EL#`   `// End list`
- `\FC[UI32]`   `// Font color`
- `\FS[float]`   `// Font size`
- `\FT[various]`   `// See description below`
- `\HA[int, int]`   `// Highlight annot, see description`
- `\LD[letter 't' or float]` `// Leading or line height, see description`
- `\LI[float, int, float]`   `// List, see description on the next page`
- `\LK[bool, bool, bool,`   `// Add a link, see description`
  `char*]`
- `\TL[int]`   `// Tab length`
- `\TR[float]`   `// Text rise`
- `\TS[float]`   `// Text scaling`

Two categories of command tags are available, those without a parameter and the others which require one or more parameters. The tags which require no parameter end with a numbersign (character code 35). Parameters are always encapsulated into brackets. The values inside the brackets must be defined without trailing spaces or other characters. Note that all characters before and after a tag will be printed.

Note that the backslash is an escape character in C/C++ literal strings. For better readability the second backslash is omitted. Note also that the backslash must be escaped in literal strings only.

Example:

`"This is a \ul#\FC[255]red\FC[0]\ul# string!"` = This is a <u>red</u> string!

Tag	Description
`\ul#`	Draw the following text as underlined text. The next `\ul#` tag deactivates underlined output: `"This is \ul#underlined\ul# text!"` = This is <u>underlined</u> text!
`\st#`	Draw the following text as strikeout text. The next `\st#` tag deactivates underlined output: `"This is \st#strikout\st# text!"` = This is ~~strikeout~~ text!
`\us#`	Draw the following text as underlined and strikeout text. The next `\us#` tag deactivates the tag: `"This is \us#underlined + strikeout\us# text!"` = This is ~~underlined + strikeout~~ text!
`\np#`	This tag creates a page break event. If the *OnPageBreak* callback function was set, the function is called without leaving the current page. The current page is still open, so that it is possible to make arbitrary changes inside the callback function such as changing the font, the output rectangle or closing the current page and opening another one so that the remaining text can be printed.

	If no callback function was set, the current page will be closed and the remaining text will be printed on the next page by using the same output rectangle. See also SetOnPageBreakProc(), ConnectPageBreakEvent().
\BM[`char*`]	WriteFText adds a bookmark with a destination of type dtXY_Zoom, and the x, y coordinate will be set to the position where this tag appears (the y-coordinate system is the baseline plus leading), the zoom factor is left unchanged. The parameter of the tag is the bookmark's title. Note that the title is **not** printed onto the page. To print the bookmarks title onto the page, add the string behind the tag again: `"\BM[My first bookmark...]My first bookmark..."` The bookmark title and the string that appears on the page can be different.
\CS[`float`]	The tag changes the current character spacing. The value must be in the range **-1000** to **1000** units. Note that negative values push the characters into the others. See also SetCharacterSpacing().
\EH#	End of highlight annotation. It requires a preceding \HA[] tag.
\EK#	This is an end of link tag. It requires a preceding \LK[] tag.
\EL#	This is an end of list marker. It requires a preceding create list tag, see above. The tag turns off the current list so that the following text can be printed in the usual way.
\FC[`UI32`]	The tag changes the current font color. The color value must be defined in the current color space. See also SetFillColor().
\FS[`float`]	The tag changes the current font size. The new font size must be greater zero.
\FT[`various`]	This tag changes the current font. The font can be changed by using a font handle or a font name. If a font handle is used, the font(s) must be loaded beforehand with SetFont(), SetFontEx(), or SetCIDFont(). Unused fonts are removed by DynaPDF, they will not be added to the file. The second way to change the font is by using a font name. If nothing else is specified, the specified name is interpreted as Postscript name. Example: "\FT[ArialMT]Some text..." Since DynaPDF 4.0.46.132 it is also possible to specify the font name as Family Name or Full Name. In addition, it is possible to define up to three alternate fonts. The search run continues until a font of the list can be successfully loaded. It is possible to combine all formats as needed. That means every font can be specified either as font handle, PostScript Name, Family Name, or Full Name. The tag will be output as plain text if it contains errors. Font loading errors are not passed to the error callback function (if set), if a font of the list can be loaded. A family name must be terminated with a colon and followed by the style flags as an

integer or hexadecimal number. A hexadecimal number must be preceeded with '0x' or '0X'. The very same syntax is used in C/C++, or C#, for example.

Example Family Name:

- `"\FT[Courier New:0x2BC00001]" -> Ok (fsBold | fsItalic)`
- `"\FT[Courier New : 0]"        -> Ok (fsNone)`
- `"\FT[Courier New : 419430400] -> Ok (fsRegular as Integer)`
- `"\FT[CourierNewPSMT, 0]Some text..." -> Ok`
- `"\FT[CourierNewPSMT: 0]Some text..." -> Wrong!`

The last example is wrong because a Postscript name was combined with style flags. Note that there is no way to distinguish between the different font names. The font cannot be found if a wrong name is provided.

Spaces between the font or style flags and the colon are ignored.

A Full Name must be terminated with a colon and followed by the keyword "full" (case-insensitive).

Example Full Name:

- `"\FT[Courier New Bold:full]Some text..."    -> Ok`
- `"\FT[ Courier New Bold : FULL]Some text..." -> Ok`

Example alternate fonts:

- `"\FT[Arial Bold:full, Helvetica: 734003200, Verdana Bold : full, Segoe UI :0x2BC00000, 4]" -> Ok`

- `"\FT[3, Arial Bold:full, Helvetica: 734003200]" -> Wrong!`
  `Since font handles must be valid, no alternate font can ever`
  `be loaded. A font handle should be the last entry in the`
  `list.`

The style flags of the fonts Helvetica and Segoe UI are identically but defined one time as an integer number and another time as hexadecimal number.

Examples:

- `Font handle:      "\FT[3]Some text..."`
- `Font handle:      "\FT[3:0x2BC00001]Some text..." (fsBold|fsItalic)`
- `Postscript name: "\FT[ArialMT]Some text..."`
- `Family name:      "\FT[Courier New:0]Some text..."`
- `Full name:        "\FT[Arial Bold:full]Some text..."`
- `Alternate font:  "\FT[Arial Italic:full, Helvetica:1, 5]"`

The second example combines a font handle with style flags. Although a font handle specifies the font to use, the style flags enable bold and italic emulation if needed.

The last example means: use the font with handle 5 if Arial Italic or Helvetica Italic cannot be found.

Notice:

Font names are case-sensitive, they must be specified exactly. Although the keyword "full"

or the prefix of hexadecimal numbers can be specified case insensitive, this is not the case for the font name.

The recommended function to enumerate available fonts is GetSysFontInfo(). It is also possible to use EnumHostFonts() or EnumHostFontsEx() but these functions do not return the Full Name of a font.

`\HA[int, int]`

This tag creates a Highlight, Underline, Strikeout, or Squiggly annotation. All these types are subtypes of a highlight annotation. The first parameter is the subtype that should be created. Valid values are:

```
4  = atHighlight
12 = atSquiggly
14 = atStrikeOut
16 = atUnderline
```

The second parameter is the highlight color. Both parameters are optional but note that the brackets are part of the syntax and therfore are required. If no parameter is provided then the subtype atHighlight in the color yellow will be created. The color can be defined as an integer or hexadecimal value. The latter one must be prefixed with "0x" or "0X" (without quotation marks).

The tag must be terminated with the tag \EH.

Example:

"This is a \HA[0xFF]Test...\EH#".

Result = This is a Test...

In the above example, the parameter subtype was omitted. It defaults to 4 = atHighlight.

`\LD['t' or float]`

This tag changes the current leading. The value can be a positive floating point number or the letter 't' (non-case sensitive and without quotes, e.g. "\LD[t]"). The letter 't' stands for typographic leading. The typographic leading or line height is available in newer TrueType and OpenType fonts which contain metrics for *TypoAscender*, *TypoDescender*, and *TypoLineGap*. Type1 fonts do not support these metrics.

If the metrics are not available, leading is set to *Ascent − Descent*. *Descent* is usually a negative number. If the calculated leading is smaller than the font size, the font size is used as leading. See also SetLeading().

`\LI[float, int,float]`

This tag creates a list. The tag has three parameters which must be delimited with a comma. The first parameter defines the left margin of the list character measured from the x-coordinate of the output rectangle. Positive values move the list character to right, negative values move it to left.

The second and third parameters are optional; they define the list character that should be used and the distance between the list character and the text. The list character must be a valid character index of the font *Wingdings-Regular*, or it must be an index into the current list font if set (see SetListFont()).

It is also possible to create a numbered list:

- If no value is defined, the character index **159** of the font *Wingdings-Regular* is used as list symbol. This is the same character as you see in this list.

- A value between **32** to **255** is treated as character index of the current list font. If no list font was set it defaults to the font *Wingdings-Regular* which is normally available on all Windows operating systems. Make sure that this font is available under Linux or UNIX.

- Values in the ranges **0** to **31** and **256** to **999** are not valid; the list will not be created.

- Since the Unicode index of the bullet character is above 1000 (decimal 8226 or hex 2022), the Unicode version of WriteFText() supports also an arbitrary Unicode character after the comma to specify the list symbol. The Unicode index of this character must be greater 33 and no number (48..57).

- Values greater or equal **1000** are used to create a numbered list. The function subtracts 1000 from the number and prints the resulting number as list symbol plus a dot using the current font:

  "\ \LI[10,1001]This is a numbered list!\ \EL#" =

    1. **This is a numbered list!**

A list supports the alignments left and justified text only. Right aligned or centred text produce unpredictable output. A list must be finished with the tag \EL#. The default distance between the list character and the text is 10 units.

`\LK[bool, bool,bool, char*,char*]`

This tag adds a link to a page. The parameters are as follows:

- IsNamedDest (false or 0, true or 1)
- IsExternalLink (false or 0, true or 1)
- OpenInNewWindow (false or 0, true or 1)
- URL or name of a named destination (see CreateNamedDest())
- PDF file in which the named destination is located

Link annotations support more properties which must be set before calling WriteFText():

- Border width   -> SetLineWidth()
- Border dash pattern   -> SetLineDashPattern()
- Border color   -> SetStrokeColor()
- Border color space   -> SetStrokeColorSpace()
- Highlight mode   -> SetLinkHighlightMode()

The first four parameters are required. The last parameter is required only if *IsNamedDest* and *IsExternalLink* are both true.

The path to an external PDF file should be a relative path. Although Unicode paths are supported since Acrobat 8, it is usually best to restrict the character set to plain english.

This tag must be terminated with a \EK# tag. At least one character must occur between \LK[...] and the corresponding \EK# tag. The link is created over these characters. If a line

or page break occurs before the terminating \EK# tag, the link annotation is created over multiple text lines. This results maybe also in multiple link annotations if a page break occurs.

Parameters of type bool can be defined as integer value 0 = false, 1 = true, or as string (non-case-sensitive). Unlike other command tags, whitespace between values are ignored and does not result in an error. All character codes below 33 are considered as whitespace.

The parameter *URL* is passed to WebLink() if *IsNamedDest* is false. If *IsNamedDest* is true, *URL* must be the name of a named destination. If *IsExternalLink* is false, *URL* is passed to PageLink3(). If true, the function creates a link annotation and a Go To Remote Action with CreateGoToRActionEx(). The Unicode version of WriteFText() calls CreateGoToRActionExU() to preserve the Unicode file name.

Examples:

- \LK[false, TRUE, false, https://www.dynaforms.com]Test\EK# // Ok
- \LK[0,1,0,https://www.dynaforms.com]Test\EK# // Same as above
- \LK[true, true, true, NameOfADestination, test.pdf]Test\EK# // Ok
- \LK[false, true, false, www.dyna forms.com   ]Test\EK# // Invalid URL!
- \LK[true, true, true, DestName, file name with spaces.pdf]Test\EK# // Ok

The file name in the last example is treated as "file name with spaces.pdf". Leading and trailing spaces are ignored.

\TL[int]	The tag changes the current tabulator length. The value must be in the range 1 to 255. See also SetTabLen().
\TR[float]	The tag changes the current text rise. Text rise can be used to create superscript or subscript text parts inside a text block. The value must be in the range **-1000** to **1000** units.
\TS[float]	The tag changes the current horizontal text scaling. The value must be greater zero. See also SetTextScaling().

Special characters

The following list defines special characters which can be used to insert new lines, tabulators and so on. Bidirectional characters require enabled complex text layout. See SetGStateFlags() for further information.

Character code (Hex)	Description
0x10 or 0x13 or 0x13 + 0x10	New line or end-of-line character code. The combination of a carriage return plus a line feed will be treaded as single end-of-line character.
0x09	A tabulator is emulated by inserting a variable count of space characters. The number of characters can be adjusted inside the text with the command tag \TL[ ] or outside with the property SetTabLen().

The function supports also specific Unicode characters which are used to control the bidirectional algorithm. The bidirectional formatting codes are also supported by all other text functions.

Character code (Hex)	Description
0x202A	*LRE* Left-to-Right Embedding (treat the following text as embedded left-to-right)
0x202B	*RLE* Right-to-Left Embedding (treat the following text as embedded right-to-left)
0x202C	*PDF* Pop Directional Format (restore the bidirectional state to what it was before the last LRE, RLE, LRO or RLO)
0x202D	*LRO* Left-to-Right Override (force the following characters to be treated as strong left-to-right characters)
0x202E	*RLO* Right-to-Left Override (force the following characters to be treated as strong right-to-left characters)
0x200E	*LRM* Left-to-Right Mark (left-to-right zero-width character)
0x200F	*RLM* Right-to-Left Mark (right-to-left zero-width character)
0x200C	Zero width non-joiner
0x200D	Zero width joiner

The bidirectional code pages 1255 and 1256 support the character codes 0x200E and 0x200F only (the mapped codes are 253 and 254).

Note that the Unicode character codes are supported by the Unicode version of the function only.

Escape Sequences

In situations where an alignement or command tag should be output as plain text it is possible to precede the tag with an additional backslash. This marks the string as plain text and the function does not interpret it as command or alignment tag. The first backslash is treated as escape character and will be removed.

Example:

```
\\       = \\
\\ce#    = \ce#
\\\le#   = \\le#
\\LI [...] = \\LI [...] // No valid tag -> always processed as text
\\LI[...]  = \LI[...]
```

The function compares the first four characters of command tags which accept parameters . If the sequence matches a command tag an additional backslash is required if the string should be output as text.

It is also possible to fully disable alignment and command tags with the flag *taPlainText* that can be passed to the parameter *Align*. Combine the flag with the wished alignment constant with a binary or operator, e.g. (TTextAlign)(taRight | taPlainText). This can be useful when it is known that the

incomming text cannot contain alignment or command tags. Line breaks and so on are still processed as usual but all kinds of alignment and command tags will be interpreted as plain text.

How to create multi-column text?

WriteFText() supports a callback function which can be used to create various formattings. The callback function is required to print text into multiple columns.

The Visual Basic and VB .Net interfaces use events instead of callback functions. However, the usage is identical but the event must be manually connected with the function ConnectPageBreakEvent() so that the event can be raised.

A good example to demonstrate the usage of the callback function is the output of multi-column text. In the following example we have a text file that contains the text that should be printed. The text that we use in this example is not of interest, it must only be large enough so that WriteFText() generates an OnPageBreak event. Note that it is also possible to create this event manually with the command tag \np#.

Example (C++):

First, we define a structure which contains all variables we need to calculate the output rectangle. We pass this structure to the SetOnPageBreakProc() function so that we don't need any global data.

```cpp
struct TOutRect
{
    void*  iPDF;      // Active PDF instance
    double PosX;      // X-coordinate of first output rectangle
    double PosY;      // Y-coordinate of first output rectangle
    double Width;     // Original width of the output rectangle
    double Height;    // Original height of the output rectangle
    double Distance;  // Space between columns
    SI32   Column;    // Current column
    SI32   ColCount;  // Number of columns
};

// This is our callback function
SI32 PDF_CALL OnPageBreakProc(const void* Data, double LastPosX, double LastPosY, SI32 PageBreak)
{
    TOutRect* r = (TOutRect*)Data; // get a pointer to our structure
    pdfSetPageCoords(r->iPDF, pcTopDown); // we use top-down coordinates
    if (!PageBreak && r->Column < r->ColCount -1)
    {
        ++r->Column;
        // Calculate the x-coordinate of the column
        double posX = r->PosX + r->Column * (r->Width + r->Distance);
        // change the output rectangle, do not close the page!
        pdfSetTextRect(r->iPDF, posX, r->PosY, r->Width, r->Height);
        switch(r->Column)
        {
            case 1:  return NEW_ALIGN_JUSTIFY;
            case 2:  return NEW_ALIGN_RIGHT;
            default: return 0; // do not change the alignment
        }
    }else
    {   // the page is full, close the current one and append a new page
        pdfEndPage(r->iPDF);
        pdfAppend(r->iPDF);
        pdfSetTextRect(r->iPDF, r->PosX, r->PosY, r->Width, r->Height);
        r->Column = 0;
        return NEW_ALIGN_LEFT;
```

```cpp
    }
}
// First we declare an error callback function.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char* ErrMessage, SI32 ErrType)
{
    printf("%s\n", ErrMessage); return 0;
}

int main(int argc, char* argv[])
{
    // The structure TOutRect holds all required variables to calculate
    // the output rectangle.
    TOutRect r;

    // The text is stored in a text file
    FILE* f = fopen("c:/sample.txt", "rb");
    if (f == NULL) return 2;

    char* fText;
    UI32 bufSize;
    fseek(f, 0, SEEK_END);
    bufSize = ftell(f);
    fseek(f, 0, SEEK_SET);

    // allocate one more character for the null-terminator
    fText = (char*)malloc(bufSize +1);
    fread(fText, 1, bufSize, f);
    fclose(f);
    fText[bufSize] = 0;   // Do NOT forget to add a null-terminator!!!

    r.iPDF = pdfNewPDF(); // Get a new PDF instance

    if (!r.iPDF)          // Out of memory?
    {
        free(fText);
        return 2;
    }
    pdfSetOnErrorProc(r.iPDF, NULL, PDFError);
    pdfSetDocInfo(r.iPDF, diCreator, "C++ test app");
    pdfSetDocInfo(r.iPDF, diSubject, "Multi-column text");
    pdfSetDocInfo(r.iPDF, diTitle, " Multi-column text ");

    // We use top-down coordinates
    pdfSetPageCoords(r.iPDF, pcTopDown);
    pdfCreateNewPDF(r.iPDF, "c:/cppout.pdf");
    pdfAppend(r.iPDF);
    pdfSetFont(r.iPDF, "Arial", fsNone, 8.0, true, cp1252);
    r.ColCount = 3;
    r.Column   = 0;
    r.Distance = 5.0;
    r.PosX     = 50.0;
    r.PosY     = 50.0;
    r.Height   = 150.0;
    r.Width    = (pdfGetPageWidth(r.iPDF) - 100.0 - (r.ColCount -1) * r.Distance) / r.ColCount;
    // The structure is passed to the callback function now
    pdfSetOnPageBreakProc(r.iPDF, &r, OnPageBreakProc);
    // Set the output rectangle first
    pdfSetTextRect(r.iPDF, r.PosX, r.PosY, r.Width, r.Height);
    pdfWriteFText(r.iPDF, taLeft, fText); // Now we can print the text

    free(fText);          // free the text buffer

    pdfEndPage(r.iPDF);   // Close the last page
    pdfCloseFile(r.iPDF); // Close the file
    pdfDeletePDF(r.iPDF); // Delete the PDF instance
}
```

The output of the previous example could look like this:

Bruder Lustig

Es war einmal ein großer Krieg, und als der Krieg zu Ende war, bekamen viele Soldaten ihren Abschied. Nun bekam der Bruder Lustig auch seinen Abschied und sonst nichts als ein kleines Laibchen Kommißbrot und vier Kreuzer an Geld; damit zog er fort. Der heilige Petrus aber hatte sich als ein armer Bettler an den Weg gesetzt, und wie der Bruder Lustig daherkam, bat er ihn um ein Almosen. Er antwortete: "Lieber Bettelmann, was soll ich dir geben? Ich bin Soldat gewesen und habe meinen Abschied bekommen, und habe sonst nichts als das kleine Kommißbrot und vier Kreuzer Geld, wenn das all ist, muß ich betteln, so gut wie du. Doch geben will ich dir was." Darauf teilte er den Laib in vier Teile und gab davon dem Apostel einen und auch einen Kreuzer. Der heilige Petrus bedankte sich und ging weiter, setzte sich aber zum dritten Mal in einer andern Gestalt als ein Bettler an den Weg und sprach den Bruder Lustig an.

As you can see, the first column is left aligned, the second is justified and the third is right aligned. By using manual page breaks it is possible to get full control over the formatting algorithm.

The parameter *PageBreak* of the callback is set to true if a page break tag was found, otherwise it is always false. Because of this, it is possible to distinguish between a manual page break and a page break that occurred due to a filled rectangle.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

WriteFTextEx

Syntax:

```
LBOOL pdfWriteFTextEx(
   const PPDF* IPDF,   // Instance pointer
   double PosX,        // X-Coordinate of output rectangle
   double PosY,        // Y-Coordinate of output rectangle
   double Width,       // Width of output rectangle
   double Height,      // Height of output rectangle or -1
   TTextAlign Align,   // Base alignment
   const char* AText)  // Null-terminated string to be printed
```

The function prints a formatted text exactly in the same way as WriteFText(). However, the function contains already the parameters to set the output rectangle. The function supports a callback function to enable the output of multi-column text in the same way as WriteFText(). However, if no callback function is set, the text continues on the next page by using the same output rectangle.

To avoid a page break set the parameter Height to **-1**. Manual page breaks which can be created with the command tag \np# are still executed. See WriteFText() for further information.

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

WriteText

Syntax:

```
LBOOL pdfWriteText(
   const PPDF* IPDF,   // Instance pointer
   double PosX,        // X-Coordinate of the text
   double PosY,        // Y-Coordinate of the text
   const char* AText)  // Null-terminated string to be printed
```

The function prints a text on the current open page, template, or pattern. The point *PosX*, *PosY* defines the baseline of the text if the coordinate system is bottom-up, otherwise the baseline lies at *PosY* + *FontSize* . The font origin can be changed with the function SetFontOrigin().

The function requires a font that must be set with the function SetFont() beforehand.

This function is implemented in an Ansi and Unicode compatible version. If non-null-terminated strings must be printed use WriteTextEx() instead.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

WriteTextEx

Syntax:

```
LBOOL pdfWriteTextEx(
   const PPDF* IPDF,  // Instance pointer
   double PosX,       // X-Coordinate of the text
   double PosY,       // Y-Coordinate of the text
   const char* AText, // Text to be printed
   UI32 Len)          // Text length in characters
```

The function prints a text on the current open page, template, or pattern. The point *PosX*, *PosY* defines the baseline of the text if the coordinate system is bottom-up, otherwise the top-left corner of the text's bounding box. The font origin can be changed with the function SetFontOrigin().

The function requires a font that must be set with the function SetFont() beforehand.

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

WriteTextMatrix

Syntax:

```
LBOOL pdfWriteTextMatrix(
   const PPDF* IPDF,  // Instance pointer
   struct TCTM* M,    // Transformation matrix
   const char* AText) // Null-terminated string to be printed
```

The function draws text on the current open page, template, or pattern by using a transformation matrix to calculate the position of the string. This function can be used in combination with GetPageText() to print a text on the same position as the original text that was found. The usage of the function is described at GetPageText().

Remarks:

This function is implemented in an Ansi and Unicode compatible version. If non-null-terminated strings must be printed use WriteTextMatrixEx() instead.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.

WriteTextMatrixEx

Syntax:

```
LBOOL pdfWriteTextMatrixEx(
   const PPDF* IPDF,  // Instance pointer
   struct TCTM* M,    // Transformation matrix
   const char* AText, // Text to be printed
   UI32 Len)          // Text length in characters
```

The function draws text on the current open page, template, or pattern by using a transformation matrix to calculate the position of the string. This function can be used in combination with GetPageText() to print a text on the same position as the original text that was found. The usage of the function is described at GetPageText().

Remarks:

This function is implemented in an Ansi and Unicode compatible version.

Return values:

If the function succeeds the return value is 1. If the function fails the return value is 0.