

DynaPDF 4.0

Reference & Manual

API Reference Version 4.0.27
March 21, 2019

Little CMS Copyright (c) 1998-2011 Marti Maria Saguer
LibPNG, PNG Reference Library, Copyright 1998-2004 Glenn Randers-Pehrson.
LibTIFF, TIFF Image Library, Copyright 1988-1997 Sam Leffler, Copyright 1991-1997 Silicon Graphics, Inc.
Zlib compression library, Copyright 1995-1998 Jean-Loup Gailly and Mark Adler.
The JBIG2 encoder in DynaPDF based on jbig2enc, Copyright 2006 Google Inc. Author: Adam Langley (agl@imperialviolet.org).
DynaPDF contains the RSA Security Inc. MD5 message digest algorithm as well as RC2, RC4, AES 128, and AES 256 encryption algorithms. DynaPDF contains also the Lempel-Ziv-Welch (LZW) compression algorithm, US Patent 4,558,302 Unisys Corporation. The patent expired worldwide in June 2004.

Legal Notices

Copyright: © 2003-2019 Jens Boschulte, DynaForms GmbH. All rights reserved.

DynaForms GmbH
Burbecker Street 24
D-58285 Gevelsberg, Germany
Trade Register HRB 9770, District Court Hagen
CEO Jens Boschulte
Phone: ++49 23 32-666 78 37
Fax: ++49 23 32-666 78 38

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).

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

Libjasper, JPEC2000 compression library, Copyright (c) 1999-2000 Image Power, Inc., Copyright (c) 1999-2000 The University of British Columbia, Copyright 2001-2003 Michael David Adams.

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

Table of Contents

Legal Notices	2	What is stored in a signature field?	57
Table of Contents	4	How to validate a signature?	57
Data types	13	PDF/A and PDF/X Compatibility	58
Var Parameters	14	PDF/X	58
Structures	14	PDF/A	60
Multi-byte Strings	14	Path Painting and Construction	62
Data types used by different programming languages	16	Nonzero Winding Number Rule	62
Exception handling	17	Even-Odd Rule	63
Exception handling in C, C++, C#, Delphi	17	Color Spaces	67
Exception handling in Visual Basic, Visual Basic .Net	18	Device Color Spaces	67
Special issues in Visual Basic and .Net	19	Device Independent Color Spaces	68
Customized Exception handling	20	Special Color Spaces	68
Custom Library Changes	21	Color Spaces and Images	71
Compiler Switches	21	Layers (Optional Content)	75
Main object types	21	PDF Transparency	77
General design requirements	22	Alpha Blending	77
Requirements to add your own code to DynaPDF	23	Transparency Groups / Soft Masks	79
Language bindings	25	Blend Modes	80
Differences between DynaPDF interfaces	25	Tables	84
Embarcadero C++ Builder	26	General properties	84
Microsoft Visual C++	28	Error Handling	84
Microsoft Visual Basic 6.0	30	Borders, Cell Spacing, Cell Padding	84
Visual Basic .Net	34	Background Objects	85
Visual C#	38	Foreground Objects	86
Embarcadero Delphi	41	Cell Alignment and Orientation	86
Compiling DynaPDF on Linux / UNIX	45	ColSpan, RowSpan	86
Compiling DynaPDF on Mac OS X	47	Page breaks	86
Interactive Forms	48	Table and cell properties	87
Field Appearance	48	Table color spaces	87
Field Properties	49	Table Creation	88
What is a Group Type?	50	Table Functions	89
How to change the tabulator order?	51	AddColumn	89
Field Names	52	AddRow	89
Actions	53	AddRows	90
Digital Signatures	55	ClearColumn	90
Supported Certificate Formats	55	ClearContent	90
External Signatures	55	ClearRow	91
How to export a Windows Certificate?	56	CreateTable	91
Importing signed PDF files	56	DeleteCol	92
How to sign a PDF file?	56	DeleteRow	92
How to create a signature field?	56	DeleteRows	92
How to modify the appearance of a signature field?	57	DeleteTable	92
		DrawTable	93
		GetFirstRow	93
		GetFlags	94
		GetNextHeight	94
		GetNextRow	95
		GetNumCols	95

GetNumRows	95	AddOutputIntent	131
GetPDFInstance	95	AddOutputIntentEx	132
GetTableHeight	95	AddPageLabel	133
GetTableWidth	96	AddRasImage (Rendering Engine)	134
HaveMore	96	AddRenderingIntent (obsolete)	135
SetBoxProperty	96	AddRenderingIntentEx (obsolete)	136
SetCellAction	97	AddValToChoiceField	136
SetCellImage	98	Append	137
SetCellImageEx	99	ApplyAppEvent	138
SetCellOrientation	100	ApplyPattern	138
SetCellTable	100	ApplyShading	142
SetCellTemplate	101	AssociateEmbFile	144
SetCellText	102	AttachFile	145
SetColor	103	AttachFileEx	145
SetColorEx	103	AttachImageBuffer (Rendering Engine)	146
SetCoiWidth	104	AutoTemplate	147
SetFlags	105	BeginClipPath (Obsolete)	148
SetFont	106	BeginContinueText	148
SetFontSelMode	106	BeginLayer	150
SetFontSize	107	BeginPageTemplate	151
SetGridWidth	107	BeginPattern	152
SetPDFInstance	107	BeginTemplate	154
SetRowHeight	108	BeginTransparencyGroup	156
SetTableWidth	108	Bezier_1_2_3	160
Bezier_1_3	161	Bezier_2_3	161
BuildFamilyNameAndStyle	162	BuildFamilyNameAndStyle	162
CalcPagePixelSize (Rendering Engine)	163	CalcPagePixelSize (Rendering Engine)	163
CalcWidthHeight	163	CaretAnnot	164
ChangeAnnot	165	ChangeAnnotName	165
ChangeAnnotPage	166	ChangeAnnotPage	166
ChangeBookmark	166	ChangeBookmark	166
ChangeFont	167	ChangeFont	167
ChangeFontSize	167	ChangeFontSize	167
ChangeFontStyle	167	ChangeFontStyle	167
ChangeFontStyleEx	168	ChangeFontStyleEx	168
ChangeJavaScript	168	ChangeJavaScript	168
ChangeJavaScriptAction	169	ChangeJavaScriptAction	169
ChangeJavaScriptName	169	ChangeJavaScriptName	169
ChangeLinkAnnot	170	ChangeLinkAnnot	170
ChangeSeparationColor	170	ChangeSeparationColor	170
CheckCollection	171	CheckCollection	171
CheckConformance	171	CheckConformance	171
CheckFieldNames	181	CheckFieldNames	181
CircleAnnot	182	CircleAnnot	182
ClearAutoTemplates	182	ClearAutoTemplates	182
ClearErrorLog	183	ClearErrorLog	183

Function Reference

Abort (Rendering Engine)	110
AddActionToObj	110
AddAnnotToPage	111
AddArticle	112
AddBookmark	112
AddBookmarkEx	115
AddBookmarkEx2	116
AddButtonImage	117
AddButtonImageEx	118
AddContinueText	118
AddDeviceNProcessColorants	118
AddDeviceNSeparations	119
AddFieldToFormAction	120
AddFieldToHideAction	121
AddFileComment	121
AddFontSearchPath	122
AddImage	123
AddLinkList	124
AddJavaScript	125
AddLayerToDispTree	126
AddMaskImage	129
AddObjectToLayer	129
AddOCCToAppEvent	130

ClearHostFonts	183	CreateImportDataAction	249
ClipPath	183	CreateIndexedColorSpace	249
CloseAndSignFileEx	184	CreateJSAction	250
CloseAndSignFileEx	186	CreateLaunchAction	251
CloseAndSignFileExt	187	CreateLaunchActionEx	252
CloseFile	190	CreateListBox	252
CloseFileEx	190	CreateNamedAction	253
CloseImage	195	CreateNamedDest	254
CloseImportFile	195	CreateNewPDF	255
ClosePath	196	CreateOC	258
CloseTag	197	CreateOCMD	260
ComputeBBox	197	CreateRadialShading	261
ConnectPageBreakEvent	198	CreateRadioButton	262
ConvColor	198	CreateRasterizer (Rendering Engine)	263
ConvertColors	201	CreateRasterizerEx (Rendering Engine)	264
ConvertEMFSpool	202	CreateRoaAction	265
ConvToUnicode	204	CreateSeparationCS	265
CopyChoiceValues	206	CreateSetOCStateAction	267
Create3DAnnot	207	CreateSigField	271
Create3DColorBackground	208	CreateSigFieldAP	273
Create3DGoToViewAction	208	CreateSigMask	274
Create3DProjection	209	CreateStdPattern	276
Create3DView	210	CreateStructureTree	279
CreateAnnotAP	212	CreateSubitAction	280
CreateArticleThread	212	CreateTextField	284
CreateAsiaShading	213	CreateURLAction	286
CreateBarcodeField	214	DecryptPDF	286
CreateButton	218	DeleteAcroForm	287
CreateCheckBox	219	DeleteActionFromObj	288
CreateCIEColorSpace	221	DeleteActionFromObjEx	288
CreateColItemDate	223	DeleteAnnotation	290
CreateColItemNumber	223	DeleteAnnotationFromPage	290
CreateColItemString	224	DeleteAppEvents	291
CreateCollection	224	DeleteBookmark	291
CreateCollectionField	226	DeleteEmbeddedFile	291
CreateColor	228	DeleteField	292
CreateDeviceNColorSpace	229	DeleteFieldEx	292
CreateExtGState	237	DeleteJavaScript	293
CreateGoToAction	240	DeleteOCFromAppEvent	293
CreateGoToActionEx	242	DeleteOutputIntent	293
CreateGoToActionEx2	242	DeletePage	294
CreateGoToActionEx3	242	DeletePageLabels	294
CreateGoToActionEx4	243	DeletePDF	294
CreateGoToActionEx5	244	DeleteRasterizer (Rendering Engine)	294
CreateGroupField	244	DeleteSeparationInfo	294
CreateHideAction	246	DeleteTemplate	295
CreateICCBasedColorSpace	246	DeleteTemplateEx	295
CreateImage	248	DeleteXFAPForm	296

DrawArc	296	GetAnnotBBox	328
DrawArcEx	298	GetAnnotCount	329
DrawChord	298	GetAnnotEx	329
DrawCircle	299	GetAnnotFlags	332
DrawFile	300	GetAnnotLink	333
EditPage	303	GetAnnotType	334
EditTemplate	303	GetAscend	335
EditTemplate2	304	GetBarcodeDict	335
Ellipse	304	GetBBox	336
EncryptPDF	305	GetBidiMode	337
EndContinueText (obsolete)	306	GetBidiModeEx	338
EndLayer	306	GetBookmark	338
EndPage	306	GetBookmarkCount	339
EndPattern	306	GetBorderStyle	339
EndTemplate	306	GetBuffer	339
EnumDocFonts	307	GetCapHeight	340
EnumHostFonts	309	GetCharacterSpacing	340
EnumHostFontsEx	310	GetCheckBoxChar	341
ExchangeBookmarks	311	GetCheckBoxCharEx	342
ExchangePages	312	GetCheckBoxDerState	342
ExtractText	312	GetCMap	342
FileAttachAnnot	313	GetCMapCount	343
FileAttachAnnotEx	314	GetColorSpace	344
FileLink	314	GetColorSpaceCount	344
FindBookmark	316	GetColorSpaceObj	344
FindEmbeddedFile	316	GetColorSpaceObjEx	345
FindField	317	GetCompressionFilter	345
FindLinkAnnot	317	GetCompressionLevel	346
FindNextBookmark	317	GetContent	346
FinishSignature	318	GetDefBitsPerPixel	347
FlattenAnnots	318	GetDescent	347
FlattenForm	319	GetDeviceNAttributes	347
FlushPageContent	320	GetDocInfo	348
FlushPages	320	GetDocInfoCount	349
FreeImageBuffer	321	GetDocInfoEx	349
FreeImageObj	321	GetDocUsesTransparency	349
FreeImageObjEx	321	GetDrawDirection	350
FreePDF	322	GetDynaPDFVersion	350
FreeTextAnnot	322	GetDynaPDFVersionInt	350
FreeUniBuf	323	GetEmbeddedFile	351
Get3DAnnotStream	324	GetEmbeddedFileCount	352
GetActionCount	324	GetEmbeddedFileNode	352
GetActionHandle	325	GetEFTSpec	353
GetActionType	325	GetEMFPatternDistance	353
GetActiveFont	326	GetErrMsg	354
GetAlignBy	327	GetErrMsgCount	355
GetAnnot (obsolete)	327	GetErrorMessage	355
GetAnnotBBox	328	GetErrMsgMode	355
GetAnnotCount	329	GetField (obsolete)	356
GetAnnotEx	329		
GetAnnotFlags	332		
GetAnnotLink	333		
GetAnnotType	334		
GetAscend	335		
GetBarcodeDict	335		
GetBBox	336		
GetBidiMode	337		
GetBidiModeEx	338		
GetBookmark	338		
GetBookmarkCount	339		
GetBorderStyle	339		
GetBuffer	339		
GetCapHeight	340		
GetCharacterSpacing	340		
GetCheckBoxChar	341		
GetCheckBoxCharEx	342		
GetCheckBoxDerState	342		
GetCMap	342		
GetCMapCount	343		
GetColorSpace	344		
GetColorSpaceCount	344		
GetColorSpaceObj	344		
GetColorSpaceObjEx	345		
GetCompressionFilter	345		
GetCompressionLevel	346		
GetContent	346		
GetDefBitsPerPixel	347		
GetDescent	347		
GetDeviceNAttributes	347		
GetDocInfo	348		
GetDocInfoCount	349		
GetDocInfoEx	349		
GetDocUsesTransparency	349		
GetDrawDirection	350		
GetDynaPDFVersion	350		
GetDynaPDFVersionInt	350		
GetEmbeddedFile	351		
GetEmbeddedFileCount	352		
GetEmbeddedFileNode	352		
GetEFTSpec	353		
GetEMFPatternDistance	353		
GetErrMsg	354		
GetErrMsgCount	355		
GetErrorMessage	355		
GetErrMsgMode	355		
GetField (obsolete)	356		

GetFieldBackColor	357	GetImageObjEx	391
GetFieldBorderColor	357	GetImageWidth	392
GetFieldBorderStyle	358	GetImportDataAction	392
GetFieldBorderWidth	358	GetImportFlags	392
GetFieldChoiceValue	358	GetImportFlags2	393
GetFieldColor	359	GetListBox	393
GetFieldCount	360	GetInDocInfo	394
GetFieldEx	360	GetInDocInfoCount	394
GetFieldEx2	363	GetInDocInfoEx	394
GetFieldExpValCount	363	GetInEncryptionFlags	395
GetFieldExpValue	364	GetInFieldCount	395
GetFieldExpValueEx	365	GetInFlagsCollection	396
GetFieldFlags	367	GetInIsEncrypted	396
GetFieldGroupType	371	GetInList	396
GetFieldHighlightMode	372	GetInMetadata	397
GetFieldIndex	372	GetInNameDest (obsolete)	398
GetFieldMapName	373	GetInNameDestCount (obsolete)	398
GetFieldName	373	GetInSigned	398
GetFieldOrientation	374	GetInIsTrapped	399
GetFieldTextAlign	375	GetInIsXFAPForm	399
GetFieldTextColor	375	GetInOrientation	399
GetFieldToolTip	375	GetInPageCount	400
GetFieldType	376	GetInPDFVersion	400
GetFillColor	376	GetInPrintSettings	400
GetFont (obsolete)	377	GetInRepairMode	401
GetFontCount	379	GetInPkgPch	401
GetFontEx (obsolete)	379	GetIsTaggingEnabled	402
GetFontInfo	379	GetItalicAngle	402
GetFontInfoEx	382	GetJavaScript	402
GetFontOrigin	382	GetJavaScriptAction (obsolete)	403
GetFontSearchOrder	383	GetJavaScriptAction2 (obsolete)	404
GetFontSelMode	383	GetJavaScriptActionEx	405
GetFontWeight	383	GetJavaScriptCount	405
GetFTTextHeight	383	GetJavaScriptEx	406
GetFTTextHeightEx	384	GetJavaScriptName	406
GetGlyphIndex	385	GetJavaScriptPos	407
GetGlyphOutline	385	GetLanguage	407
GetGoToAction	388	GetLastTextPosX, GetLastTextPosY	407
GetGoToActionEx	388	GetLaunchAction	409
GetGoToActionEx2	389	GetLayerConfig	410
GetGoToActionEx3	389	GetLayerConfigCount	410
GetGoToActionEx4	389	GetLeading	410
GetImageBuffer	389	GetLineStyle	411
GetImageCount	390	GetLineJoinStyle	411
GetImageCountEx	390	GetLineLengthMode	412
GetImageHeight	390	GetLinkHighLightMode	412
GetImageObj	390	GetLogMetafileSize	412
GetImageObjCount	391	GetLogMetafileSizeEx	415

GetMatrix	415
GetMaxFieldLen	416
GetMeasureObj	416
GetMetaConvFlags	417
GetMetadata	418
GetMissingGlyphs	419
GetMiterLimit	420
GetMovieAction	420
GetNamedAction	421
GetNamedDest	422
GetNamedDestCount	422
GetNeedAppearance	423
GetNumberFormatObj	423
GetObjActionCount (obsolete)	424
GetObjActions	424
GetObjEvent	425
GetOCC	426
GetOCCContUsage	427
GetOCCCount	428
GetOCCUsageUserName	428
GetOCHandle	428
GetOCUINode	429
GetOpacity	431
GetOrientation	431
GetOutputIntent	431
GetOutputIntentCount	432
GetPageAnnot (obsolete)	432
GetPageAnnotEx	433
GetPageAnnotCount	433
GetPageBBox (Rendering Engine)	433
GetPageCoords	434
GetPageCount	434
GetPageField (obsolete)	434
GetPageFieldCount	435
GetPageFieldEx	435
GetPageHeight	436
GetPageLabel	437
GetPageLabelCount	438
GetPageLayout	438
GetPageMode	438
GetPageNum	438
GetPageObject (Rendering Engine)	439
GetPageOrientation (Rendering Engine)	439
GetPageText	439
GetPageWidth	450
GetPDFVersion	451
GetPrintSettings	451
GetPDataArray	452

GetPDataObj	452
GetRelFileNode	452
GetResetAction	453
GetResolution	453
GetSaveNewImageFormat	454
GetSeparationInfo	454
GetSigDict	454
GetSpaceWidth	455
GetStrokeColor	455
GetSubmitAction	456
GetSysFontInfo	456
GetTabLen	458
GetTempCount	458
GetTempHandle	458
GetTempHeight	458
GetTempWidth	459
GetTextDrawMode	459
GetTextFieldValue	459
GetTextRect	460
GetTextRise	460
GetTextScaling	460
GetTextWidth	461
GetTextWidth (Font API)	462
GetTextWidthEx	462
GetTransparentColor	463
GetTrapped	463
GetURLAction	463
GetUseExactPvd	464
GetUseGlobalImpFiles	464
GetUserRights	465
GetUserInit	465
GetUseSystemFonts	466
GetUseSystemFonts	466
GetUsesTransparency	467
GetUseTransparency	468
GetUseVisibleCoords	468
GetViewerPreferences	468
GetViewPort	470
GetViewPortCount	470
GetWMFDefExtent	471
GetWMFFixedPerInch	471
GetWordSpacing	471
GetXFAStream	472
GetXFAStreamCount	472
HaveOpenDoc	473
HaveOpenPage	473
HighlightAnnot	473
ImportBookmarks	474

ImportCatalogObjects	474
ImportDocInfo	474
ImportEncryptionSettings	475
ImportOCProperties	475
ImportPage	476
ImportPageEx	478
ImportPDFFile	479
InitColorManagement	481
InitColorManagementEx	482
InitExGState	483
InitOCCContUsage	483
InitStack	484
InkAnnot	484
InsertBMPFromBuffer (obsolete)	485
InsertBMPFromHandle	485
InsertBookmark	486
InsertBookmarkEx	486
InsertImage (obsolete)	487
InsertImageEx	488
InsertImageFromBuffer	496
InsertMetallic	497
InsertMetallicEx	502
InsertMetallicExt	503
InsertMetallicExtEx	503
InsertMetallicFromHandle	504
InsertMetallicFromHandleEx	504
InsertRawImage	505
InsertRawImageEx	507
IsBidText	509
IsColorPage	510
IsEmptyPage	510
IsImagePage	510
LineAnnot	511
LineTo	512
LoadCMap	512
LoadFont	515
LoadImage	516
LoadFFDFData	518
LoadFFDFDataEx	518
LoadLayerConfig	518
LockLayer	519
MaxPage	519
MoveTo	520
MultiplyMatrix	520
NewPDF	521
OpenImportBuffer	521
OpenImportFile	523
OpenOutputFile	526

OpenOutputFileEncrypted	527
OpenTag	528
Optimize	531
PageLink	535
PageLink2	535
PageLink3	536
PageLinkEx	537
ParseContent	539
PlaceImage	562
PlaceFieldValidateIcon	563
PlaceTemplate	563
PlaceTemplateEx	564
PlaceTempByMatrix	567
PolygonAnnot	568
PolyLineAnnot	568
PrintPage	570
PrintPDFFile	570
ReadImageFormat (obsolete)	573
ReadImageFormat2	574
ReadImageFormatEx	575
ReadImageFormatFromBuffer	575
ReadImageResolution	576
ReadImageResolutionEx	576
Rectangle	577
Redraw (Rendering Engine)	577
RefObjTypePDF	578
RenameSpotColor	579
RenderAnnotOfField	579
RenderPage (Rendering Engine)	582
RenderPageEx (Rendering Engine)	594
RenderPageToImage (Rendering Engine)	595
RenderPDFFile (obsolete)	598
RenderPDFFileEx	598
ReplaceFont	600
ReOpenImportFile	600
ReplaceFontEx	601
ReplaceOCProfile	601
ReplaceCCProfileEx	602
ReplaceImage	602
ReplaceImageEx	603
ReplacePageText	604
ReplacePageTextEx	604
ResetEncryptionSettings	605
ResetLineDashPattern	605
ResizeBitmap (Rendering Engine)	606
RestoreGraphicState	606
RotateCoords	607
RoundRect	608

RoundRectEx	609
SaveGraphicState	610
ScaleCoords	611
SetTest	612
Set3DAnnotProps	612
Set3DAnnotScript	614
SetAllocBy	614
SetAnnotBorderEffect	615
SetAnnotBorderStyle	615
SetAnnotBorderWidth	616
SetAnnotColor	617
SetAnnotFlags	618
SetAnnotFlagsEx	619
SetAnnotHighLightMode	620
SetAnnotIcon	620
SetAnnotLineEndStyle	621
SetAnnotLineDashPattern	621
SetAnnotMigrationState	622
SetAnnotOpacity	623
SetAnnotOpenState	623
SetAnnotOfFieldDate	624
SetAnnotQuadPoints	624
SetAnnotString	625
SetAnnotSubject	625
SetBBox	626
SetBldMode	628
SetBookmarkDest	629
SetBookmarkStyle	631
SetBorderStyle	632
SetCharacterSpacing	632
SetCheckBoxChar	633
SetCheckBoxDefState	634
SetCheckBoxState	634
SetCIDFont	634
SetCMapDir	637
SetColDefFile	638
SetColorMask	639
SetColors	639
SetColorSpace	640
SetCompressionFilter	640
SetCompressionLevel	641
SetContent	641
SetDateTimeFormat	642
SetDeBtsPerPixel	643
SetDocInfo	643
SetDocInEx	644
SetDrawDirection	645

SetEMFFrameDPI	645
SetEMFPatternDistance	646
SetErrorMode	646
SetExtColorSpace	647
SetExtFillColorSpace	647
SetExtState	647
SetExtStrokeColorSpace	648
SetFieldBackColor	648
SetFieldBBox	648
SetFieldBorderColor	649
SetFieldBorderStyle	649
SetFieldBorderWidth	650
SetFieldColor	650
SetFieldExpValue	651
SetFieldExpValueEx	652
SetFieldFlags	653
SetFieldFont	656
SetFieldFontEx	657
SetFieldFontSize	658
SetFieldHighLightMode	658
SetFieldIndex	659
SetFieldMapName	661
SetFieldName	661
SetFieldOrientation	662
SetFieldTextAlign	662
SetFieldTextColor	663
SetFieldToolTip	663
SetFillColor	663
SetFillColorEx	664
SetFillColorF	664
SetFillColorSpace	665
SetFillPrecision	665
SetFont	666
SetFontEx	676
SetFontOrigin	677
SetFontSearchOrder	677
SetFontSearchOrderEx	678
SetFontSetMode	679
SetFontWeight	679
SetGStateFlags	680
SetIconColor	682
SetImportFlags	682
SetImportFlags2	686
SetItalicAngle	688
SetJPEGQuality	688
SetLanguage	689
SetLoading	690
SetLicenseKey	690

SetLineAnnotParams	691
SetLineCapStyle	692
SetLineDashPattern	693
SetLineDashPatternEx	694
SetLineJoinStyle	694
SetLineWidth	695
SetLinkHighlightMode	696
SetListFont	696
SetMatrix	697
SetMaxErrLogMsgCount	697
SetMaxFieldLen	698
SetMetaConvFlags	698
SetMetadata	702
SetMinLineWidth2 (Rendering Engine)	703
SetMiterLimit	703
SetNeedAppearance	704
SetNumberFormat	704
SetOCCContUsage	705
SetOCCState	707
SetOnErrorProc	707
SetOnPageBreakProc	708
SetOpacity	709
SetOrientation	709
SetOrientationEx	710
SetPageBBox	711
SetPageCoords	711
SetPageFormat	712
SetPageHeight	713
SetPageLayout	713
SetPageMode	714
SetPageWidth	714
SetPDFVersion	715
SetPrintSettings	716
SetProgressProc	717
SetResolution	718
SetSaveNewImageFormat	718
SetSeparationInfoEx	718
SetStrokeColor	719
SetStrokeColorEx	719
SetStrokeColorF	720
SetStrokeColorSpace	720
SetTabLen	721
SetTextDrawMode	722
SetTextFieldValue	724
SetTextFieldValueEx	725
SetTextRect	725
SetTextRise	726
SetTextScaling	726

SetTransparentColor	727
SetTrapped	727
SetUseExactPvd	727
SetUseGlobalImpFiles	728
SetUseImageInterpolation	729
SetUseSystemFonts	732
SetUseUnit	730
SetUseStdFonts	730
SetUseSwapFile (obsolete)	731
SetUseSwapFileEx (obsolete)	732
SetUseSystemFonts	732
SetUseTransparency	733
SetUseVisibleCoords	733
SetViewerPreferences	734
SetWMFDefExtent	736
SetWMFFixedPerInch	737
SetWordSpacing	737
SkewCoords	738
SortFieldsByIndex	739
SortFieldsByName	739
SquareAnnot	739
StampAnnot	740
StrokePath	741
TestGlyphs	742
TestGlyphsEx	742
TestAnnot	742
TranslateCoords	743
TranslateRawCode (Font API)	744
TranslateString (obsolete)	745
TranslateString2 (Font API)	746
Triangle	747
UnLockLayer	747
UTF16ToUTF32	747
UTF16ToUTF32Ex	748
UTF32ToUTF16	749
UTF32ToUTF16Ex	750
WatermarkAnnot	750
WebLink	751
WriteAngleText	752
WriteText	754
WriteTextEx	765
WriteText2	765
WriteTextEx	766
WriteTextMatrix	766
WriteTextMatrixEx	767

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):

```
typedef unsigned char BYTE;
typedef signed short SI16;
typedef unsigned short UI16;
typedef double FLOAT; // 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
#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

```
#ifndef __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
  #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 `dynampdf.dll` (Cannot find `dynampdf.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 `dynampdf.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!

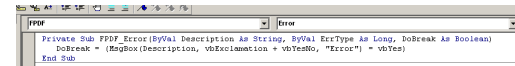
Exception handling in Visual Basic, Visual Basic .Net

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_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_JPEK 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_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. 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 `WriteText()` (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 width
#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 assigns 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_tmpl.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
<code>_WINDOWS</code>	<code>#ifdef _WINDOWS</code>	Set on Windows.
<code>MAC_OS_X</code>	<code>#ifdef MAC_OS_X</code>	Set on Mac OS X.
<code>DRV_BIG_ENDIAN</code>	<code>#if (DRV_BIG_ENDIAN == 1)</code>	Endian configuration.
<code>VS_2005_OR_HIGHER</code>	<code>#if (VS_2005_OR_HIGHER == 1)</code>	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.

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):

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

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

```
// 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 ErrorCode, 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 (MessageBox(errMsg, mError,
    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);
    pdfWriteText(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 `/msvs_lib/dynapdf.lib` or `/msvs_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++):

```
// 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 ErrorCode, 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);
    pdfWriteText(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/DynapdfInt.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.

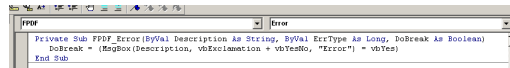
Exception handling in Visual Basic

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:

```
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.



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)
    DoBreak = (MsgBox(Description, vbInformation + vbYesNo, "Error") = vbYes)
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 not 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 stable.

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.CreateNewPDF "c:/vbout.pdf"
    FPDF.SetDocInfo diAuthor, "Jens Boschulte"
    FPDF.SetDocInfo diSubject, "My first VB output"
    FPDF.SetDocInfo diTitle, "My first VB output"
    FPDF.Append
    FPDF.SetFont "Arial", fsItalic, 30#, True, cp1252
    FPDF.WriteText taCenter, "My first VB output"
    FPDF.EndPage
```



```

FFDF_CloseFile
End Sub
' Error event procedure
Private Sub FFDF_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.ob` to your project (menu Project/Add Existing Element...).
- Finally, make sure that the `dynampdf.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 `dynampdf.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 `dynampdf.dll`. If you develop a 32 bit application on a 64 bit Windows version then copy the 32 bit version of the `dynampdf.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:

```

Private WithEvents FFDF 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 stable.

Example (Visual Basic .Net):

```

Private WithEvents FFDF 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
    FFDF = New CPDF()
End Sub

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

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

```

```

End Sub

' Error event procedure
Private Sub FPDF_PDFError(ByVal Description As String, ByVal ErrType
As Integer, ByVal 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.

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 `dynampdf.dll` can be found; just copy the DLL into the `Windows/System32` directory, finished!

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 `dynampdf.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 `dynampdf.dll`. If you develop a 32 bit application on a 64 bit Windows version then copy the 32 bit version of the `dynampdf.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 files 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#):

```

using System;
using DynaPDF;
using System.Runtime.InteropServices;

namespace hello_world
{
    class Hello_World
    {
        // Error callback function.
        static int PDFError(IntPtr Data, int ErrCode, IntPtr ErrMessage,
            int ErrType)
        {
            // The error type is a bitmask.
            Console.WriteLine("{0}\n", PtrToStringAnsi(ErrMessage));
            Console.WriteLine("\n");
            return 0; // We try to continue if an error occurs.
        }

        [STAThread]
        static void Main(string[] args)
        {
            try
            {
                String outFile = "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);
pdf.Append(); // An empty page to the file
// Before printing text you must set a font
pdf.SetFont("Arial", DynaPDF.TFStyle.fsItalic, 20.0, true,
    DynaPDF.TCodepage.cp1252);
pdf.WriteTextA(50.0, 50.0, "My first PDF output...");
pdf.WriteTextA(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))
    {
        Console.WriteLine("Make sure that you have write access
            to drive C:\nor change the output file path!\n");
        Console.ReadLine();
        return;
    }
    if (pdf.CloseFile())
    {
        Console.WriteLine("PDF file \"{0}\" successfully
            created!\n", outFile);
    }
}
pdf = null;
} catch (Exception e)
{
    Console.WriteLine(e.Message);
} Console.ReadLine();
}
}
}

```

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.

```
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: PChar; 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.SetDocInfo(diAuthor, 'Jens Boschulte');
    pdf.SetDocInfo(diCreator, 'Delphi sample project');
    pdf.SetDocInfo(diSubject, 'My first PDF file...');
```

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

pdf.CreateNewPDFR('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 are compiled with GCC 3.2 or higher which is freely available for most machine types. The configuration files are mainly designed for use with GCC and Autoconf. If you want to use another compiler you must at least change the compiler flags in the file `configure.in` and rebuild then the configure script with Autoconf. Note that finding the right compiler flags can be difficult and time consuming so that it is usually best to use GCC instead of another compiler.

System requirements:

1. Properly installed GCC (3.0 or higher) C and C++ compiler. We strongly recommended GCC 4.0 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 Mac OS X. 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.

However, if you want to change certain compiler settings, or the compiler itself, modify the file `configure.in` and execute `autoconf` without parameters. Autoconf will then 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`.

Compiler flags

When compiling DynaPDF on HP-UX the flag `-PIC` (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 stable. 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

Most DynaPDF versions for Linux and Unix are compiled with GCC 4.2 or higher. Due to certain bugs in older versions of GCC make sure that DynaPDF can be compiled at the minimum with GCC 3.0. Especially version 2.96 contains many bugs. If you use a precompiled library of DynaPDF, please note that DynaPDF is not binary compatible to GCC 2.96 or earlier. The GCC compiler should be configured with POSIX compatible thread handling if possible, although the library does not depend on it.

Compiling DynaPDF on Mac OS X

The build process on Mac OS X is very similar to Linux and UNIX operating systems. Since GCC is already the default compiler on Mac OS X it is very easy to compile DynaPDF on this operating system. All you need is a properly installed GCC compiler including make. The XCode package, which is delivered with Mac OS X, contains already the GCC compiler. The easiest way to install GCC is to install XCode 2.5 or higher because it installs GCC 3.3 as well as GCC 4.0 and all required build tools.

Once the GCC compiler was installed it is already possible to compile DynaPDF in the very same way as described for Linux or UNIX. However, Apple Computers are available with different CPU types and each CPU type requires a DynaPDF version that was compiled for the target CPU.

To make the development easier GCC 4.0 or higher can be used as cross compiler to build libraries for different CPU types on the same host system as well as Universal libraries which support multiple targets.

The build system of DynaPDF creates by default static and dynamic Universal binaries for the CPU targets `i386`, `ppc`, and `x86_64`. 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 i386 -arch ppc -arch x86_64 -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`.

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 an Interactive Form 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 of an existing 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

Global field appearance properties:

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

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.

- `GetFieldCount()` // Number of fields in the document
- `GetFieldEx()/GetFieldEx2()` // Most important properties
- `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
- `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 type of `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, 150, 20);
pdfCreateTextField(pdf, "Test", prt, false, -1, 50, 80, 150, 20);
pdfCreateTextField(pdf, "Test", -1, false, -1, 50, 110, 150, 20);
...
```

or

```
// Same result but requires more processing time
pdfCreateTextField(pdf, "Test", -1, false, -1, 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 directly 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()`
- `GetGoToActionEx()`
- `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
<code>oeOnOpen</code>	*			*
<code>oeOnClose</code>				*
<code>oeOnMouseUp</code>		*	*	*
<code>oeOnMouseDown</code>		*	*	*
<code>oeOnMouseExit</code>		*	*	*
<code>oeOnMouseDown</code>		*	*	*
<code>oeOnFocus</code>		*	*	*
<code>oeOnBlur</code>		*	*	*
<code>oeOnKeyStroke</code>		*	*	*
<code>oeOnFormat</code>		*	*	*
<code>oeOnCalc</code>		*	*	*
<code>oeOnValidate</code>		*	*	*
<code>oeOnPageVisible</code>		*	*	*
<code>oeOnPageInvisible</code>		*	*	*
<code>oeOnPageOpen</code>		*	*	*
<code>oeOnPageClose</code>		*	*	*
<code>oeOnBeforeClosing</code>	*	*	*	*
<code>oeOnBeforeSaving</code>	*	*	*	*
<code>oeOnAfterSaving</code>	*	*	*	*
<code>oeOnBeforePrinting</code>	*	*	*	*
<code>oeOnAfterPrinting</code>	*	*	*	*

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 indefinite length encoded ASN1 objects for strong encryption key lengths whereas 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 encryption 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 system'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++):

```
// Error callback function.
SI32 PDF_CALL PDFError(const void* Data, SI32 ErrCode, const char*
ErrMsg, SI32 ErrType)
{
    printf("%s\n", ErrMsg);
    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, (void*)PDFError);
    pdfCreateNewPDF(pdf, "c:/cpout.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));
    pdfWriteText(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*
ErrMsg, SI32 ErrType)
{
    printf("%s\n", ErrMsg);
    // 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;
}
```