PDF-Tools SDK

North/South America, Australia, Asia:

Tracker Software Products (Canada) Ltd.,
PO Box 79
Chemainus, BC
V0R 1K0, Canada

Sales & Admin
Tel: Canada (+00) 1-250-324-1621
Fax: Canada (+00) 1-250-324-1623

European Office:

7 Beech Gardens
Crawley Down., RH10 4JB
Sussex, United Kingdom
Sales
Tel: +44 (0) 20 8555 1122
Fax: +001 250-324-1623
http://www.tracker-software.com
sales@tracker-software.com

Support:
support@tracker-software.com
Support Forums:
http://www.tracker-software.com/forum/

Table of Contents

Part I Overview 12

1 Introduction 12
2 License Agreement 13
3 Redistribution of PDF-XChange Library components 17

Part II PXCLIB40 LIB Functions 18

1 Document Operations 18
PXC_EnableLinkAnalyzer 19
PXC_EnableSecurity 20
PXC_EnableSecurityEx 21
PXC_GetCompression 23
PXC_NewDocument 25
PXC_NewDocumentEx 26
PXC_ReleaseDocument 27
PXC_SetCallback 28
PXC_SetCompression 30
PXC_SetDocumentInfoA 33
PXC_SetDocumentInfoExA 34
PXC_SetDocumentInfoExW 35
PXC_SetDocumentInfoW 36
PXC_SetJBIG2Method 37
PXC_SetPageLayout 38
PXC_SetPageMode 39
PXC_SetPermissions 40
PXC_SetPermissions128 41
PXC_SetPermissions256 43
PXC_SetPermissions40 44
PXC_SetSpecVersion 45
PXC_SetViewerPreferences 46
PXC_SignDocumentA 48
PXC_SignDocumentBufW 51
PXC_SignDocumentUsingPFXA 53
PXC_SignDocumentUsingPFXW 55
PXC_SignDocumentW 57
PXC_WriteDocumentA 61
PXC_WriteDocumentExA 62
PXC_WriteDocumentExW 64
PXC_WriteDocumentToIStream 66
PXC_WriteDocumentW 68
2 Page Operations 69
PXC_AddPage 69
PXC_EndPage 70
PXC_GetPage 71
PXC_GetPageBox 72
PXC_GetPageIndex 73

PXC_GetPageRotation **74**
PXC_GetPagesCount **75**
PXC_GetPageSize **76**
PXC_InsertPage **77**
PXC_RemovePage **78**
PXC_SetPageBox **79**
PXC_SetPageDuration **81**
PXC_SetPageRotation **82**
PXC_SetPageTransition **82**

3 Content Creation **85**

Coordinate System **85**
PXC_CS_Concat 87
PXC_CS_Get 89
PXC_CS_Rotate 90
PXC_CS_Scale 92
PXC_CS_Skew 93
PXC_CS_Translate 94
Text and Fonts **96**
Font selection details 97
PXC_AddFontA 98
PXC_AddFontFromFileA 99
PXC_AddFontFromFileW 100
PXC_AddFontW 102
PXC_ClearNoEmbeddList 103
PXC_Draw TextExW 104
PXC_GetCurrentFont 106
PXC_GetFontInfo 107
PXC_GetStringWidthA 108
PXC_GetStringWidthW 109
PXC_GetTextOptions 110
PXC_SetCharSpacing 111
PXC_SetCurrentFont 112
PXC_SetEmbeddingOptions 114
PXC_SetFontEmbeddA 115
PXC_SetFontEmbeddW 116
PXC_SetTextLeading 117
PXC_SetTextOptions 118
PXC_SetTextRise 120
PXC_SetTextRMode 121
PXC_SetTextScaling 123
PXC_SetWordSpacing 124
PXC_TCS_Get 125
PXC_TCS_Transform 126
PXC_TextOutA 127
PXC_TextOutW 128
Images **130**
PXC_Add3DAnnotationA 130
PXC_Add3DAnnotationW 134
PXC_AddEnhMetafile 137
PXC_AddImageA 138
PXC_AddImageExA 139
PXC_AddImageExW 141
PXC_AddImageFromHBITMAP 143
PXC_AddImageFromImageXChangePage 144

PXC_AddImageFromIStream 145
PXC_AddImageFromIStreamEx 147
PXC_AddImageFromMemory 148
PXC_AddImagePattern 151
PXC_AddImageW 152
PXC_AddStdMetafile 153
PXC_AddU3DStream 154
PXC_AddU3DView ToStream 155
PXC_CloseImage 157
PXC_CropImage 158
PXC_GetImageColors 159
PXC_GetImageDimension 160
PXC_GetImageDPI 161
PXC_MakeImageGrayscale 162
PXC_MarkImageAsMask 163
PXC_PlaceImage 164
PXC_ReduceImageColors 165
PXC_ScaleImage 167
PXC_SetImageMask 168
PXC_SetImageTransColor 169
Draw ing 170
Path Construction And Draw ing 171
Graphics State Stack Operations 172
PXC_GetStateLevel 173
PXC_RestoreState 174
PXC_SaveState 174
PXC_ApplyPattern 175
PXC_Arc 176
PXC_ArcN 177
PXC_Chord 179
PXC_ChordEx 181
PXC_Circle 183
PXC_ClipPath 184
PXC_ClosePath 185
PXC_CurveTo 186
PXC_Ellipse 188
PXC_EllipseArc 189
PXC_EllipseArcEx 191
PXC_EndPath 193
PXC_FillPath 194
PXC_GetContentDC 196
PXC_GetLineInfo 197
PXC_GradientFill 199
PXC_LineTo 201
PXC_MoveTo 203
PXC_NoDash 204
PXC_Pie 205
PXC_PieEx 206
PXC_PolyCurve 208
PXC_Polygon 210
PXC_Rect 211
PXC_ReleaseContentDC 213
PXC_SetBlendMode 214
PXC_SetDash 217

PXC_SetDrawingColor 218
PXC_SetDrawingGray 219
PXC_SetFillColor 220
PXC_SetFillGray 221
PXC_SetFlat 222
PXC_SetLineCap 223
PXC_SetLineJoin 224
PXC_SetLineWidth 225
PXC_SetMiterLimit 226
PXC_SetPolyDash 227
PXC_SetStrokeAdjust 229
PXC_SetStrokeColor 230
PXC_SetStrokeGray 231
PXC_SetTransparency 232
PXC_StrokePath 233

4 AcroForms 234

AcroForms Common Functions 234
PXC_AddJavaScript 234
PXC_FormField_GetBorderInfo 235
PXC_FormField_SetAppearanceInfo 235
PXC_FormField_SetBorderInfo 236
PXC_FormField_SetFlags 237
PXC_FormField_SetReadOnly 237
PXC_FormField_SetRequired 238
PXC_FormField_SetTooltip 238

AcroForms Check Box 239
PXC_AddCheckBox 239
PXC_CheckBox_GetOptions 240
PXC_CheckBox_SetOnStateValue 240
PXC_CheckBox_SetOptions 241
PXC_CheckBox_Options 241

AcroForms Choice 243
PXC_AddChoice 243
PXC_Choice_AddItem 244
PXC_Choice_AddItems 245
PXC_Choice_SelectItems 245
PXC_Choice_SetOptions 246

AcroForms Actions 246
PXC_FormField_AddGoTo3DAction 247
PXC_FormField_AddGoToAction 248
PXC_FormField_AddJSAction 249
PXC_FormField_AddLaunchAction 250
PXC_FormField_AddNamedAction 251
PXC_FormField_AddNamedActionEx 252
PXC_FormField_AddRemoteGoToAction 253
PXC_FormField_AddResetFormAction 254
PXC_FormField_AddSubmitFormAction 255
PXC_FormField_AddURIAction 256

AcroForms Push Button 257
PXC_AddPushButton 258
PXC_PushButton_SetCaption 258
PXC_PushButton_SetIcon 259
PXC_PushButton_SetIconEx 259
PXC_PushButton_SetOptions 260

PXC_PushButton_GetOptions 260
AcroForms Radio Button 261
PXC_AddradioButtonList 261
PXC_AddRadioButton 262
PXC_RadioButton_SetCheck 262
PXC_RadioButton_SetFlags 263
PXC_RadioButton_SetOnStateValue 263
PXC_RadioButtonList_GetOptions 264
PXC_RadioButtonList_SetOptions 264
AcroForms Text Box 265
PXC_AddTextBox 265
PXC_TextBox_SetCaption 266
PXC_TextBox_SetOptions 266
5 Memory Management 267
PXC_Alloc 267
PXC_Free 267
6 Viewing/Display Options 268
Common Structures 268
PXC_3DView (COPY) 268
PXC_BorderInfo 269
PXC_ChoiceOptions 270
PXC_CommonAnnotInfo (COPY) 270
PXC_CommonFieldAppearance 272
PXC_Draw TextStruct 273
PXC_FF_IconAppearance 275
PXC_FF_SubmitAction 276
PXC_FontInfo 277
PXC_Matrix 278
PXC_PB_IconFit 279
PXC_PushButton_Options 280
Rectangles 282
PXC_PointF 282
PXC_RectF 283
PXC_TB_Options 283
PXC_TextOptions 284
PXC_TRIVERTEX 286
PXC_Watermark (COPY) 286
PXC_AddGotoAction 290
PXC_AddLaunchActionA 292
PXC_AddLaunchActionW 294
PXC_AddLineAnnotation 296
PXC_AddLink 298
PXC_AddOutlineEntryA 300
PXC_AddOutlineEntryW 305
PXC_AddTextAnnotationA 311
PXC_AddTextAnnotationW 313
PXC_AddWatermark 315
PXC_SetAnnotsInfo 317
Part III XCPRO40 LIB Functions 319
1 High-Level API 319
Document Operations 319
PXCp_AddWatermark 319

PXCp_CheckPassw ord 321
PXCp_Delete 322
PXCp_EnableSecurity 323
PXCp_EnableSecurityEx 324
PXCp_FinishReadDocument 325
PXCp_GetPermissions 327
PXCp_Init 328
PXCp_IsEncrypted 329
PXCp_PlaceContents 330
PXCp_ReadDocumentA 331
PXCp_ReadDocumentW 333
PXCp_SetCallBack 334
PXCp_SetPermissions 337
PXCp_SignDocumentBufW 338
PXCp_SignDocumentUsingPFXW 341
PXCp_SignDocumentW 343
PXCp_WriteDocumentA 346
PXCp_WriteDocumentW 348
Document Information 350
PXCp_SetSpecVersion 350
PXCp_GetDocumentInfoA 350
PXCp_GetDocumentInfoW 352
PXCp_GetDocumentInfoExA 353
PXCp_GetDocumentInfoExW 355
PXCp_GetPageLayout 357
PXCp_GetPageMode 358
PXCp_GetSpecVersion 359
PXCp_GetView erPreferences 360
PXCp_SetDocumentInfoA 360
PXCp_SetDocumentInfoW 362
PXCp_SetDocumentInfoExA 363
PXCp_SetDocumentInfoExW 364
PXCp_SetPageLayout 365
PXCp_SetPageMode 366
PXCp_SetView erPreferences 366
Document Optimization 368
PXCp_OptimizeFonts 368
PXCp_OptimizeRemoveNamedDests 369
PXCp_OptimizeStreamCompression 370
Page Information 371
PXCp_GetPagesCount 371
PXCp_PageGetBox 372
PXCp_PageGetRotate 373
PXCp_PageSetBox 374
PXCp_PageSetRotate 376
Bookmarks 377
PXCp_BMDeleteAllItems 377
PXCp_BMDeleteItem 377
PXCp_BMGetItem 378
PXCp_BMGetItem 380
PXCp_BMInsertItem 382
PXCp_BMMoveItem 383
PXCp_BMSetItemInfo 385
PXCp_GetRootBMItem 386

Page Manipulation ... **387**
PXCp_InsertPagesTo ... 387
PXCp_RemovePage ... 388
PXCp_TransformPage ... 389
Image Manipulation ... **390**
PXCp_GetDocImageAsXCPage ... 390
PXCp_ImageClearAllData ... 392
PXCp_ImageClearPageData ... 394
PXCp_ImageGetCount ... 395
PXCp_ImageGetCountOnPage ... 396
PXCp_ImageGetFromPage ... 397
PXCp_ImageGetFromPageEx ... 399
PXCp_SaveDocImageIntoFileA ... 401
PXCp_SaveDocImageIntoFileW ... 403
Thumbnails Manipulation ... **405**
PXCp_PageGetThumbnail ... 405
PXCp_PageHasThumbnail ... 406
PXCp_PageRemoveThumbnail ... 407
PXCp_PageSaveThumbnailToFile ... 408
PXCp_PageSetThumbnail ... 409
PXCp_PageSetThumbnailFromFile ... 410
PXCp_PageSetThumbnailFromHBITMAP ... 412
Annotations Manipulation ... **414**
PXCp_Add3DAnnotationA ... 414
PXCp_Add3DAnnotationW ... 419
PXCp_AddGotoAction ... 423
PXCp_AddLaunchActionA ... 426
PXCp_AddLaunchActionW ... 428
PXCp_AddLineAnnotationA ... 430
PXCp_AddLineAnnotationW ... 433
PXCp_AddLink ... 436
PXCp_AddTextAnnotationA ... 437
PXCp_AddTextAnnotationW ... 439
PXCp_SetAnnotsInfo ... 441
Text Extraction ... **443**
PXCp_ET_AnalyzePageContent ... 443
PXCp_ET_Finish ... 449
PXCp_ET_GetCurrentComposeParams ... 450
PXCp_ET_GetElement ... 451
PXCp_ET_GetElementCount ... 453
PXCp_ET_GetFontCount ... 454
PXCp_ET_GetFontInfo ... 455
PXCp_ET_GetFontName ... 456
PXCp_ET_GetFontObj ... 458
PXCp_ET_GetFontStyle ... 459
PXCp_ET_GetPageContentAsTextW ... 460
PXCp_ET_Prepare ... 463
2 Low-Level API ... **464**
General Functions ... **466**
PXCp_llGetDocTrailer ... 466
PXCp_llGetObjectByIndex ... 467
PXCp_llGetObjectsCount ... 468
PXCp_llGetPageByIndex ... 469
PXCp_llGetPageCount ... 470

PXCp_llGetRootObject 471
Variant Functions 472
PXCp_VariantCopyFrom 472
PXCp_VariantCreate 473
PXCp_VariantDelete 473
PXCp_VariantGetArray 474
PXCp_VariantGetBOOL 475
PXCp_VariantGetDictionary 476
PXCp_VariantGetDouble 477
PXCp_VariantGetInt 478
PXCp_VariantGetName 478
PXCp_VariantGetObject 479
PXCp_VariantGetString 480
PXCp_VariantGetType 481
PXCp_VariantGetType 482
PXCp_VariantSetBOOL 483
PXCp_VariantSetDictionary 484
PXCp_VariantSetDouble 485
PXCp_VariantSetInt 486
PXCp_VariantSetName 487
PXCp_VariantSetNull 488
PXCp_VariantSetObject 488
PXCp_VariantSetString 489
Object Manipulation Functions 490
PXCp_ObjectCreateNew 490
PXCp_ObjectGetBody 491
PXCp_ObjectGetDictionary 492
PXCp_ObjectGetStream 493
PXCp_ObjectSetBody 494
PXCp_ObjectSetDictionary 495
PDF Array Functions 496
PXCp_ArrayAppendFrom 496
PXCp_ArrayClearAll 497
PXCp_ArrayCreate 497
PXCp_ArrayDelete 498
PXCp_ArrayDeleteAt 499
PXCp_ArrayGetByIndex 500
PXCp_ArrayGetCount 501
PXCp_ArrayInsertAt 502
PXCp_ArraySetValue 503
PDF Dictionary Functions 505
PXCp_DictionaryAppendFrom 505
PXCp_DictionaryClearAll 506
PXCp_DictionaryClearValue 507
PXCp_DictionaryCreate 508
PXCp_DictionaryDelete 508
PXCp_DictionaryGetCount 509
PXCp_DictionaryGetKeyByPDFStr 510
PXCp_DictionaryGetKeyByStr 511
PXCp_DictionaryGetPair 512
PXCp_DictionarySetKeyByPDFStr 514
PXCp_DictionarySetKeyByStr 515
PDF String Functions 517
PXCp_StringCreate 517

PXCp_StringDelete 518
PXCp_StringGetB 518
PXCp_StringSetA 520
PXCp_StringSetB 520
PXCp_StringSetW 521
PDF Stream Functions 522
PXCp_StreamCreate 522
PXCp_StreamDecode 523
PXCp_StreamEncode 524
PXCp_StreamGetFilterParameters 525
PXCp_StreamGetFiltersCount 527
PXCp_StreamGetLength 528
PXCp_StreamReadData 529
PXCp_StreamRemove 531
PXCp_StreamRevertToOriginal 532
PXCp_StreamWrite 533
PXCp_StreamWriteEncodedData 534
3 Common Structures 536
PXC_3DView 536
PXC_CommonAnnotInfo 537
PXC_AnnotBorder 538
PXC_Watermark 539
PXCp_BMInfo 543
PXCp_BMDestination 545
PXCp_CompressParam 548
PXCp_ContentPlaceInfo 549
PXCp_CopyPageRange 550
PXCp_FilterParam 551
PXCp_SaveImageOptions 554
PXP_TEFontInfo 559
PXP_TETextComposeOptions 561
PXP_TextElement 563

Part IV Error Handling 565

1 LIB Error Handling 567
LIB Error Codes 567
PXC_Err_FormatErrorCode 568
PXC_Err_FormatFacility 569
PXC_Err_FormatSeverity 569
Example (C++) 570
2 PRO Error Handling 571
PRO Error Codes 571
PXCp_Err_FormatErrorCode 574
PXCp_Err_FormatFacility 575
PXCp_Err_FormatSeverity 575
Example (C++) 576
3 DSErrorLookUp Utility 577

Part V Tracker Software Products 577

1 Our Products 578
2 Contact Us 578

Index 581

1 Overview

1.1 Introduction

Introduction

PDF-XChange & Tools Developer Library for the PDF format – Version 4.x

Introduction

PDF-XChange and Tools are a comprehensive and expanding set of professional tools designed for use by software developers for the creation and manipulation of highly optimized Adobe compatible PDF format files. None of our products require any 3rd party dll's or tools installed)other than MS Windows and some secondary MS Library files (e.g. for Windows GDI+ in W2K systems).

All of the products detailed in this documentation, once licensed, allow the developer to distribute the 'Run Time' components on a Royalty Free basis to the developers End User clients of an 'End user' application.

They may not be used to develop Toolkits or Components of any type for use by other non-licensed developers. For more information on licensing please read the license agreement and if any doubt as to whether your intended use would be in breach of the license terms please contact us to discuss your needs in more depth as we do offer alternate licensing and will tailor our agreement to meet your needs in most circumstances under different terms of supply.

The PDF-XChange & Tools Library Dll's for Developers.

These are offered for purchase either individually or in a 'PRO' package containing (limited) Royalty Free distribution rights to both the Driver API and library packages.

PDF-XChange Viewer SDK

When Licensing the PDF-XChange/Tools SDK's - you also benefit from a limited distribution license to use the PDF-XChange Viewer SDK in your software applications at no extra cost.

You can view more info on your free entitlement and any extra costs etc - from this web page : http://www.docu-track.com/home/dev_tools/pdf/PDF-XChange_Viewer_SDK/

Please note: Your PDF-XChange/Tools SDK license info is **not functional for the Viewer SDK** - you must complete and return your Viewer SDK License to us before your License info will be sent to you to ensure you fully understand the viewer SDK licensing and any additional costs you may be required to pay after your limited free distribution rights are exhausted.

Support

Support is available direct from our user forums http://www.docu-track.com/forum/index.php.

We recommend that developers use the evaluation download as extensively as possible prior to purchase. The evaluation versions are fully functional with no time out or other crippling mechanism – save that a watermark is stamped on any PDF page generated by the Driver/Library tools – only on purchase will you be provided with the serial number and unlock string required by each component to be passed in your code (see the demo applications provided) to enable PDF generation without this demo watermark stamp.

By virtually creating your application to the point where you are ready for distribution before you purchase, you are guaranteed satisfaction and we do not disappoint developers asking for refunds once purchased – we do not offer any money back options – so please ensure you are 100% satisfied before you purchase.

Developer's may also find it useful when developing applications for the purpose of creating and manipulating Adobe PDF Formats - to download the documentation relevant to this format from the Adobe web site - at the time of writing this is currently free and may prove useful in explaining in more detail the functionality available. http://www.adobe.com/.

Tracker Software Products Ltd also provide End User and Developer Tool Kits for the creation and manipulation of PDF and Image files and Virtual Printer Drivers. For more information please visit http://www.docu-track.com/.

1.2 License Agreement

License Agreement

Top Previous Next

License Agreement PDF-XChange PRO, Drivers API and PDF-Tools Software Developer Kits (SDK) from Tracker Software Products Ltd 2001- 2008. Versions 1 - 4.x

PRINTED BELOW IN ITS ENTIRETY IS THE LICENSE AGREEMENT GOVERNING YOUR USE OF THE SOFTWARE.

PLEASE READ THE LICENSE AGREEMENT.

IMPORTANT

TRACKER SOFTWARE PRODUCTS LTD. IS WILLING TO LICENSE THE ENCLOSED SOFTWARE TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THE LICENSE AGREEMENT PRINTED BELOW. PLEASE READ THE TERMS CAREFULLY BEFORE OPENING THE PACKAGE CONTAINING THE DISKETTE(S)/CD-R(S), Electronic File OR CLICKING THE ACCEPT BUTTON DURING INSTALLATION, AS SUCH CONDUCT INDICATES YOUR ACCEPTANCE TO ALL OF THE TERMS OF THIS LICENSE AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS, TRACKER SOFTWARE PRODUCTS LTD IS UNWILLING TO LICENSE THE SOFTWARE TO YOU, IN WHICH CASE YOU MUST IMMEDIATELY RETURN THE PACKAGE AND ALL ACCOMPANYING MATERIAL TO TRACKER SOFTWARE PRODUCTS LTD. OR YOUR AUTHORIZED DEALER FOR A FULL REFUND.

This License Agreement ("Agreement") is a legal agreement between Tracker Software Products Ltd, (Tracker), a Company registered in England, principally located in Turners Hill, West Sussex, England, and you, the user ("Licensee"), and is effective the date Licensee opens the package containing the diskette(s)/CD-R(s) or otherwise uses the enclosed software product.

This Agreement covers all materials associated with Tracker's PDF-XChange API/SDK developer's toolkit products including the enclosed software product ("Software").

1. GRANT OF DEVELOPMENT LICENSE

TRACKER grants Licensee a non-exclusive, non-transferable, worldwide license for one (1) programmer to install the Software on a single personal computer and use the Software and one copy of the associated user documentation contained in the accompanying user manual, "online" help and Acrobat files ("Documentation") in the development of End User software application's as contemplated in section 2 below (herein, the "Application Software"). If additional programming seats are needed, Licensee should contact TRACKER for discounted license pricing. The license granted hereunder applies only to the designated version of the enclosed Software. If the Software is an upgrade or cross grade, it, and the product that was upgraded/cross graded constitute a single copy of the Software for purposes hereof and the new version and product that was upgraded/cross graded cannot be used by two people at the same time. This License allows you to create and distribute a maximum 100,000 end user application licenses, for single desktop access, when installed on a Windows Server - each user attached to that server and with access to the application shall be considered to be using a single desktop license, concurrent licensing shall not be used to account for licenses used, see Clause 3(i) below for further details on Royalty Free Distribution rights included.

2. END USER APPLICATION

The Application Software developed by Licensee must be an "End User Application." An "end user application" is a specific application program that is licensed to a person or firm for business or personal use and not with a view toward redistributing the application or any part of the application, and may be either an application that is used by Licensee internally, or an application that is commercially distributed to end users for their use. A user of an end user application may not modify or redistribute the application and may not copy it (other than for archival purposes). Licensee's license agreement covering the Application Software must contain restrictions prohibiting redistribution, modification and copying of the Application Software. The license rights hereunder do not apply to development and deployment of software products such as Printer Drivers, ActiveX controls, plug-ins, authoring tools, development toolkits, compilers, operating systems and also software products where the sole or a significant function is to generate 'PDF' format files (as defined by Adobe Systems Inc') and other file formats from 3rd party software applications not developed by the licensee, indirectly or otherwise, - such as Microsoft's 'Office' suite and component applications other than for the purpose of creating and then storing such files within a structured application for the archival and management of documents that is developed by the licensee and any other software not falling within the definition of an end user application. If Licensee wishes to develop a product outside the scope of this license, Licensee should contact TRACKER'S OEM Sales department to see if a special license is available.

3. GRANT OF DUPLICATION AND DISTRIBUTION LICENSE

The Software includes certain runtime libraries and files intended for duplication and distribution by Licensee within the Application Software to the user of Application Software ("Redistributables"). The Redistributable components of the Software are those files specifically designated as being distributable in the "Files to be Included with Your Application" section of the Online Help file, the terms of which are hereby incorporated herein by reference. Licensee should refer to the Documentation and specifically the "Online Help" file for additional information regarding the Redistributables. Under TRACKER'S copyright, and subject to all the restrictions and conditions set forth in this Agreement and the Documentation, TRACKER hereby grants Licensee (and only Licensee) a non-exclusive, non-transferable, worldwide license to reproduce exact copies of the Redistributables and include such files in the Application Software, and to deploy the Application Software internally and/or distribute the Application Software, directly or through customary distribution channels, to end users on a royalty free basis*, provided that such redistribution does not exceed 100,000 units other than for demonstration purposes, see clause item 3.(i) should the quantity to be distributed exceed this (The foregoing sentence does not apply if Licensee has licensed Tracker's PDF-XChange Printer Driver for Windows - for 'End User' use (as Opposed to the API/SDK Toolkit Licensed products. This product requires additional run time licensing based on use/distribution of the Application Software: see Section 4, "Duplication and Distribution of Royalty Bearing Versions " below.) If Licensee wishes to use an OEM who will modify the Application Software and copy it, Licensee must first obtain an OEM distribution license from TRACKER or must require the OEM to obtain a license from TRACKER. Duplication or Redistribution of the Application Software, or any portion thereof, by the users of the Application Software, without a separate written redistribution license from TRACKER is prohibited. If the enclosed Software is packaged "For Evaluation Only," no right to copy and/or distribute the Redistributables is granted. No rights to copy or redistribute the Application Software are granted until such time as Licensee has properly registered the Software with TRACKER and otherwise complied with this Agreement. Unless otherwise agreed in writing by Tracker, developer must distribute any Print drivers included using the Tracker Installation executable file provided for this purpose to ensure correct distribution and licensing adherence

3(i) Royalty Free Distribution Limitations : Once licensed, you may create and distribute a maximum of 100,000 end user application licenses incorporating any part of the allowed elements of this developer's kit -

should the number of licenses you intend to distribute (or have already distributed) exceed this figure (other than for demonstration, evaluation or publicity purposes) then you must contact Tracker Software Products immediately and prior to (further) distribution or as soon as it becomes known to you that this figure will or has been exceeded to discuss alternative licensing options. Further you agree at any time, on request and within 30 days of such request, to supply a duly audited and notarised account of application licenses delivered/sold where components of this licensed developer kit have been used and distributed - other than for demonstration, evaluation or publicity purposes.

4. DUPLICATION AND DISTRIBUTION OF ROYALTY BEARING VERSIONS OF THE SOFTWARE

If the enclosed Software is Trackers PDF-XChange Printer Driver designed for 'End User' use as opposed to a component of the SDK kits available, (collectively known as "Royalty Bearing Versions") Licensee is required to enter into a separate duplication and distribution license prior to deployment of the Application Software. No duplication or distribution rights are granted hereunder with respect to the Royalty Bearing Versions.

5. OTHER RESTRICTIONS

The licenses granted under this Agreement are expressly conditioned upon Licensee’s compliance with all the terms and conditions of this Agreement. Licensee may not use, copy, rent, lease, sell, sublicense, assign or otherwise transfer the Software except as expressly provided for in this Agreement. Licensee may make a reasonable number of archival copies of the Software. Except for the Redistributables, Licensee shall not distribute any files contained in the Software, including without limitation, .CLW, .INC, .TPL, . CHM, .DRV, .LIB, .H, .MAK, .DEF, .TXT, .PDF or .HLP files. Licensee shall not reproduce, copy or transfer any Documentation, except Licensee may use the sample source code examples contained in the Documentation for the purpose of developing the Application Software. Upon TRACKER’S request, Licensee agrees to send TRACKER one demonstration copy of the Application Software. If the Software is PDF-XChange, the Application Software may only access the .DLL file(s) directly and not through the PDF-XChange Print driver (.DRV file(s).) Licensee may only directly access the .DLL file(s) if Licensee has a license in good standing for a PDF-XChange product with an API/SDK License such as PDF-XChange SDK or PDF-Tools SDK, and only then the appropriat Library DLL’s relevant to each SDK. Any distributor or reseller of Application Software appointed by Licensee must be subject to a binding agreement that includes provisions no less protective of TRACKER’S intellectual property rights in the Software as it is protective of Licensee’s rights in its own software. Licensee acknowledges that the Software, in source code form, remains a confidential trade secret of TRACKER and/or its suppliers and therefore Licensee agrees that it shall not modify, decompile, disassemble or reverse engineer the Software or attempt to do so except as permitted by applicable legislation. Licensee agrees to refrain from disclosing the Software (and to take reasonable measures with its employees to ensure they do not disclose the Software) to any person, firm or entity except as expressly permitted herein. Specifically, Licensee will not disclose or publish any unlock codes or instruction sets provided by TRACKER relating to the Software. If Licensee wishes to use the Software in a manner prohibited by this Agreement, Licensee should contact TRACKER’S OEM department to determine whether a special license may be obtained.

6. PROPRIETARY RIGHTS; COPYRIGHT NOTICES

Except for the limited license granted herein, TRACKER, and its suppliers, retains exclusive ownership of all proprietary rights (including all ownership rights, title, and interest) in and to the Software. Licensee agrees not to represent that TRACKER is affiliated with or approves of Licensee’s Application Software in any way. Except as required hereby, Licensee shall not use TRACKER’S name, trademarks, or any TRACKER designation in association with Licensee’s Application Software. The Application Software may contain the following copyright notice in the "About box": "Portions of this product were created using PDF-XChange From Tracker Software Products Ltd ©20001, ALL RIGHTS RESERVED."

7. EXPORT LAW

Licensee acknowledges and agrees that the Software and Application Software may be subject to restrictions and controls imposed by the United States Export Administration Act, as amended (the "ACT"), and the regulations there under. Licensee agrees and certifies that neither the Software nor any direct product thereof (e.g. the Application Software) is being or will be acquired, shipped, transferred or re-exported, directly or indirectly, into any country prohibited by the ACT and the regulations there under or will be used for any purpose prohibited by the same. Licensee acknowledges that the Software may include "technical data" subject to export and re-export restrictions imposed by U.S. law. Licensee bears all responsibility for export law compliance and will indemnify TRACKER against all claims based on Licensee’s exporting of the Application Software.

8. U.S. GOVERNMENT RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19, as applicable. Manufacturer/Contractor is TRACKER SOFTWARE PRODUCTS LTD, Units 1-3, Burleigh Oaks, East Street, Turners Hill, West Sussex. England.RH10 4PZ

9. TERM

The license granted hereby is effective until terminated. Licensee may terminate the license by returning the Software and Documentation to TRACKER, without refund, and destroying all copies thereof in any form. TRACKER may terminate the licenses if Licensee fails to comply with any term or condition of this Agreement or any corresponding duplication and distribution agreement for Printer Driver Products. Upon such termination, Licensee shall cease using the Software and cease using or distributing the Application Software containing the Redistributables. All restrictions prohibiting Licensee’s use of the Software and intellectual property provisions relating to Software running to the benefit of TRACKER will survive termination of the license pursuant hereto. Termination will not affect properly granted end user licenses of the Application Software distributed by Licensee prior to termination.

10. EXCLUSION OF WARRANTIES

TRACKER and its suppliers offer and Licensee accepts the Software "AS IS." TRACKER and its suppliers do not warrant the Software will meet Licensee’s requirements or will operate uninterrupted or error-free. ALL WARRANTIES, EXPRESS OR IMPLIED, ARE EXCLUDED FROM THIS AGREEMENT AND SHALL NOT APPLY TO ANY SOFTWARE LICENSED UNDER THIS AGREEMENT, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

11. LICENSEE’S REMEDIES: LIMITATIONS

LICENSEE’S SOLE AND EXCLUSIVE REMEDIES AGAINST TRACKER ON ANY AND ALL LEGAL OR EQUITABLE THEORIES OF RECOVERY SHALL BE, AT TRACKER’S SOLE DISCRETION, (A) REPAIR OR REPLACEMENT OF DEFECTIVE SOFTWARE; OR (B) REFUND OF THE LICENSE FEE PAID BY LICENSEE.

12. NO LIABILITY FOR CONSEQUENTIAL DAMAGES

In no event shall TRACKER, or its suppliers, be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information or other pecuniary loss) arising out of use of or inability to use the Software, even if TRACKER or its dealer have been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of certain implied warranties or the exclusion or limitation of incidental or consequential damages, in which case and to the extent such exclusion or limitation is not allowed, some of the foregoing limitations and exclusions may not apply to Licensee.

13. GENERAL

This Agreement shall be interpreted, construed, and enforced according to the laws of England. In the event of any action under this Agreement, the parties agree that courts located in England will have exclusive jurisdiction and that a suit may only be brought in England, and Licensee submits itself for the jurisdiction and venue of the courts located in England. This Agreement constitutes the entire agreement and understanding of the parties and may be modified only in writing signed by both parties. No officer, salesman, or agent has any authority to obligate TRACKER by any terms, stipulations or conditions not expressed in the Agreement. All previous representations and agreements, if any, either verbal or written, referring to the subject matter of this Agreement are void. If any portion of this Agreement is determined to be legally invalid or unenforceable, such portion will be severed from this Agreement and the remainder of the Agreement will continue to be fully enforceable and valid. This Agreement, and the rights hereunder, may not be assigned by Licensee, whether by oral or written assignment, sale of assets, merger, consolidation or otherwise, without the express written consent of TRACKER. Licensee agrees to be responsible for any and all losses or damages arising out of or incurred in connection with the Application Software. Licensee agrees to defend, indemnify and hold TRACKER harmless from any such loss or damage, including attorney’s fees, arising from the use, operation or performance of the Application Software or Licensee’s breach of any terms of this Agreement. Licensee shall be responsible for paying all state and federal use, sales or value added taxes, duties or governmental charges, whether presently in force or which come into force in the future, related to the distribution and sale of the Application Software and will indemnify TRACKER against any claim made against TRACKER relating to any such taxes or assessments.

Copyright © 2001-2008 Tracker Software Products Ltd, Units 1-3, Burleigh Oaks, East Street, Turners Hill, West Sussex. England.RH10 4PZ

ALL RIGHTS RESERVED. All Other Trademark’s acknowledged & are the property of their respective owners.

PDF-XChange Templates & Classes for Clarion for Windows (PDF-XChange-API/SDK customers only)
PDF-XChange API/SDK (PDF-XChange-API/SDK customers only)
PDF-XChange SDK Printer Driver (PDF-XChange-Print Driver customers only)
PDF-Tools SDK Templates & Classes for Clarion for Windows (PDF-Tools-API/SDK customers only)
Delphi Components for PDF-XChange and/or PDF-Tools SDK products.
All Demo/Evaluation components and examples for PDF-XChange and/or PDF-Tools SDK products.

1.3 Redistribution of PDF-XChange Library components

Redistribution of PDF-XChange Library components

Top Previous Next

The required library dll distribution list for specific functionality is detailed below along with any dependency on other library dll’s.

Functionality Type	**DLL(s) Name**	**Version**
Common Libraries	dscrt40.dll, xccdx40.dll	4.00.x
PXC-XChange/Tools Library Core DLL's (always required)	pxclib40.dll, XCPRO40.dll	4.00.x
Image-XChange Core Built-in support of: **BMP**, **WMF**, **AMF**, and **EMF** formats.	ixclib40.dll	4.00.x

Image-XChange Formats Module Supports following file formats: **PNG**, **JNG**, **GIF**, **ICO**, **PBM**, **PGM**, **PPM**, **JBIG**, **JBIG2**, **JPEG**, **JPEG2000**, **WBMP**, **PCX**, **DCX**, **TGA**	fm40base.dll	4.0 0.x
Image-XChange Formats Module Supports **TIFF** file format.	fm40tiff.dll	4.0 0.x
Net Library used by **PDF-XChange/Tools Library**.	netlib40.dll	4.0 0.x
Scanner Support Library.	xcscan40.dll	4.0 0.x

2 PXCLIB40 LIB Functions

2.1 Document Operations

Document Operations

Top Previous Next

The following operations are applied at the Document level:

- PXC_EnableLinkAnalyzer
- PXC_EnableSecurity
- PXC_EnableSecurityEx
- PXC_GetCompression
- PXC_NewDocument
- PXC_ReleaseDocument
- PXC_SetCallback
- PXC_SetCompression
- PXC_SetDocumentInfoA
- PXC_SetDocumentInfoExA
- PXC_SetDocumentInfoExW
- PXC_SetDocumentInfoW
- PXC_SetJBIG2Method
- PXC_SetPageLayout
- PXC_SetPageMode
- PXC_SetPermissions
- PXC_SetPermissions128
- PXC_SetPermissions40
- PXC_SetSpecVersion
- PXC_SetViewerPreferences
- PXC_SignDocumentA
- PXC_SignDocumentBufW
- PXC_SignDocumentUsingPFXA
- PXC_SignDocumentUsingPFXW
- PXC_SignDocumentW
- PXC_WriteDocumentA
- PXC_WriteDocumentExA
- PXC_WriteDocumentExW
- PXC_WriteDocumentToIStream
- PXC_WriteDocumentW

2.1.1 PXC_EnableLinkAnalyzer

PXC_EnableLinkAnalyzer

Top Previous Next

The **PXC_EnableLinkAnalyzer** enables or disables the internal link analyzer which analyzes the drawn text for valid URLs. If a valid link is found, it makes the link 'Hot' in the specified location.

```
HRESULT  PXC_EnableLinkAnalyzer(
    _PXCDocument* pdf,
    BOOL bEnable
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

bEnable

[in] If this parameter is `TRUE`, the link analyzer will be enabled; otherwise it will be disabled.
By default, the link analyzer is disabled..

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Enable link analyzer


    _PXCDocument*         pdf;

    ...

    // Enable analyzer

    HRESULT hr = PXC_EnableLinkAnalyzer(pdf, TRUE);

    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.1.2 PXC_EnableSecurity

PXC_EnableSecurity

Top Previous Next

The **PXC_EnableSecurity** function enables or disables PDF security for the document.

This function is deprecated by function **PXC_EnableSecurityEx**.

```
HRESULT  PXC_EnableSecurity(
    _PXCDocument* pdf,
    BOOL bEnable,
    LPCSTR UserPwd,
    LPCSTR OwnerPwd
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

bEnable

[in] If the value of this parameter is TRUE, security will be enabled with the specified user and owner passwords. If this parameter is FALSE, security will be disabled. In this case the parameters *UserPwd* and *OwnerPwd* will be ignored.

UserPwd

[in] This parameter specifies the User password for the document. The User password grants access only to the document elements which have been enabled with the designated permissions.

OwnerPwd

[in] This parameter specifies the Owner password for the document. The Owner password grants access to the entire document. If the Owner and User passwords are the same the User password will be used.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    _PXCDocument*        pdf;
...

// Document passwords:

LPCSTR        UserPassword = "user pass";
LPCSTR        OwnerPassword = "owner pass";

// Switch on securety for the document, and set user and owner password:
```

```
HRESULT res = PXC_EnableSecurity(pdf, TRUE, UserPassword, OwnerPassword);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
// Write the document

...

// Now after writing the document while opening it in the PDF viewr
// one will be asked for the password
```

2.1.3 PXC_EnableSecurityEx

PXC_EnableSecurityEx

Top Previous Next

PXC_EnableSecurityEx enables or disables PDF security for the document.

```
HRESULT  PXC_EnableSecurityEx(
    _PXCDocument* pdf,
    PXC_SecurityMethod nMethod,
    LPCSTR UserPwd,
    LPCSTR OwnerPwd
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

nMethod

[in] Specifies the enctyption method used to encrypt the document content. Can be one of the following value:

Constant	Value	Meaning
psm_None	0	Security disabled. Parameters *UserPwd* and *OwnerPwd* will be ignored.
psm_RC4	1	RC4 encryption method will be used. RC4 can has 40-bit or 128-bit key length.
psm_AES	2	AES 128 bit encryption method will be used.
psm_AES256	3	AES 256 bit encryption method will be used.

If the value of this parameter is `psm_AES`, or `psm_RC4`, security will be enabled with the specified user and owner passwords. One of functions **PXC_SetPermissions40**, **PXC_SetPermissions128**, or **PXC_SetPermissions** should be called after this function to specify user's permission for the document.

Note:

With AES encryption only key length 128 (for psm_AES) or 256 (for psm_AES256) can be used.

UserPwd

[in] This parameter specifies the User password for the document. With the User password you will have access only to the document elements which have been enabled with the designated permissions.

OwnerPwd

[in] This parameter specifies the Owner password for the document. With Owner password you will have access to the entire document. If the Owner and User passwords are the same the User password will be used.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    _PXCDocument* pdf;
    ...

    // Document passwords:

    LPCSTR         UserPassword = "user pass";
    LPCSTR         OwnerPassword = "owner pass";

    // Switch on securety for the document, and set user and owner password:

    HRESULT res = PXC_EnableSecurityEx(pdf, psm_RC4, UserPassword,
OwnerPassword);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
    // Write the document

    ...

    // Now after writing the document while opening it in the PDF viewr
    // one will be asked for the password
```

2.1.4 PXC_GetCompression

PXC_GetCompression

Top Previous Next

PXC_GetCompression retrieves the current setting for compression method(s) for images and text in the PDF document.

```
HRESULT  PXC_GetCompression(
    const _PXCDocument* pdf,
    BOOL* bText,
    BOOL* bAscii,
    PXC_CompressionType* cColor,
    DWORD* jpegQual,
    PXC_CompressionType* cIndexed,
    PXC_CompressionType* cMono
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by function **PXC_NewDocument**.

bText

[in] if *bText* is equal to **TRUE** then text content will be compressed.

bAscii

[in] if *bAscii* is equal to **TRUE** then text content will be written in ASCII format.

cColor

[in] *cColor* specifies the compression method for color images. *(See comments for possible values)*

jpegQual

[in] *jpegQual* pointer to a variable of the type `DWORD` that recieves the current compression level for `JPEG` compression method. Possible values are in the range from **1** (low quality) to **10** (hi quality).

cIndexed

[in] *cIndexed* specifies the compression method for indexed images. *(See comments for possible values)*

cMono

[in] *cMono* specifies the compression method for monochrom images. *(See comments for possible values)*

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Possible values of **`PXC_CompressionType`** for **Color/Grayscale** images are:

Method	Value	Definition

ComprType_C_NoCompress	0	No compression.
ComprType_C_JPEG	0x0001	Use **JPEG** compression (DCT method).
ComprType_C_Deflate	0x0002	Use **Flate** (ZIP) method.
ComprType_C_JPEG_Deflate	**(ComprType_C_JPEG | ComprType_C_Deflate)**	Same as ComprType_C_JPEG, and then after **JPEG** compression the **Flate** method is applied to the data.
ComprType_C_J2K	0x0004	Use **JPEG 2000** compression.
ComprType_C_J2K_Deflate	**(ComprType_C_J2K | ComprType_C_Deflate)**	Same as ComprType_C_J2K, and then after **JPEG 200** compression the **Flate** method is applied to the data.
ComprType_C_Auto	0xFFFF	Use automatically selected method for best compression.

Possible values for **PXC_CompressionType** for **Indexed** images are:

Method	Value	Definition
ComprType_I_NoCompress	**ComprType_C_NoCompress**	No compression.
ComprType_I_Deflate	**ComprType_C_Deflate**	Use **Flate** (ZIP) method.
ComprType_I_RunLength	0x0008	Use **RLE** method.
ComprType_I_LZW	0x0010	Use **LZW** compression.
ComprType_I_Auto	**ComprType_C_Auto**	Use automatically selected method for best compression.

Possible values of **PXC_CompressionType** for **Monochrome** images are:

Method	Value	Definition
ComprType_M_NoCompress	**ComprType_C_NoCompress**	No compression.
ComprType_M_Deflate	**ComprType_C_Deflate**	Use **Flate** (ZIP) method.
ComprType_M_RunLength	**ComprType_I_RunLength**	Use **RLE** method.
ComprType_M_CCITT3	0x0020	Use CCITT Fax Mode 3 method.
ComprType_M_CCITT4	0x0040	Use CCITT Fax Mode 4 method.
ComprType_M_JBIG2	0x0080	Use **JBIG2** compression.
ComprType_M_Auto	**ComprType_C_Auto**	Use automatically selected method for best compression.

Example (C++).

```
_PXCDocument*        pdf;
    ...

    // Get next compression options:

    // 1. Compress text?

    BOOL bText = FALSE;

    // 2. Text content will be written in ASCII format?

    BOOL bAscii = FALSE;
    // 3. Color images compression

    PXC_CompressionType cColor;

    // 4. JPEG quality

    DWORD jpegQuality;

    // 5. Indexed images compression

    PXC_CompressionType iColor;

    // 6. Monochrome images compression

    PXC_CompressionType mColor;

    // Get compression settings

    HRESULT res = PXC_GetCompression(pdf, &bText, &bAscii, &cColor,
&jpegQuality, &iColor, &mColor);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.1.5 PXC_NewDocument

PXC_NewDocument

Top Previous Next

PXC_NewDocument creates a PDF object, as required by the majority of functions in the **PDF-XChange Library** - either explicitly or implicitly.

When no longer required the PDF object must be destroyed using **PXC_ReleaseDocument**.

```
HRESULT  PXC_NewDocument(
    _PXCDocument** pdf,
    LPCSTR key,
    LPCSTR devCode
);
```

Parameters

pdf

[in, out] Pointer to the variable of a type `_PXCDocument*` that will receive the created PDF object.

key

[in] Pointer to a null-terminated string which contains your licence key for the **PDF-XChange Library**. This parameter can be `NULL`, if the latter is true, the library will operate in 'evaluation' mode

and a demo stamp will be printed on all output.

devCode

[in] Pointer to a null-terminated string which contains your individual developer code for the **PDF-XChange Library**. This parameter can be `NULL`, if the latter is true, the library will operate in 'evaluation' mode and a demo stamp will be printed on all output.

Return Values

If the function succeeds, the return value is `PXC_OK`, and a variable pointer to *pdf* will contain valid PDF object.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
_PXCDocument* pdf = NULL;
   // Please note - RegCode and DevCode are case sensitive
   LPCSTR regcode = "<Your personal serial/keycode code here>";
   LPCSTR devcode = "<Your personal developers' code here>";
   HRESULT res = PXC_NewDocument(&pdf, regcode, devcode);
   if (IS_DS_FAILED(res))
       return res;
   ...
   PXC_ReleaseDocument(pdf);
```

2.1.6 PXC_NewDocumentEx

PXC_NewDocumentW

Top Previous Next

PXC_NewDocumentW creates a PDF object, as required by the majority of functions in the **PDF-XChange Library** - either explicitly or implicitly. This function allows to create new document in PDF/A-1b mode.

When no longer required the PDF object must be destroyed using **PXC_ReleaseDocument**.

```
HRESULT  PXC_NewDocumentEx(
    _PXCDocument** pdf,
    LPCSTR key,
    LPCSTR devCode,
    PXC_PDFX_Mode pdfMode
);
```

Parameters

pdf

[in, out] Pointer to the variable of a type `_PXCDocument*` that will receive the created PDF object.

key

[in] Pointer to a null-terminated string which contains your licence key for the **PDF-XChange Library**. This parameter can be `NULL`, if the latter is true, the library will operate in 'evaluation' mode and a demo stamp will be printed on all output.

devCode

[in] Pointer to a null-terminated string which contains your individual developer code for the **PDF-XChange Library**. This parameter can be `NULL`, if the latter is true, the library will operate in

'evaluation' mode and a demo stamp will be printed on all output.

pdfMode

[in] Specify the mode of new document. Possible values are:

Constant	Value	Meaning
PDFX_None	0	Standard mode. With this value this function is equal to **PXC_NewDocument**.
PDFA_1a	1	Document conforming to PDF/A-1a standard will be created. Currently in beta stage and not supported.
PDFA_1b	2	Document conforming to PDF/A-1b standard will be created. When document created in this mode, many functions (for example, PXC_SetSpecVersion or PXC_EnableSecurity) may return an error PXC_ERR_NOT_AVAIL_IN_PDFA because not all operations are allowed in this mode.

Return Values

If the function succeeds, the return value is `PXC_OK`, and a variable pointer to *pdf* will contain valid PDF object.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
_PXCDocument* pdf = NULL;
// Please note - RegCode and DevCode are case sensitive
LPCSTR regcode = "<Your personal serial/keycode code here>";
LPCSTR devcode = "<Your personal developers' code here>";
HRESULT res = PXC_NewDocumentEx(&pdf, regcode, devcode, PDFA_1b);
if (IS_DS_FAILED(res))
    return res;
...
PXC_ReleaseDocument(pdf);
```

2.1.7 PXC_ReleaseDocument

PXC_ReleaseDocument

Top Previous Next

PXC_ReleaseDocument releases the PDF object, created previously using the **PXC_NewDocument** function. You must call this function either when the PDF object is no longer rquired or is complete.

Warning! If you refer to the PDF object after calling this function your program will 'crash'!

```
HRESULT  PXC_ReleaseDocument(
    _PXCDocument* pdf
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
 _PXCDocument* pdf = NULL;
  // Please note - RegCode and DevCode are case sensitive
  LPCSTR regcode = "<Your personal serial/keycode code here>";
  LPCSTR devcode = "<Your personal developers' code here>";
  HRESULT res = PXC_NewDocument(&pdf, regcode, devcode);
  if (IS_DS_FAILED(res))
      return res;
  ...

  // After all operations with the document are done
  // document's handle should be released:

  PXC_ReleaseDocument(pdf);
```

2.1.8 PXC_SetCallback

PXC_SetCallback

Top Previous Next

PXC_SetCallback sets the callback function, to be called during saving the document to a (disk) file.

```
HRESULT  PXC_SetCallback(
    _PXCDocument* pdf,
    CALLBACK_FUNC clbFn,
    LPARAM clbParam
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

clbFn

[in] *clbFn* specifies the callback function. It must be defined as CALLBACK_FUNC:

```
typedef BOOL (__stdcall *CALLBACK_FUNC)(DWORD state, DWORD level, LPARAM param);
```

The first parameter of this function is the callback state, the second indicates the progress level (see table below), and the third will always have the same value as passed in the *clbParam*.

Callback function's state constants table

Constant	Value	Meaning of level
PXClb_Start	1	`MaxVal` - maximum value of the level which will be passed
PXClb_Processing	2	Current progress level - any value from `0` to `MaxVal`
PXClb_Finish	3	May be any value from `0` to `MaxVal` (`MaxVal` if all passed),

may be ignored

Note: The Callback function should return `TRUE` (any non-zero value) to continue processing or `FALSE` (zero) to abort the operation.

clbParam

[in] *clbParam* specifies a user-defined callback parameter to be passed as a third parameter to the function specified by *clbFn*.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
Here is example of a simple callback function and its usage.

// This variable will store maximum value of progress indicator
// We will pass its address as clbParam, so we will be able to
// access it in our callback function.
DWORD dwMaxLevel;

BOOL __stdcall SampleCallback(DWORD state, DWORD level, LPARAM param)
{
    // Get pointer to dwMaxLevel;
    DWORD* pMaxLevel = (DWORD*)param;
    // Analise state
    switch (state)
    {
    case PXClb_Start: // start
        // store maximum value into dwMaxLevel
        *pMaxLevel = level;
        break;
    case PXClb_Processing: // processing
        // display current progress in percents
        {
            double p = 100.0 * (double)level / (double)(*pMaxLevel);
            printf("\r%.2f%%", p);
        }
        break;
    case PXClb_Finish: // finished
        // display final progress
        {
            double p = 100.0 * (double)level / (double)(*pMaxLevel);
            printf("\r%.2f%%, done.\n", p);
        }
        break;
    }
    return TRUE; // Always return TRUE to continue work
}
```

```
HRESULT SampleOfUse()
{
    // Create document
    _PXCDocument* pdf = NULL;
    HRESULT res = PXC_NewDocument(&pdf, NULL, NULL);
    if (IS_DS_FAILED(res))
        return res;
    // Set callback function and address of dwMaxLevel as parameter
    res = PXC_SetCallback(pdf, SampleCallback, (LPARAM)&dwMaxLevel);
    if (IS_DS_FAILED(res))
    {
        // Do not forget to free the pdf document!
        PXC_ReleaseDocument(pdf);
        return res;
    }
    // Some pdf generation code ommited
    ...
    // here we will write the document
    printf("Saving document:\n");
    res = PXC_WriteDocumentA(pdf, "c:\\dummy.pdf");
    // and free it
    PXC_ReleaseDocument(pdf);
    return res;
}

// If you will use this code in console application you will get following
output:

// Saving document:
// <here will be displayed the count of percents during saving>

// And after printing it should be

// Saving document:
// 100.00%, done.
```

2.1.9 PXC_SetCompression

PXC_SetCompression

Top Previous Next

PXC_SetCompression sets compression method(s) and compression level for images and text within the PDF document.

```
HRESULT  PXC_SetCompression(
    _PXCDocument* pdf,
    BOOL bText,
    BOOL bAscii,
    PXC_CompressionType cColor,
    DWORD jpegQual,
    PXC_CompressionType cIndexed,
    PXC_CompressionType cMono
);
```

Parameters

pdf
: [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

bText
: [in] if *bText* is equal to **TRUE** then text content will be compressed.

bAscii
: [in] if *bAscii* is equal to **TRUE** then text content will be written in ASCII format. Note: A larger file is also a consequence of applying this property.

cColor
: [in] *cColor* specifies compression method for color images. *(See comments for possible values)*

jpegQual
: [in] *jpegQual* specifies compression level for the `JPEG` compression method. Possible values are in the range from **1** (low quality) to **100** (high quality).

cIndexed
: [in] *cIndexed* specifies compression method for indexed images. *(See comments for possible values)*

cMono
: [in] *cMono* specifies compression method for monochrome images. *(See comments for possible values)*

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Possible values of **`PXC_CompressionType`** for **Color/Grayscale** images are:

Method	Value	Definition
`ComprType_C_NoCompress`	0	No compression.
`ComprType_C_JPEG`	0x0001	Use **JPEG** compression (DCT method).
`ComprType_C_Deflate`	0x0002	Use **Flate** (ZIP) method.
`ComprType_C_JPEG_Deflate`	**(ComprType_C_JPEG | ComprType_C_Deflate)**	Same as ComprType_C_JPEG, and then after **JPEG** compression the **Flate** method is applied to the data.
`ComprType_C_J2K`	0x0004	Use **JPEG 2000** compression.
`ComprType_C_J2K_Deflate`	**(ComprType_C_J2K | ComprType_C_Deflate)**	Same as ComprType_C_J2K, and then after **JPEG 200** compression the **Flate** method is applied to the data.
`ComprType_C_Auto`	0xFFFF	Use automatically selected method for

best compression.

Possible values for **`PXC_CompressionType`** for **Indexed** images are:

Method	Value	Definition
`ComprType_I_NoCompress`	**ComprType_C_NoCompress**	No compression.
`ComprType_I_Deflate`	**ComprType_C_Deflate**	Use **Flate** (ZIP) method.
`ComprType_I_RunLength`	0x0008	Use **RLE** method.
`ComprType_I_LZW`	0x0010	Use **LZW** compression.
`ComprType_I_Auto`	**ComprType_C_Auto**	Use automatically selected method for best compression.

Possible values of **`PXC_CompressionType`** for **Monochrome** images are:

Method	Value	Definition
`ComprType_M_NoCompress`	**ComprType_C_NoCompress**	No compression.
`ComprType_M_Deflate`	**ComprType_C_Deflate**	Use **Flate** (ZIP) method.
`ComprType_M_RunLength`	**ComprType_I_RunLength**	Use **RLE** method.
`ComprType_M_CCITT3`	0x0020	Use CCITT Fax Mode 3 method.
`ComprType_M_CCITT4`	0x0040	Use CCITT Fax Mode 4 method.
`ComprType_M_JBIG2`	0x0080	Use **JBIG2** compression.
`ComprType_M_Auto`	**ComprType_C_Auto**	Use automatically selected method for best compression.

Example (C++).

```
      _PXCDocument*        pdf;
    ...

    // Set next compression options:
    // 1. Compress text (second parameter eq. FALSE)
    // 2. Text content will not be written in ASCII format (to avoid large
file sizes)
        (third parameter eq. FALSE)
    // 3. Compress images automatically (cColor = ComprType_C_Auto)
    // 4. JPEG quality is 75 (jpegQual = 75)
    // 5. Compress indexed images automatically (cIndexed = ComprType_I_Auto)
    // 6. Compress monochrome images automatically (cMono = ComprType_M_Auto)


    HRESULT res = PXC_SetCompression(pdf, FALSE, FALSE, ComprType_C_Auto, 75,
ComprType_I_Auto, ComprType_M_Auto);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.1.10 PXC_SetDocumentInfoA

PXC_SetDocumentInfoA

Top Previous Next

PXC_SetDocumentInfoA stores additional information in the structure of the pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

```
HRESULT  PXC_SetDocumentInfoA(
    _PXCDocument* pdf,
    PXC_StdInfoField field,
    LPCSTR value
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by function **PXC_NewDocument**.

field

[in] Specifies the type of additional info. Can be one of the following values:

Value	**Definition**
InfoField_Title	Sets the **Title** field in the info structure of the pdf file.
InfoField_Subject	Sets the **Subject** field in the info structure of the pdf file.
InfoField_Author	Sets the **Author** field in the info structure of the pdf file.
InfoField_Keywords	Sets the **Keywords** field in the info structure of the pdf file.
InfoField_Creator	Sets the **Creator** field in the info structure of the pdf file.
InfoField_Producer	Sets the **Producer** field in the info structure of the pdf file.

value

[in] Pointer to a null-terminated string that specifies the value for the field *field*. This value can be NULL or an empty string.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function has the UNICODE equivalent **PXC_SetDocumentInfoW** function.

Example (C++).

```
_PXCDocument*        pdf;
    ...

    // This will be the title of the document

    LPCSTR        DocTitle = "This is the new document title";

    // Set new title:
```

```
    HRESULT res = PXC_SetDocumentInfoA(pdf, InfoField_Title, DocTitle);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.1.11 PXC_SetDocumentInfoExA

PXC_SetDocumentInfoExA

Top Previous Next

PXC_SetDocumentInfoExA stores additional information in the structure of your pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

Note that you are NOT limited to the "standard" PDF Info fields; this function allows you to create *new info keys* within the PDF document.

```
HRESULT  PXC_SetDocumentInfoExA(
    _PXCDocument* pdf,
    LPCSTR key,
    LPCSTR value
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by function **PXC_NewDocument**.

key

[in] Pointer to a null-terminated string that specifies the name of the information field in the pdf file. For example, **Title**.

value

[in] Pointer to a null-terminated string that specifies the value for the field *key*. This value can be `NULL` or empty string.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function has a UNICODE equivalent **PXC_SetDocumentInfoExW** function.

Example (C++).

```
_PXCDocument*        pdf;
    ...

    // This will be the new key and it's value
    LPCSTR          MyKeyName = "CoolTagName";
    LPCSTR          MyKeyValue = "Cool tag value";

    // Set new key

    HRESULT res = PXC_SetDocumentInfoEx(pdf, MyKeyName, MyKeyValue);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.1.12 PXC_SetDocumentInfoExW

PXC_SetDocumentInfoExW

Top Previous Next

PXC_SetDocumentInfoExW stores additional information in the structure of your pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

Note that you are NOT limited to the "standard" PDF Info fields; this function allows you to create *new info keys* within the PDF document.

```
HRESULT  PXC_SetDocumentInfoExW(
    _PXCDocument* pdf,
    LPCWSTR key,
    LPCWSTR value
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by function **PXC_NewDocument**.

key

[in] Pointer to a null-terminated string that specifies the name of the information field in the pdf file. For example, **Title**.

value

[in] Pointer to a null-terminated string that specifies the value for the field *key*. This value can be NULL or empty string.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function is the UNICODE equivalent of the PXC_SetDocumentInfoExA function.

Example (C++).

```
_PXCDocument*        pdf;
    ...

    // This will be the new key and it's value

    LPCWSTR        MyKeyName = L"CoolTagName";
    LPCWSTR        MyKeyValue = L"Cool tag value";

    // Set new key

    HRESULT res = PXC_SetDocumentInfoExW(pdf, MyKeyName, MyKeyValue);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.1.13 PXC_SetDocumentInfoW

PXC_SetDocumentInfoW

Top Previous Next

PXC_SetDocumentInfoW stores additional information in the structure of your pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

```
HRESULT  PXC_SetDocumentInfoW(
    _PXCDocument* pdf,
    PXC_StdInfoField field,
    LPCWSTR value
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by function PXC_NewDocument.

field

[in] Specifies the type of additional info. Can be any one of the following values:

Value	Definition
InfoField_Title	Sets the **Title** field in the info structure of the pdf file.
InfoField_Subject	Sets the **Subject** field in the info structure of the pdf file.
InfoField_Author	Sets the **Author** field in the info structure of the pdf file.
InfoField_Keywords	Sets the **Keywords** field in the info structure of the pdf file.
InfoField_Creator	Sets the **Creator** field in the info structure of the pdf file.
InfoField_Producer	Sets the **Producer** field in the info structure of the pdf file.

value

[in] Pointer to a null-terminated string that specifies the value for the field *field*. This value can be `NULL` or empty string.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function is the UNICODE equivalent of the function **PXC_SetDocumentInfoA**.

Example (C++).

```
_PXCDocument*        pdf;
    ...

    // This will be the title of the document

    LPCWSTR        DocTitle = L"This is the new document title";

    // Set new title:

    HRESULT res = PXC_SetDocumentInfoW(pdf, InfoField_Title, DocTitle);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.1.14 PXC_SetJBIG2Method

PXC_SetJBIG2Method

Top Previous Next

PXC_SetJBIG2Method sets compression method for JBIG2 encoded images within the PDF document.

```
HRESULT  PXC_SetJBIG2Method(
    _PXCDocument* pdf,
    DWORD nMethod
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

nMethod

[in] *nMethod* specifies the method. Posible values are:

Method	Value	Description

Standard	0	Standard method.
Crop Borders	1	This method crop 'white' borders of image than use standard JBIG2 encoding for rest of image.
Symbols	2	This method tryies to find similar blocks in the image (symbols) and encode image as a set of such blocks.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    _PXCDocument*        pdf;
...

// Set standard compression method for JBIG2 encoder

HRESULT res = PXC_SetCompression(pdf, 0);
if (IS_DS_FAILED(res))
{
    // Handle error
    ...
}
...
```

2.1.15 PXC_SetPageLayout

PXC_SetPageLayout

Top Previous Next

PXC_SetPageLayout sets the desired page layout mode.

```
HRESULT  PXC_SetPageLayout(
    _PXCDocument* pdf,
    PXC_PageLayout layout
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

layout

Specifies the page layout mode. The possible layouts are enumerated in `PXC_PageLayout`. *(See comments for possible values)*

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.

To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Possible values of `PXC_PageLayout` are:

Constant	Value	Meaning
PageLayout_SinglePage	0	Display one page at a time.
PageLayout_OneColumn	1	Display the pages in one column.
PageLayout_TwoColumns_Left	2	Display the pages in two columns, with odd numbered pages on the left.
PageLayout_TwoColumns_Right	3	Display the pages in two columns, with odd numbered pages on the right.

Example (C++).

```
	_PXCDocument*		pdf;
	...

	// Display one page at a time:

	HRESULT res = PXC_SetPageLayout(pdf, PageLayout_SinglePage);
	if (IS_DS_FAILED(res))
	{
	    // Handle error
	}
	...
```

2.1.16 PXC_SetPageMode

PXC_SetPageMode

Top Previous Next

The **PXC_SetPageMode** sets the desired page mode for the document.

```
HRESULT  PXC_SetPageMode(
    _PXCDocument* pdf,
    PXC_PageMode mode
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

mode

[in] Specifies page mode. The possible modes are enumerated in `PXC_PageMode`. *(See comments for possible values)*

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.

To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Possible values of `PXC_PageMode` are:

Constant	Value	Description
PageMode_None	0	Neither document outline nor thumbnail images visible.
PageMode_Outlines	1	Document outline visible.
PageMode_Thumbnails	2	Thumbnail images visible.
PageMode_FullScreen	3	Full-screen mode, with no menu bar, window controls, or any other window visible.

Example (C++).

```
_PXCDocument*        pdf;
...

// Let's set Thumbnail mode:

HRESULT res = PXC_SetPageMode(pdf, PageMode_Thumbnails);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.1.17 PXC_SetPermissions

PXC_SetPermissions

Top Previous Next

PXC_SetPermissions applies the specified encryption level and user's permissions to the document.

```
HRESULT  PXC_SetPermissions(
    _PXCDocument* pdf,
    DWORD enclevel,
    DWORD permFlags
);
```

Parameters

pdf
[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

enclevel
[in] Specifies the encryption level for the document. Supported values are 40 and 128.

permFlags
[in] This parameter specifies the permission flags for the document. For more information about the field values of this flag, this topic is too complex to document here and interested developers should see Adobe's comprehensive documentation for the PDF format available free from the Adobe web

site.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
_PXCDocument*        pdf;
 ...

 // Sets the encryption level to 40 bits
 // and only permit to print the document


 HRESULT res = PXC_SetPermissions(pdf, 40, Permit_Printing);
 if (IS_DS_FAILED(res))
 {
     // Handle error
 }
 ...
```

2.1.18 PXC_SetPermissions128

PXC_SetPermissions128

Top Previous Next

PXC_SetPermissions128 sets the document encryption level to 128 bits and sets the User's permissions as detailed.

This function may be used only for PDF documents set to the PDF Format V1.4 or later (see **PXC_SetSpecVersion**). When called, automatically in turn calls the **PXC_SetSpecVersion** function and applies the `PDF_VER14` parameter.

```
HRESULT  PXC_SetPermissions128(
    _PXCDocument* pdf,
    UINT bContentAccess,
    UINT bCopyExtract,
    UINT Changing,
    UINT Printing
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

bContentAccess

[in] If the value of this parameter is non-zero, content access for the visually impaired user will be enabled.

bCopyExtract

[in] If the value of this parameter is non-zero, the user has access to document content copying and extraction.

Changing

[in] This parameter specifies the access rights of the user with regards document changes and alterations. This parameter may have the following values:

Value	Description
0	Not allowed.
1	Only Document Assembly.
2	Only Form Field Fill-in or Signing.
3	Comment Authoring, Form Field Fill-in or Signing.
4	General Editing, Comment and Form Field Authoring.

Printing

[in] This parameter specifies printing permissions for the user. This parameter may have any of the following values:

Value	Description
0	Not allowed.
1	Low resolution.
2	Fully allowed.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
_PXCDocument*        pdf;
...

// Sets the encryption level to 128 bits
// and only fully allowe.to print the document


HRESULT res = PXC_SetPermissions128(pdf, 0, 0, 0, 2);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.1.19 PXC_SetPermissions256

PXC_SetPermissions256

PXC_SetPermissions256 sets the document encryption level to 256 bits and sets the User's permissions as detailed.

This function may be used only for PDF documents set to the PDF Format V1.7 or later (see **PXC_SetSpecVersion**). When called, automatically in turn calls the **PXC_SetSpecVersion** function and applies the `PDF_VER17` parameter. 256 bits encryption can be used only with psm_AES256 mode (see **PXC_EnableSecurityEx**).

```
HRESULT  PXC_SetPermissions256(
    _PXCDocument* pdf,
    UINT bContentAccess,
    UINT bCopyExtract,
    UINT Changing,
    UINT Printing
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

bContentAccess

[in] If the value of this parameter is non-zero, content access for the visually impaired user will be enabled.

bCopyExtract

[in] If the value of this parameter is non-zero, the user has access to document content copying and extraction.

Changing

[in] This parameter specifies the access rights of the user with regards document changes and alterations. This parameter may have the following values:

Value	Description
0	Not allowed.
1	Only Document Assembly.
2	Only Form Field Fill-in or Signing.
3	Comment Authoring, Form Field Fill-in or Signing.
4	General Editing, Comment and Form Field Authoring.

Printing

[in] This parameter specifies printing permissions for the user. This parameter may have any of the following values:

Value	Description
0	Not allowed.
1	Low resolution.
2	Fully allowed.

Return Values

If the function succeeds, the return value is non-negative integer.

If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
_PXCDocument*         pdf;
 ...

 // Sets the encryption level to 128 bits
 // and only fully allowe.to print the document


 HRESULT res = PXC_SetPermissions256(pdf, 0, 0, 0, 2);
 if (IS_DS_FAILED(res))
 {
     // Handle error
 }
 ...
```

2.1.20 PXC_SetPermissions40

PXC_SetPermissions40

Top Previous Next

PXC_SetPermissions40 sets the encryption level to 40 bits and sets the relevant user's permissions. This function can be used for and is compatible with the PDF document formats - versions 1.3 and 1.4 (see **PXC_SetSpecVersion**).

```
HRESULT  PXC_SetPermissions40(
    _PXCDocument* pdf,
    UINT bComments,
          UINT bCopyExtract,
    UINT bChanging,
    UINT bPrinting
);
```

Parameters

pdf
[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

bComments
[in] If the value of this parameter is zero, the user does not have permission to add or change comments and form fields in the document.

bCopyExtract
[in] If the value of this parameter is zero, the user does not have permission to copy or extract content; accessibility to this functionality will be disabled in the document.

bChanging
[in] If the value of this parameter is zero, the user does not have permission to change or edit the document in any way.

bPrinting

[in] If the value of this parameter is zero, the user user does not have permission to print the document.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
_PXCDocument*        pdf;
 ...

 // Sets the encryption level to 40 bits
 // and only permit to print the document


 HRESULT res = PXC_SetPermissions40(pdf, 0, 0, 0, 1);
 if (IS_DS_FAILED(res))
 {
     // Handle error
 }
 ...
```

2.1.21 PXC_SetSpecVersion

PXC_SetSpecVersion

Top Previous Next

PXC_SetSpecVersion sets the version of the Adobe PDF Format for the document.

```
HRESULT  PXC_SetSpecVersion(
    _PXCDocument* pdf,
    PXC_SpecVersion ver,
    BOOL bCompatMode
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

ver

[in] Specifies PDF specification version number for the created pdf file. May be any one of the following values:

Constant	Value	Description
SpecVersion13	0x13	PDF Specification version 1.3. To view such files you must use Adobe Acrobat Reader v.4.0 or higher.

`SpecVersion14`	0x14	PDF Specification version 1.4. To view such files you must use Adobe Acrobat Reader v.5.0 or higher.
`SpecVersion15`	0x15	PDF Specification version 1.5. To view such files you must use Adobe Acrobat Reader v.6.0 or higher.
`SpecVersion16`	0x16	PDF Specification version 1.6. To view such files you must use Adobe Acrobat Reader v.7.0 or higher.

bCompatMode

[in] If *ver* isn't equal to `SpecVersion15`, this parameter is ignored. Otherwise it specifies the compatibility mode: if this parameter is `TRUE`, the generated PDF file can be opened with Versions older than V6.0 of Acrobat Readers; otherwise Adobe Acrobat Reader version 6.0 or higher must be used to view such files.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
   _PXCDocument*         pdf;
    ...

    // Set specification version to 1.5 and make it compatible with the older
PDF readers

    HRESULT res = PXC_SetSpecVersion(pdf, SpecVersion15, TRUE);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.1.22 PXC_SetViewerPreferences

PXC_SetViewerPreferences

Top Previous Next

PXC_SetViewerPreferences applies the viewer preferences.

```
HRESULT  PXC_SetViewerPreferences(
    _PXCDocument* pdf,
    DWORD vprefs
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by function **PXC_NewDocument**.

vprefs

[in] Specifies the viewer preferences. This bit field is a combination of the following flags:

Constant	Value	Description
`VP_HideToolbar`	`0x0001`	A flag specifying whether to hide the viewer application's tool bars when the document is active.
`VP_HideMenubar`	`0x0002`	A flag specifying whether to hide the viewer application's menu bar when the document is active.
`VP_HideWindowUI`	`0x0004`	A flag specifying whether to hide user interface elements in the document's window (such as scroll bars and navigation controls), leaving only the document's contents displayed.
`VP_FitWindow`	`0x0008`	A flag specifying whether to resize the document's window to fit the size of the first displayed page.
`VP_CenterWindow`	`0x0010`	A flag specifying whether to position the document's window in the center of the screen.
`VP_DisplayDocTitle`	`0x0020`	A flag specifying whether the window's title bar should display the document title taken from the Title field of the document information (see **PXC_SetDocumentInfoA**). If this flag is not set, the title bar should instead display the name of the PDF file containing the document. **Note:** Has meaning only when PDF specification is 1.4 or greater.
`VP_Direction_R2L`	`0x0040`	A flag specifies the predominant reading order for text: Left-to-Right, when flag not set, or Right-to-Left (including vertical writing systems such as Chinese, Japanese, and Korean), when it is set. This flag has no direct effect on the document's contents or page numbering, but can be used to determine the relative positioning of pages when displayed side by side or printed *n*-up.

In addition, one of the following flags can be used to specify the document's *page mode*, specifying how to display the document on exiting *full-screen mode*:

Constant	Value	Description
`VP_FSPM_None`	`0x0000`	Neither document outline nor thumbnail images visible.
`VP_FSPM_Outlines`	`0x0100`	Document outline visible.
`VP_FSPM_Tumbnails`	`0x0200`	Thumbnail images visible.
`VP_FSPM_OC`	`0x0400`	Optional content group panel visible.

Note: These flags are meaningful only if the *PageMode* of the document is `PageMode_FullScreen`; it is ignored otherwise.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    _PXCDocument*           pdf;
```

```
...

DWORD        flags = 0;

// Let's construct the desired options for the document:
// 1. Hide toolbar and menubar:

flags = VP_HideToolbar | VP_HideMenubar

// 2. Show bookmarks in the left panel:

flags |= VP_FSPM_Outlines;

// Set this options:

HRESULT res = PXC_SetViewerPreferences(pdf, flags);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.1.23 PXC_SignDocumentA

PXC_SignDocumentA

Top Previous Next

PXC_SignDocumentA adds to the document a digital signature, signs and places as required on the specified page.

```
HRESULT  PXC_SignDocumentA(
    _PXCDocument* pdf,
    PCCERT_CONTEXT pCert,
    _PXCPage* page,
    LPCPXC_RectF rect,
    LPCSTR lpszReason,
    LPCSTR lpszLocation,
    LPCSTR lpszContactInfo,
    LPCSTR lpszImageFile,
    DWORD dwFlags
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

pCert

[in] *pCert* specifies the certificate context (for more information see MSDN regarding *CryptoAPI* documentation) to be used when signing the document.

page

[in] Parameter *page* specifies the identifier of page content on which the signature field should be placed.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the signature field.

lpszReason

[in] Pointer to a null-terminated string that specifies the reason for the signing, such as (`I agree...`). This parameter may be `NULL`.

lpszLocation

[in] Pointer to a null-terminated string that specifies the CPU host name or physical location of the signing. This parameter may be `NULL`.

lpszContactInfo

[in] Pointer to a null-terminated string that specifies the information provided by the signatory to enable a recipient to contact the signer to verify the signature; for example, a phone number. This parameter may be `NULL`.

lpszImageFile

[in] Specifies the full path and file name of the image which may be displayed within the signature field. This parameter may be `NULL`.

dwFlags

[in] Combination of flags which determines how the signature field should appear on the page. May be combination of the following values:

Value	Value	Description
`Sign_GR_NoGraphics`	`0x00000`	No graphics part of the signature field will be displayed.
`Sign_GR_Image`	`0x00010`	In the graphics part of the signature field the image specified by the *lpszImageFile* parameter will be displayed.
`Sign_GR_Name`	`0x00020`	In the graphics element of the signature field the signer's name will be displayed.
`Sign_TX_Name`	`0x00100`	In the text part of the signature field the signer's name will be displayed.
`Sign_TX_Date`	`0x00200`	In the text part of the signature field the date and time of signing will be displayed.
`Sign_TX_Location`	`0x00400`	In the text part of the signature field the location specified by *lpszLocation* parameter will be displayed.
`Sign_TX_Reason`	`0x00800`	In the text part of the signature field the reason of signing specified by *lpszReason* parameter will be displayed.
`Sign_TX_DName`	`0x01000`	In the text part of the signature field the detailed information about signer will be displayed.
`Sign_TX_Labels`	`0x08000`	If this flag specified, all text information (like Name, Date, etc.) will be labeled on signature field.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.

To determine if the function was successful use the defined macro's as described here: Error Handling.

Comments

This function is a ASCII equivalent of the function PXC_SignDocumentW.

Example (C++).

```
// Example shows how to sign the document before writing
// Certificate is being obtained from the system

    void SignDocument(_PXCDocument* pDoc, _PXCPage* pPage, LPCSTR
SysStorageName, LPCSTR FileName_Image)
    {
        HRESULT hr = DS_OK;
        HCERTSTORE       hCertStore = NULL;
        PCCERT_CONTEXT   pCertContext = NULL;

        // Open Storage

        hCertStore = CertOpenSystemStore(NULL, SysStorageName));

        if (!hCertStore)
        {
            // Handle error
            ...
        }

        // Obtain certificate from the system

        pCertContext = CryptUIDlgSelectCertificateFromStore(
                hCertStore,      // Open store containing the certificates to
display
                NULL,
                NULL,
                NULL,
                CRYPTUI_SELECT_LOCATION_COLUMN,
                0,
                NULL);

        if (!pCertContext)
        {
            // Handle error
            ...
        }


        // Setup rectangle

        PXC_RectF sr;
        sr.left = I2L(1);
        sr.right = I2L(4);
        sr.top = I2L(9);
        sr.bottom = I2L(8);

        // Sign the document

        hr = PXC_SignDocumentA(pDoc, pCertContext, pPage, &sr,
                    "Test Reason", "Test Location", "Test Contact Info",
FileName_Image,
                    Sign_GR_Name | Sign_TX_Name | Sign_TX_Date |
Sign_TX_Location | Sign_TX_Reason | Sign_TX_DName);

        if (IS_DS_FAILED(hr))
        {
            // Handle error
            ...
        }

        // When all processing is completed, clean up

        if(pCertContext)
        {
            CertFreeCertificateContext(pCertContext);
        }

        if(hCertStore)
        {
             if (!CertCloseStore(hCertStore,0))
             {
                     // Handle error
                     ...
             }
        }

        // done.
    }
```

2.1.24 PXC_SignDocumentBufW

PXC_SignDocumentBufW

Top Previous Next

PXC_SignDocumentBufW adds to the document a digital signature, signs and places as required on the specified page. Uses a certificate stored in PKCS#7 format within the specified memory buffer.

```
HRESULT  PXC_SignDocumentBufW(
    _PXCDocument* pdf,
    LPBYTE pPXCBuf,
    DWORD nPFXLen,
    LPCWSTR lpwszPFXPassword,
    _PXCPage* page,
    LPCPXC_RectF rect,
    LPCWSTR lpwszReason,
    LPCWSTR lpwszLocation,
    LPCWSTR lpwszContactInfo,
    LPCWSTR lpwszImageFile,
    DWORD dwFlags
);
```

Parameters

pdf
: [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

pPXCBuf
: [in] Pointer to a memory buffer where the signatories certificate (in PKCS#7 format) is stored. If there is more than one certificate stored within the specified buffer, the first located will be used.

nPFXLen
: [in] Specifies the length in bytes of the buffer addressed by the *pPXCBuf* parameter.

lpwszPFXPassword
: [in] A string password used to decrypt and verify the PFX packet from the *pPXCBuf* buffer.

page
: [in] Parameter *nPFXLen* specifies the identifier of page content on which the signature field should be placed.

rect
: [in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the signature field.

lpwszReason
: [in] Pointer to a null-terminated string that specifies the reason for the signing, such as (`I agree...`). This parameter may be `NULL`.

lpwszLocation
: [in] Pointer to a null-terminated string that specifies the CPU host name or physical location of the signing. This parameter may be `NULL`.

lpwszContactInfo
: [in] Pointer to a null-terminated string that specifies the information provided by the signatory to enable a recipient to contact the signer to verify the signature; for example, a phone number. This parameter may be `NULL`.

lpwszImageFile
: [in] Specifies the full path and file name of the image which may be displayed within the signature field. This parameter may be `NULL`.

dwFlags
: [in] Combination of flags which determines how the signature field should appear on the page. For more information about possible values, see the **PXC_SignDocumentW** function.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error**

Handling.

Example (C++).

```
// Example shows how to sign the document before writing
// using certificate stored in the buffer

   void SignDocument(_PXCDocument* pDoc, _PXCPage* pPage, LPBYTE buffer_PFX,
DWORD bufLen, LPCWSTR Password_PFX, LPCWSTR FileName_Image)
   {
       HRESULT hr = DS_OK;

       // Setup rectangle

       PXC_RectF sr;
       sr.left = I2L(1);
       sr.right = I2L(4);
       sr.top = I2L(9);
       sr.bottom = I2L(8);

       // Sign the document

       hr = PXC_SignDocumentBufW(pDoc, buffer_PFX, DWORD bufLen,
Password_PFX, pPage, &sr,
                   L"Test Reason", L"Test Location", L"Test Contact Info",
FileName_Image,
                   Sign_GR_Name | Sign_TX_Name | Sign_TX_Date |
Sign_TX_Location | Sign_TX_Reason | Sign_TX_DName);

       if (IS_DS_FAILED(hr))
       {
           // Handle error
           ...
       }

       // done.
   }
```

2.1.25 PXC_SignDocumentUsingPFXA

PXC_SignDocumentUsingPFXA

Top Previous Next

PXC_SignDocumentUsingPFXA adds to the document a digital signature, signs and places as required on the specified page. Uses a certificate stored in a PKCS#7 file.

```
HRESULT  PXC_SignDocumentUsingPFXA(
    _PXCDocument* pdf,
    LPCSTR lpszPFXFile,
    LPCSTR lpszPFXPassword,
    _PXCPage* page,
    LPCPXC_RectF rect,
    LPCSTR lpszReason,
    LPCSTR lpszLocation,
    LPCSTR lpszContactInfo,
    LPCSTR lpszImageFile,
    DWORD dwFlags
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

lpszPFXFile

[in] Pointer to a null-terminated string that specifies the full path and name of the PKCS#7 file where signatories certificate is stored. If there is more than one certificate stored within the specified file, the first located will be used.

lpszPFXPassword

[in] String password used to decrypt and verify the PFX packet from the *lpszPFXFile* file.

page

[in] Parameter *page* specifies the identifier of page content on which the signature field should be placed.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the signature field.

lpszReason

[in] Pointer to a null-terminated string that specifies the reason for the signing, such as (`I agree...`). This parameter may be `NULL`.

lpszLocation

[in] Pointer to a null-terminated string that specifies the CPU host name or physical location of the signing. This parameter may be `NULL`.

lpszContactInfo

[in] Pointer to a null-terminated string that specifies the information provided by the signatory to enable a recipient to contact the signer to verify the signature; for example, a phone number. This parameter may be `NULL`.

lpszImageFile

[in] Specifies the full path and file name of the image which may be displayed within the signature field. This parameter may be `NULL`.

dwFlags

[in] Combination of flags which determine how signature field should appear on the page. For more information about possible values, see the **PXC_SignDocumentW** function.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function is a ASCII equivalent of the function PXC_SignDocumentUsingPFXW.

Example (C++).

```
// Example shows how to sign the document before writing
    // using certificate stored in the file

    void SignDocument(_PXCDocument* pDoc, _PXCPage* pPage, LPCSTR
FileName_PFX, LPCSTR Password_PFX, LPCSTR FileName_Image)
    {
        HRESULT hr = DS_OK;

        // Setup rectangle

        PXC_RectF sr;
        sr.left = I2L(1);
        sr.right = I2L(4);
        sr.top = I2L(9);
        sr.bottom = I2L(8);

        // Sign the document

        hr = PXC_SignDocumentUsingPFXA(pDoc, FileName_PFX, Password_PFX,
pPage, &sr,
                    "Test Reason", "Test Location", "Test Contact Info",
FileName_Image,
                    Sign_GR_Name | Sign_TX_Name | Sign_TX_Date |
Sign_TX_Location | Sign_TX_Reason | Sign_TX_DName);

        if (IS_DS_FAILED(hr))
        {
            // Handle error
            ...
        }

        // done.
    }
```

2.1.26 PXC_SignDocumentUsingPFXW

PXC_SignDocumentUsingPFXW

Top Previous Next

PXC_SignDocumentUsingPFXW adds to the document a digital signature, signs and places as required on the specified page. Uses a certificate stored in a PKCS#7 file.

```
HRESULT  PXC_SignDocumentUsingPFXW(
    _PXCDocument* pdf,
    LPCWSTR lpwszPFXFile,
    LPCWSTR lpwszPFXPassword,
    _PXCPage* page,
    LPCPXC_RectF rect,
    LPCWSTR lpwszReason,
    LPCWSTR lpwszLocation,
    LPCWSTR lpwszContactInfo,
    LPCWSTR lpwszImageFile,
    DWORD dwFlags
);
```

Parameters

pdf
: [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

lpwszPFXFile
: [in] Pointer to a null-terminated string that specifies the full path and name of the PKCS#7 file where signatories certificate is stored. If there is more than one certificate stored within the specified file, the first located will be used.

lpwszPFXPassword
: [in] String password used to decrypt and verify the PFX packet from the *lpwszPFXFile* file.

page
: [in] Parameter *page* specifies the identifier of page content on which the signature field should be placed.

rect
: [in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the signature field.

lpwszReason
: [in] Pointer to a null-terminated string that specifies the reason for the signing, such as (`I agree...`). This parameter may be `NULL`.

lpwszLocation
: [in] Pointer to a null-terminated string that specifies the CPU host name or physical location of the signing. This parameter may be `NULL`.

lpwszContactInfo
: [in] Pointer to a null-terminated string that specifies the information provided by the signatory to enable a recipient to contact the signer to verify the signature; for example, a phone number. This parameter may be `NULL`.

lpwszImageFile
: [in] Specifies the full path and file name of the image which may be displayed within the signature field. This parameter may be `NULL`.

dwFlags
: [in] Combination of flags which determine how signature field should appear on the page. For more information about possible values, see the **PXC_SignDocumentW** function.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function is a UNICODE equivalent of the function **PXC_SignDocumentUsingPFXA**.

Example (C++).

```
// Example shows how to sign the document before writing
// using certificate stored in the file

   void SignDocument(_PXCDocument* pDoc, _PXCPage* pPage, LPCWSTR
FileName_PFX, LPCWSTR Password_PFX, LPCWSTR FileName_Image)
   {
       HRESULT hr = DS_OK;

       // Setup rectangle

       PXC_RectF sr;
       sr.left = I2L(1);
       sr.right = I2L(4);
       sr.top = I2L(9);
       sr.bottom = I2L(8);

       // Sign the document

       hr = PXC_SignDocumentUsingPFXW(pDoc, FileName_PFX, Password_PFX,
pPage, &sr,
                   L"Test Reason", L"Test Location", L"Test Contact Info",
FileName_Image,
                   Sign_GR_Name | Sign_TX_Name | Sign_TX_Date |
Sign_TX_Location | Sign_TX_Reason | Sign_TX_DName);

       if (IS_DS_FAILED(hr))
       {
           // Handle error
           ...
       }

       // done.
   }
```

2.1.27 PXC_SignDocumentW

PXC_SignDocumentW

Top Previous Next

PXC_SignDocumentW adds to the document digital signature, signs and places as required on the specified page.

```
HRESULT  PXC_SignDocumentW(
    _PXCDocument* pdf,
    PCCERT_CONTEXT pCert,
    _PXCPage* page,
    LPCPXC_RectF rect,
    LPCWSTR lpwszReason,
    LPCWSTR lpwszLocation,
    LPCWSTR lpwszContactInfo,
    LPCWSTR lpwszImageFile,
    DWORD dwFlags
);
```

Parameters

pdf
: [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

pCert
: [in] *pCert* specifies the certificate context (for more information see MSDN regarding CryptoAPI documentation) to be used when signing the document.

page
: [in] Parameter *page* specifies the identifier of page content on which the signature field should be placed.

rect
: [in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the signature field.

lpwszReason
: [in] Pointer to a null-terminated string that specifies the reason for the signing, such as (`I agree...`). This parameter may be `NULL`.

lpwszLocation
: [in] Pointer to a null-terminated string that specifies the CPU host name or physical location of the signing. This parameter may be `NULL`.

lpwszContactInfo
: [in] Pointer to a null-terminated string that specifies the information provided by the signatory to enable a recipient to contact the signer to verify the signature; for example, a phone number. This parameter may be `NULL`.

lpwszImageFile
: [in] Specifies the full path and file name of the image which may be displayed within the signature field. This parameter may be `NULL`.

dwFlags
: [in] Combination of flags which determines how the signature field should appear on the page. May be combination of the following values:

Value	Value	Description
`Sign_GR_NoGraphics`	`0x00000`	No graphics part of the signature field will be displayed.
`Sign_GR_Image`	`0x00010`	In the graphics part of the signature field the image specified by the *lpwszImageFile* parameter will be displayed.
`Sign_GR_Name`	`0x00020`	In the graphics element of the signature field the signer's name will be displayed.
`Sign_TX_Name`	`0x00100`	In the text part of the signature field the signer's name will be displayed.
`Sign_TX_Date`	`0x00200`	In the text part of the signature field the date and time

		of signing will be displayed.
Sign_TX_Location	0x00400	In the text part of the signature field the location specified by *lpwszLocation* parameter will be displayed.
Sign_TX_Reason	0x00800	In the text part of the signature field the reason of signing specified by *lpwszReason* parameter will be displayed.
Sign_TX_DName	0x01000	In the text part of the signature field the detailed information about signer will be displayed.
Sign_TX_Labels	0x08000	If this flag specified, all text information (like Name, Date, etc.) will be labeled on signature field.

Return Values

Comments

This function is a UNICODE equivalent of the function **PXC_SignDocumentA**.

Example (C++).

```
// Example shows how to sign the document before writing
// Certificate is being obtained from the system

    void SignDocument(_PXCDocument* pDoc, _PXCPage* pPage, LPCSTR
SysStorageName, LPCWSTR FileName_Image)
    {
        HRESULT hr = DS_OK;
        HCERTSTORE       hCertStore = NULL;
        PCCERT_CONTEXT   pCertContext = NULL;

        // Open Storage

        hCertStore = CertOpenSystemStore(NULL, SysStorageName));

        if (!hCertStore)
        {
            // Handle error
            ...
        }

        // Obtain certificate from the system

        pCertContext = CryptUIDlgSelectCertificateFromStore(
                hCertStore,      // Open store containing the certificates to
display
                NULL,
                NULL,
                NULL,
                CRYPTUI_SELECT_LOCATION_COLUMN,
                0,
                NULL);
        if (!pCertContext)
        {
            // Handle error
            ...
        }


        // Setup rectangle

        PXC_RectF sr;
        sr.left = I2L(1);
        sr.right = I2L(4);
        sr.top = I2L(9);
        sr.bottom = I2L(8);

        // Sign the document

        hr = PXC_SignDocumentW(pDoc, pCertContext, pPage, &sr,
                    L"Test Reason", L"Test Location", L"Test Contact Info",
FileName_Image,
                    Sign_GR_Name | Sign_TX_Name | Sign_TX_Date |
Sign_TX_Location | Sign_TX_Reason | Sign_TX_DName);

        if (IS_DS_FAILED(hr))
        {
            // Handle error
            ...
        }

        // When all processing is completed, clean up

        if(pCertContext)
        {
            CertFreeCertificateContext(pCertContext);
        }

        if(hCertStore)
        {
             if (!CertCloseStore(hCertStore,0))
             {
                     // Handle error
                     ...
             }
        }

        // done.
    }
```

2.1.28 PXC_WriteDocumentA

PXC_WriteDocumentA

Top Previous Next

PXC_WriteDocumentA writes the generated PDF document to a file.

There is an extended version of this function (**PXC_WriteDocumentExA**), which allows additional parameters to be applied: PDF File and path settings, run an application once the file is generated (i.e. to view the saved file or perform additional processing of some kind etc.)

```
HRESULT  PXC_WriteDocumentA(
    _PXCDocument* pdf,
    LPCSTR fName
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

fName

[in] Pointer to a null-terminated string which specifies the full path and name of the file, where the PDF object will be stored.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

A `UNICODE` version of this function **PXC_WriteDocumentW** is also implemented in the **PDF-XChange Library** .

Remarks

Note: Any action performed on the document (new page creation, image additions etc) after saving may result in unpredictable behaviour and results. After saving the PDF object it is recommended to call the **PXC_ReleaseDocument** function.

Example (C++).

```
// Example shows how to save document into file

    _PXCDocument*        pdf;

    ...

    // File name
    LPCSTR        fName = "C:\\MyPdfFileName.pdf";
    // Save file

    HRESULT hr = PXC_WriteDocumentA(pdf, fName);

    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // document is written now
```

2.1.29 PXC_WriteDocumentExA

PXC_WriteDocumentExA

Top Previous Next

PXC_WriteDocumentExA function saves the generated pdf object to a file, and, if enabled, runs the specified application with the name of the saved file as a parameter.

```
HRESULT  PXC_WriteDocumentExA(
    _PXCDocument* pdf,
    LPSTR lpszFileName,
    DWORD dwFileNameLen,
    DWORD dwFlags,
    LPCSTR appname
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

lpszFileName

[in, out] Pointer to the buffer that specifies the full path and name of the file, where the PDF object will be stored and designated to receive this name (depends on *dwFlags*).

dwFileNameLen

[in] Specifies the size of the *lpszFileName* buffer. May be `0`, if *dwFlags* does not hold the `WEF_ShowSaveDialog` flag value.

dwFlags

[in] Defines actions to be performed. May be a combination of the following flags:

Flag	Value	Description
WEF_ShowSaveDialog	0x0001	If this flag is specified, **PDF-XChange Library** will display the **Save As** dialog. This dialog uses the path and name

		from the *lpszFileName* parameter as a default. The filename, chosen by the user, will be stored in the buffer *lpszFileName*.
`WEF_NoWrite`	`0x0002`	If this flag is specified, **PDF-XChange Library** will not save the input. May be useful where the users desired filename is required in the **Save As** dialog without actually saving the PDF file itself.
`WEF_AskForOverwrite`	`0x0004`	If this flag is specified the **Save As** dialog will ask for confirmation before overwriting a file, if the file name exists in the specified path already.
`WEF_RunApp`	`0x0008`	If this flag is specified, the application specified by the *appname* parameter will be executed after file generation. If *appname* is `NULL` or an empty string, the default application for the file type *lpszFileName* will be called (as specified with the Windows 'associate with' parameter for the file's extension type).

appname

[in] Pointer to a null-terminated string which specifies the full path for the application that will be run for the file type *lpszFileName*, if the flag `WEF_RunApp` is specified in *dwFlags*.
This value may be `NULL` or an empty string.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Note: Any action performed on the document (new page creation, image additions etc) after saving may result in unpredictable behaviour and results. After saving the PDF object it is recommended to call the **PXC_ReleaseDocument** function.

Comments

This function has a UNICODE equivalent function **PXC_WriteDocumentExW**.

Example (C++).

```
// Example shows how to save document into file
// using 'Save As' dialog and after that
// run defaut PDF viwer to show the document

    _PXCDocument*        pdf;

    ...

    // Buffer for file name that will be returned
    LPSTR        FileName[MAX_PATH];
    DWORD        FileNameLength = MAX_PATH;

    FileName[0] = '\0';

    // Flags: Show dialog + run default application

    DWORD Flag = WEF_ShowSaveDialog | WEF_RunApp;

    // Save file

    HRESULT hr = PXC_WriteDocumentExA(pdf, FileName, FileNameLength, Flag,
NULL);

    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // document is written now
    // File name choosen is stored in 'FileName'
```

2.1.30 PXC_WriteDocumentExW

PXC_WriteDocumentExW

Top Previous Next

PXC_WriteDocumentExW saves the generated pdf object to a file, and, if enabled, runs the specified application with the name of the saved file as a parameter.

```
HRESULT  PXC_WriteDocumentExW(
    _PXCDocument* pdf,
    LPWSTR lpwszFileName,
    DWORD dwFileNameLen,
    DWORD dwFlags,
    LPCWSTR appname
);
```

Parameters

pdf

[in] Specifies the PDF object previously created by the function **PXC_NewDocument**.

lpwszFileName

[in, out] Pointer to the buffer that specifies the full path name of the file, where the PDF object will be

stored, and which will receive this name (dependant on *dwFlags*).

dwFileNameLen

[in] Specifies the size of the *lpwszFileName* buffer. May be `0`, if *dwFlags* does not contain `WEF_ShowSaveDialog` flag.

dwFlags

[in] Defines actions to be performed by this function. May be a combination of the following flags:

Flag	Value	Description
`WEF_ShowSaveDialog`	`0x0001`	If this flag is specified, **PDF-XChange Library** will show a **Save As** dialog. This dialog uses the path and name from the *lpwszFileName* parameter as a default. The filename, chosen by the user, will be stored in the buffer *lpwszFileName*.
`WEF_NoWrite`	`0x0002`	If this flag is specified, **PDF-XChange Library** will not be saved. May be useful where it is required only to know the filename, chosen by the user in the **Save As** dialog without saving the PDF file itself.
`WEF_AskForOverwrite`	`0x0004`	If this flag is specified, the **Save As** dialog will ask before overwriting when the user tries to select an existing PDF file.
`WEF_RunApp`	`0x0008`	If this flag is specified, the application specified by the *appname* parameter will be executed. If *appname* is `NULL` or an empty string, the defaut application for the file *lpwszFileName* will be called (as specified with the Windows 'associate with' parameter for file's extention).

appname

[in] Pointer to a null-terminated string which specifies the full path for the application that will be run for the file *lpwszFileName*, if flag `WEF_RunApp` specified into *dwFlags*.
This value can be `NULL` or an empty string.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Note: Any action performed on the document (new page creation, image additions etc) after saving may result in unpredictable behaviour and results. After saving the PDF object it is recommended to call the **PXC_ReleaseDocument** function.

Comments

This function is the UNICODE equivalent of the function PXC_WriteDocumentExA.

Example (C++).

```
// Example shows how to save document into file
// using 'Save As' dialog and after that
// run defaut PDF viwer to show the document

    _PXCDocument*        pdf;

    ...

    // Buffer for file name that will be returned

    LPWSTR        FileName[MAX_PATH];
    DWORD        FileNameLength = MAX_PATH;

    FileName[0] = L'\0';

    // Flags: Show dialog + run default application

    DWORD Flag = WEF_ShowSaveDialog | WEF_RunApp;

    // Save file

    HRESULT hr = PXC_WriteDocumentExW(pdf, FileName, FileNameLength, Flag,
NULL);

    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // document is written now
    // File name choosen is stored in 'FileName'
```

2.1.31 PXC_WriteDocumentToIStream

PXC_WriteDocumentToIStream

Top Previous Next

PXC_WriteDocumentToIStream writes the generated PDF document to a IStream object.

```
HRESULT  PXC_WriteDocumentToIStream(
    _PXCDocument* pdf,
    IStream* stream
);
```

Parameters

pdf

[in] Specifies the PDF object previously created by the function PXC_NewDocument.

stream

[in] Pointer to a `IStream` object.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an error code.
To determine if the function was successful use the defined macro's as described here: Error Handling.

Remarks

Note: Any action performed on the document (new page creation, image additions etc) after saving may result in unpredictable behaviour and results. After saving the PDF object it is recommended to call the PXC_ReleaseDocument function.

Example (C++).

```
// Example shows how to save document into IStream object.

    _PXCDocument*        pdf;

    ...

    // Create IStream object

    LPSTREAM iStream = NULL;
    CreateStreamOnHGlobal(NULL, TRUE, &iStream);

    // Write file

    HRESULT hr = PXC_WriteDocumentToIStream(pdf, iStream);

    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // document is written now to 'iStream'

    ...
```

2.1.32 PXC_WriteDocumentW

PXC_WriteDocumentW

PXC_WriteDocumentW writes the generated PDF document to a file.

There is an extended version of this function (**PXC_WriteDocumentExW**), which allows additional parameters to be applied: PDF File and path settings, run an application once the file is generated (i.e. to view the saved file or perform additional processing of some kind etc.)

```
HRESULT  PXC_WriteDocumentW(
    _PXCDocument* pdf,
    LPCWSTR fName
);
```

Parameters

pdf

[in] Specifies the PDF object previously created by the function **PXC_NewDocument**.

fName

[in] Pointer to a null-terminated string which specifies the full path and name of the file, where the PDF object will be stored.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

An `ANSI` version of this function **PXC_WriteDocumentA** is also implemented see: **PDF-XChange Library** .

Remarks

Note: Any action performed on the document (new page creation, image additions etc) after saving may result in unpredictable behaviour and results. After saving the PDF object it is recommended to call the **PXC_ReleaseDocument** function.

Example (C++).

```
// Example shows how to save document into file

    _PXCDocument*        pdf;

    ...

    // File name
    LPCWSTR         fName = L"C:\\MyPdfFileName.pdf";

    // Save file
```

```
HRESULT hr = PXC_WriteDocumentW(pdf, fName);

if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

// document is written now
```

2.2 Page Operations

Page Operations

Top Previous Next

These functions affect Page-level properties:

- PXC_AddPage
- PXC_EndPage
- PXC_GetPage
- PXC_GetPageBox
- PXC_GetPageIndex
- PXC_GetPageRotation
- PXC_GetPagesCount
- PXC_GetPageSize
- PXC_InsertPage
- PXC_RemovePage
- PXC_SetPageBox
- PXC_SetPageDuration
- PXC_SetPageRotation
- PXC_SetPageTransition

2.2.1 PXC_AddPage

PXC_AddPage

Top Previous Next

PXC_AddPage adds a new page to the end of the page list in the PDF object.

```
HRESULT  PXC_AddPage(
    _PXCDocument* pdf,
    double width,
    double height,
    _PXCPage** page
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

width

[in] *width* specifies the width of the page in points.

height

[in] *height* specifies the height of the page in points.

page

[in, out] Pointer to a variable of the `_PXCPage*` type, that specifies the identifier of the created page.

Return Values

If the function succeeds, the return value is `PXC_OK`, and the variable pointer *page* will contain the valid identifier of the new page.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

There are maximum and minimum allowed page sizes. The minimum allowed page size is 3 by 3 points (approximately 0.04 by 0.04 inch); the maximum is 14,400 by 14,400 points (200 by 200 inches).

Note:

Adobe Acrobat viewers prior to version 4.0, have a minimum allowed page size of 72 by 72 points (1 by 1 inch) and a maximum page size of 3240 by 3240 (45 by 45 inches).

Example (C++).

```
_PXCDocument*        pdf;

    ...

    // Add 'A4' page to the document

    _PXCPage*                 page = NULL;

    HRESULT res = PXC_AddPage(pdf, I2L(8.5), I2L(11), &page);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.2.2 PXC_EndPage

PXC_EndPage

Top Previous Next

PXC_EndPage optimizes memory used by the specified page object. This function is useful in minimizing resource use when a document contains a number of small pages - increasing demands on system resources. After calling this function the specified page can be used for any operation.

```
HRESULT  PXC_EndPage(
    _PXCPage* page
);
```

Parameters

page

[in] Parameter *page* specifies the document page previously created by the **PXC_AddPage** or **PXC_InsertPage**.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to optimize memory usage
// while working with the page object

    _PXCPage* page;

    ...

    PXC_EndPage(page);

    ...
```

2.2.3 PXC_GetPage

PXC_GetPage

Top Previous Next

PXC_GetPage returns the identifier of the page, based on its index value (position within the document.)

N.B. PDF Files are 'Zero' based for page numbering, for example the first viewed page is actually page '0'.

```
HRESULT PXC_GetPage(
    _PXCDocument* pdf,
    LONG index,
    _PXCPage** page
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

index

[in] *index* specifies zero based page index.

page

[out] *page* specifies a pointer to the variable which contains the identifier of the page.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    _PXCDocument* pdf;

...

// Retrieve the first page identifier

_PXCPage*                page = NULL;

HRESULT res = PXC_GetPage(pdf, 0, &page);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.2.4 PXC_GetPageBox

PXC_GetPageBox

Top Previous Next

PXC_GetPageBox retrieves the specified pages boundaries rectangle. For more information regarding page boundaries see **PXC_SetPageBox** function.

```
HRESULT  PXC_GetPageBox(
    const _PXCPage* page,
    PXC_PageBox pBoxID,
    LPPXC_RectF rect
);
```

Parameters

page

[in] Parameter *page* specifies the document page previously created by the **PXC_AddPage** or **PXC_InsertPage**.

pBoxID

[in] *pBoxID* specifies the ID of the page's rectangle, and may be any one of the following values.

Constant	Value	Definition
PB_MediaBox	0	Specifies *media box* to be retrieved.
PB_CropBox	1	Specifies *crop box* to be retrieved.
PB_BleedBox	2	Specifies *bleed box* to be retrieved.

`PB_TrimBox`	3	Specifies *trim box* to be retrieved.
`PB_ArtBox`	4	Specifies *art box* to be retrieved.

rect

[in] Pointer to **PXC_RectF** structure that receives the coordinates of the specified page's box.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
_PXCDocument*         pdf;
    ...
    PXC_RectF                 MediaRect;

    // Get 'MediaBox' of the page

    HRESULT res = PXC_GetPageBox(pdf, PB_MediaBox, &MediaRect);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.2.5 PXC_GetPageIndex

PXC_GetPageIndex

Top Previous Next

PXC_GetPageIndex returns the page's index value (position) within the document, based on its identifier. Note PDF files are 'Zero based' for page numbering.

```
HRESULT  PXC_GetPageIndex(
    _PXCDocument* pdf,
    _PXCPage* page,
    DWORD* index
);
```

Parameters

pdf

[in] *_PXCDocument** specifies the PDF object previously created by function **PXC_NewDocument**.

page

[in] *_PXCPage** specifies the identifier of the page.

index

[out] *index* specifies a pointer to a DWORD variable to receive the page index.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    _PXCDocument* pdf;
_PXCPage* page;
...

// Retrive the index of the specified page

DWORD PageIndex = 0;

HRESULT res = PXC_GetPageIndex(pdf, page, &PageIndex);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.2.6 PXC_GetPageRotation

PXC_GetPageRotation

Top Previous Next

PXC_GetPageRotation retrieves the specified page's rotation angle.

```
HRESULT  PXC_GetPageRotation(
    const _PXCPage* page,
    LONG* angle
);
```

Parameters

page

[in] Parameter *page* specifies the document page previously created by the **PXC_AddPage** or **PXC_InsertPage**.

angle

[in] Specifies the pointer to the variable which receives the rotation angle of the page.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    _PXCPage* page;
```

```
...

// Retrive current page rotation angle

LONG angle = 0;

HRESULT res = PXC_GetPageRotation(page, &angle);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.2.7 PXC_GetPagesCount

PXC_GetPagesCount

Top Previous Next

PXC_GetPagesCount returns a count of the pages in the PDF object.

```
HRESULT  PXC_GetPagesCount(
    _PXCDocument* pdf,
    UINT* count
);
```

Parameters

pdf

[in] *_PXCDocument** specifies the PDF object previously created by the function **PXC_NewDocument**.

count

[out] *count* specifies a pointer to the DWORD variable to receive the page count.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
_PXCDocument* pdf;
...
```

```
// Retrieve number of pages in the document

UINT PageCount = 0;

HRESULT res = PXC_GetPagesCount(pdf, &PageCount);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.2.8 PXC_GetPageSize

PXC_GetPageSize

Top Previous Next

PXC_GetPageSize retrieves the selected page's dimensions.

```
HRESULT  PXC_GetPageSize(
    const _PXCPage* page,
    double* width,
    double* height
);
```

Parameters

page

[in] Parameter *page* specifies the document page previously created by the **PXC_AddPage** or **PXC_InsertPage**.

width

[in] Specifies the pointer to the variable which recieves the page's width in points.

height

[in] Specifies the pointer to the variable which recieves the page's heighs in points.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    _PXCPage*         page;
...

// Retrieve page size

double width = 0.0;
double height = 0.0;
```

```
    HRESULT res = PXC_GetPageSize(page, &width, &height);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.2.9 PXC_InsertPage

PXC_InsertPage

Top Previous Next

PXC_InsertPage inserts a new page into specified PDF object and this will be added to the specified position in the page list of the PDF object, please note that PDF files are 'ZERO' page based.

```
HRESULT  PXC_InsertPage(
    _PXCDocument* pdf,
    LONG index,
    double width,
    double height,
    _PXCPage** page
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

index

[in] Parameter*index* specifies the position of the new page insertion. If *index* is equal to `-1` the new page will be added to the end of the page list.

width

[in] Parameter *width* defines the width of the page in points.

height

[in] Parameter *height* defines the height of the page in points.

page

[in, out] Pointer to the variable of the `_PXCPage*` type, containing the identifier for the created page.

Return Values

If the function succeeds, the return value is `PXC_OK`, and variable pointed by *page* will contain valid identifier of the new page.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
        _PXCDocument* pdf;

    ...
    // insert 'A4' page to the document at the begining
    // (the new page will be first one)

    _PXCPage*                page = NULL;

    HRESULT res = PXC_InsertPage(pdf, 0, I2L(8.5), I2L(11), &page);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.2.10 PXC_RemovePage

PXC_RemovePage

Top Previous Next

PXC_RemovePage removes the specified page from the current PDF document.

```
HRESULT  PXC_RemovePage(
    _PXCDocument* pdf,
    LONG index
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by function **PXC_NewDocument**.

index

[in] The Specified zero-based page number which must be removed.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

If the *index* removed is the current page number, or is the only page in the document, the current page number will be reset to -1 (no current page).

Example (C++).

```
        _PXCDocument* pdf;

    ...

    // Remove the first page from the document
```

```
HRESULT res = PXC_RemovePage(pdf, 0);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.2.11 PXC_SetPageBox

PXC_SetPageBox

Top Previous Next

PXC_SetPageBox allows the specification of different bounding rectangles for a PDF page.

For more detail see **Comments** below.

```
HRESULT  PXC_SetPageBox(
    _PXCPage* page,
    PXC_PageBox pBoxID,
    LPCPXC_RectF rect
);
```

Parameters

page

[in] Parameter *page* specifies the document page previously created by the **PXC_AddPage** or **PXC_InsertPage**.

pBoxID

[in] *pBoxID* specifies the ID of the page's rectangle, and may be any one of the following values.

Constant	Value	Meaning
PB_CropBox	1	Specifies *crop box* to be set.
PB_BleedBox	2	Specifies *bleed box* to be set.
PB_TrimBox	3	Specifies *trim box* to be set.
PB_ArtBox	4	Specifies *art box* to be set.

Note: Also there is a constant `PB_MediaBox` (value `0`), however this constant may not be used in conjunction with this function, because the page's *media box* is read-only, and is specified during page creation by the width and height of the page.

rect

[in] Pointer to a **PXC_RectF** structure that contains the coordinates of the specified page's box. If this parameter is `NULL`, the specified rectangle will be set to its default value.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

A PDF page may be prepared either for a finished medium, such as a sheet of paper, or as part of a prepress process in which the content of the page is placed on an intermediate medium, such as film or an imposed reproduction plate. In the latter case, it is important to distinguish between the intermediate page and the finished page. The intermediate page may often include additional production-related content, such as bleeds or printer marks, that falls outside the boundaries of the finished page. To handle such cases, a PDF page can define as many as five separate boundaries to control various aspects of the imaging process:

- The *media box* defines the boundaries of the physical medium on which the page is to be printed. It may include any extended area surrounding the finished page for bleed, printing marks, or other such purposes. It may also include areas close to the edges of the medium that cannot be marked because of physical limitations of the output device. Content falling outside this boundary can safely be discarded without affecting the meaning of the PDF file.
- The *crop box* defines the region to which the contents of the page are to be clipped (cropped) when displayed or printed. Unlike the other boxes, the crop box has no defined meaning in terms of physical page geometry or intended use; it merely imposes clipping on the page contents. However, in the absence of additional information, the crop box will determine how the page’s contents are to be positioned on the output medium. The default value is the page’s media box.
- The *bleed box* (PDF Spec >= 1.3) defines the region to which the contents of the page should be clipped when output in a production environment. This may include any extra “bleed area” needed to accommodate the physical limitations of cutting, folding, and trimming equipment. The actual printed page may include printing marks that fall outside the bleed box. The default value is the page’s crop box.
- The *trim box* (PDF Spec >= 1.3) defines the intended dimensions of the finished page after trimming. It may be smaller than the media box, to allow for productionrelated content such as printing instructions, cut marks, or color bars. The default value is the page’s crop box.
- The *art box* (PDF Spec >= 1.3) defines the extent of the page’s meaningful content (including potential white space) as intended by the page’s creator. The default value is the page’s crop box.

Example (C++).

```
// Example shows how to 'clip' the half of the page
// by setting CropBox of the half size of the media one
// It is assumed that the page has no CropBox set to it originally

    _PXCDocument*        pdf;
    ...
    PXC_RectF                PageRect;

    // Get 'MediaBox' of the page

    HRESULT res = PXC_GetPageBox(pdf, PB_MediaBox, &PageRect);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }

    double width  = PageRect.right - PageRect.left;
    PageRect.right = PageRect.left + width / 2;

    // Now set this rect as 'CropBox'

    res = PXC_SetPageBox(pdf, PB_CropBox, &PageRect);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }

    ...
```

2.2.12 PXC_SetPageDuration

PXC_SetPageDuration

Top Previous Next

The **PXC_SetPageDuration** function sets the current page's display duration: the maximum amount of time that the page will be displayed before the presentation automatically advances to the next page.

```
HRESULT  PXC_SetPageDuration(
    _PXCPage* page,
    DWORD miliseconds
);
```

Parameters

page
: [in] Parameter *page* specifies the document page previously created by the **PXC_AddPage** or **PXC_InsertPage**.

miliseconds
: [in] Specifies the maximum amount of time, in milliseconds, that the current page will be displayed. If this value is 0 (the default) the page does not advance automatically.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
   _PXCPage*        page;
    ...

    // Set display durration to 2 seconds

    HRESULT res = PXC_SetPageDuration(page, 2000);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.2.13 PXC_SetPageRotation

PXC_SetPageRotation

PXC_SetPageRotation rotates page by specified angle.

```
HRESULT  PXC_SetPageRotation(
    _PXCPage* page,
    LONG angle
);
```

Parameters

page

[in] Parameter *page* specifies the document page previously created by the **PXC_AddPage** or **PXC_InsertPage**.

angle

[in] Specifies rotation angle. Possible values are: 0, 90, 180, 270, -90, -180, -270.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

If *angle* is positive then page is rotated counterclockwise, otherwise it is rotated clockwise.

Example (C++).

```
   _PXCPage*        page;
    ...

    // Rotate page by 90 degree clockwise

    HRESULT res = PXC_SetPageRotation(page, -90);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.2.14 PXC_SetPageTransition

PXC_SetPageTransition

PXC_SetPageTransition sets the style and duration of the visual transition to use when moving from one page to another (the new page dictates the transition type) during presentation.

```
HRESULT  PXC_SetPageTransition(
    _PXCPage* page,
    PXC_TransitionStyle style,
    DWORD miliseconds,
    DWORD v1,
    DWORD v2
);
```

Parameters

page

[in] Parameter *page* specifies the document page previously created by the **PXC_AddPage** or **PXC_InsertPage**.

style

[in] Specifies the transition style to use when moving to the current page from another during a presentation. The possible styles are enumerated in `PXC_TransitionStyle`. *(See comments for possible values)*

miliseconds

[in] Specifies the duration of the transition effect, in milliseconds.

v1

[in] Additional parameter. See **Comments** for more info.

v2

[in] Additional parameter. See **Comments** for more info.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Possible values of `PXC_TransitionStyle` are:

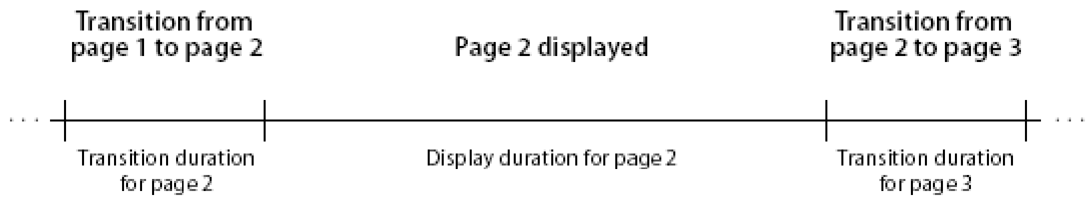
Value	Description		
TransitionStyle_Replace	The new page simply replaces the old one with no special transition effect. Parameters *miliseconds*, *v1*, and *v2* are ignored. This is the default value.		
TransitionStyle_Split	Two lines sweep across the screen, revealing the new page. The lines may be either horizontal or vertical and may move inward from the edges of the page or outward from the center, as specified by the *v1* and *v2* parameters, respectively.		
	v1	The dimension for which the transition effect should occur. Select from the following values:	
		`Transition_Dim_Horizontal`	Horizontal.
		`Transition_Dim_Vertical`	Vertical.
	v2	TThe direction of motion for the transition effect. Select from the following values:	
		`Transition_Moti`	Inward from the

		`on_In`	edges of the page.
		`Transition_Motion_Out`	Outward from the center of the page.
`TransitionStyle_Blinds`	Multiple lines, evenly spaced across the screen, synchronously sweeping in the same direction to reveal the new page. The lines may be either horizontal or vertical, as specified by the *v1* parameter. Horizontal lines move downward, vertical lines to the right. *v2* is ignored.		
	v1	The dimension for which the transition effect should occur. Select from the following values:	
		`Transition_Dim_Horizontal`	Horizontal.
		`Transition_Dim_Vertical`	Vertical.
`TransitionStyle_Box`	A rectangular box sweeps inward from the edges of the page or outward from the center, as specified by the *v1* parameter, revealing the new page. *v2* is ignored.		
	v1	The direction of motion for the transition effect. Select from the following values:	
		`Transition_Motion_In`	Inward from the edges of the page.
		`Transition_Motion_Out`	Outward from the center of the page.
`TransitionStyle_Wipe`	A single line sweeps across the screen from one edge to the other in the direction specified by the *v1* parameter, revealing the new page. *v2* is ignored.		
	v1	The direction in which the specified transition effect moves, expressed in degrees counterclockwise starting from a left-to-right direction. Can be one of the following values:	
		`Transition_WDir_LeftToRight`	Left to right.
		`Transition_WDir_BottomToTop`	Bottom to top.
		`Transition_WDir_RightToLeft`	Right to left.
		`Transition_WDir_TopToBottom`	Top to bottom.
`TransitionStyle_Dissolve`	The old page "dissolves" gradually to reveal the new one. Parameters *v1* and *v2* are ignored.		
`TransitionStyle_Glitter`	Similar to `TransitionStyle_Dissolve`, however the effect sweeps across the page in a wide band moving from one side of the screen to the other in the direction specified by the *v1* parameter. *v2* is ignored.		
	v1	The direction in which the specified transition effect moves, expressed in	

degrees counterclockwise starting from a left-to-right direction. Can be one of the following values:

`Transition_GDir_LeftToRight`	Left to right.
`Transition_GDir_TopToBottom`	Top to bottom.
`Transition_GDir_TopLeftToBottomRight`	Top-left to bottom-right.

The figure below illustrates the relationship between the *transition duration* and *display duration*. Note that the transition duration specified for a page (page 2 in the figure) governs the transition to that page from another page; the transition from the page is governed by the next page's transition duration:


Transition from page 1 to page 2
Page 2 displayed
Transition from page 2 to page 3
Transition duration for page 2
Display duration for page 2
Transition duration for page 3

Example (C++).

```
      _PXCPage*         page;
    ...

    // Set options for visual transition during presentation
    // The old page will "dissolves" gradually to reveal the new one.
    // Process durration will be set to 2 seconds

    HRESULT res = PXC_SetPageTransition(page, TransitionStyle_Dissolve, 2000,
0, 0);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.3 Content Creation

2.3.1 Coordinate System

Coordinate System

Top Previous Next

Coordinate systems define the “canvas” on which all painting occurs. They determine the position, orientation, and size of the text, graphics, and images that appear on a page.

Paths and positions are defined in terms of pairs of *coordinates* on the Cartesian plane. A coordinate pair is a pair of real numbers *x* and *y* that locate a point horizontally and vertically within a two-dimensional *coordinate space*. A coordinate space is determined by the following properties in respect to the current page:

- The location of the origin
- The orientation of the *x* and *y* axis
- The length of the units a long each axis

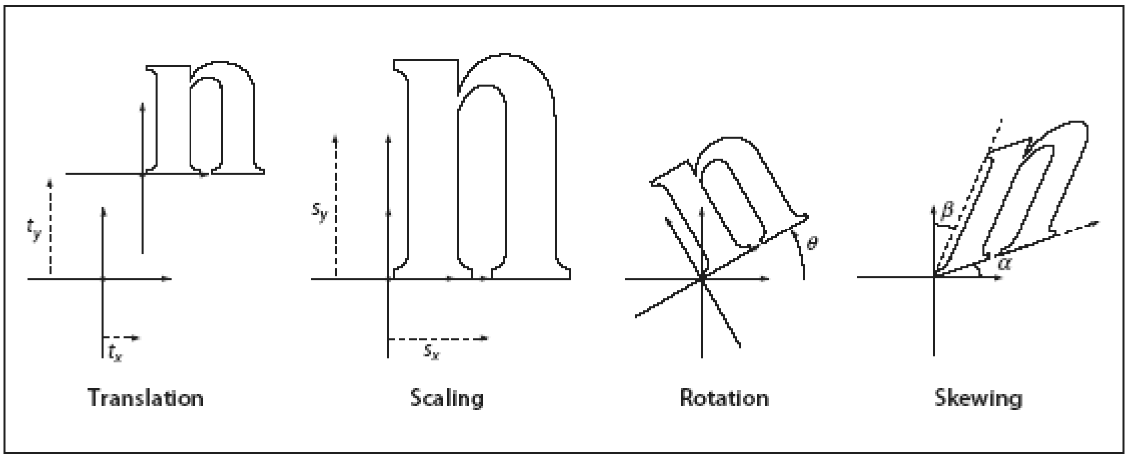
Transformations among coordinate spaces are defined by *transformation matrices*, which can specify any linear mapping of two-dimensional coordinates, including translation, scaling, rotation, reflection, and skewing.

The **PDF-XChange Library** uses a PDF's coordinate system. The coordinate system is initialized to a default state for each page of a document. The positive *x* axis extends horizontally to the right and the positive *y* axis vertically upward, as in standard mathematical practice. The length of a unit along both the *x* and *y* axis is equal to 1/72 inch (point).

If necessary, coordinate space can be be modified by specifying transformation matrix. For example, content originally composed to occupy an entire page can be incorporated without change as an element of another page by shrinking the coordinate system within which it is drawn.

A *transformation matrix* specifies the relationship between two coordinate spaces. By modifying a transformation matrix, objects can be scaled, rotated, translated, or transformed in other ways.

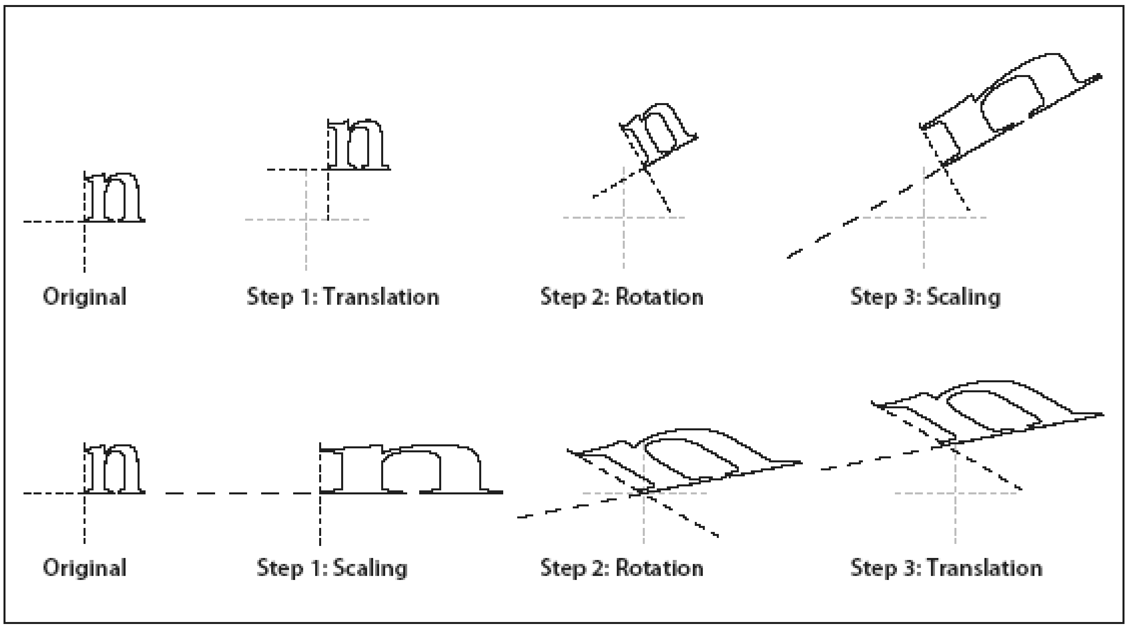
The Figure below shows examples of each common transformation (translation, scaling, rotation, skewing).



If several transformations are combined, the order in which they are applied is significant. For example, first scaling and then translating the *x* axis is not the same as first translating and then scaling it. In general, to obtain the expected results, transformations should be performed in the following order:

1. Translate
2. Rotate
3. Scale or skew

The figure below shows the effect of the order in which transformations are applied. The figure shows two sequences of transformations applied to a coordinate system. After each successive transformation, an outline of the letter `n` is drawn.



The transformations shown in the figure are as follows:

- A translation of 10 units in the x direction and 20 units in the y direction
- A rotation of 30 degrees
- A scaling by a factor of 3 in the x direction

In the figure, the axis are shown with a dash pattern having a 2-unit dash and a 2-unit gap. In addition, the original (un-transformed) axis are shown in a lighter color for reference. Notice that the scale-rotate-translate ordering, results in a distortion of the coordinate system, leaving the *x* and *y* axis no longer perpendicular, while the recommended translate-rotate-scale ordering does not have this adverse effect.

2.3.1.1 PXC_CS_Concat

PXC_CS_Concat

Top Previous Next

PXC_CS_Concat function sets a two-dimensional linear transformation of the coordinate system. This transformation can be used to scale, rotate, skew, or translate path drawing.

```
HRESULT  PXC_CS_Concat(
    _PXCContent* content,
    LPCPXC_Matrix m
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

m

[in] Pointer to a **PXC_Matrix** structure that contains the transformation data.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

For any coordinates (`x`, `y`) the transformed coordinates (`x'`, `y'`) may be determined by the following algorithm:

```
x' = x * m11 + y * m21 + Dx,
y' = x * m12 + y * m22 + Dy,
```

where the transformation matrix is represented by the following:

```
| m11 m12 0 |

| m21 m22 0 |

| Dx  Dy  1 |
```

The transformation is usually used to scale or rotate the coordinate system.

The default transformation is the identity matrix with a zero offset.

Remarks

Note that sequential calls to **PXC_CS_Concat** are additive. To restore the previous state of the coordinate system, you must save its state, and then restore it.

Example (C++).

```
// Example shows how rotate coordinate system by specified angle

   // Mathematic functions are required (for sin and cos calculations)
   #include <math.h>

   // Define 'Pi' constant if not defined
   #ifndef M_PI
       #define M_PI       3.14159265358979323846
   #endif

   // Rotate text matrix by 'angle'

   HRESULT Rotate_CS(_PXCPage* page, double angle)
   {
       double a, sina, cosa;
       PXC_Matrix CSMatrix;

       // Some intermediate calculations

       a = angle * M_PI / 180.0;
       sina = sin(a);
       cosa = cos(a);
```

```
        // Set matrix elements

        CSMatrix.a = cosa;
        CSMatrix.b = sina;
        CSMatrix.c = -sina;
        CSMatrix.d = cosa;
        CSMatrix.e = 0.0;
        CSMatrix.f = 0.0;

        // Do transformation

        return PXC_CS_Concat((_PXCContent*)page, &CSMatrix);
    }
```

2.3.1.2 PXC_CS_Get

PXC_CS_Get

Top Previous Next

PXC_CS_Get returns the current matrix of the coordinate system transformation.

```
HRESULT  PXC_CS_Get(
    const _PXCContent* content,
    LPPXC_Matrix m
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

m

[out] Pointer to the **PXC_Matrix** type structure that contains the current matrix of the page coordinate system transformation.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to retrive current transformation matrix

    _PXCContent*        pContent;

    ...
    PXC_Matrix CurMatrix;

    HRESULT hr = PXC_CS_Get(pContent, &CurMatrix);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

...
```

2.3.1.3 PXC_CS_Rotate

PXC_CS_Rotate

Top Previous Next

PXC_CS_Rotate function rotates the coordinate system by the specified angle.

```
HRESULT  PXC_CS_Rotate(
    _PXCContent* content,
    double phi
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

phi

[in] Specifies the rotation angle in degrees. If this parameter is positive, the page will be rotated in counterclockwise direction; negative values specify a clockwise rotation.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

The following algorithm computes the new x-coordinate (x') for the point (x, y) that is rotated by the angle *A* in relation to the coordinate-space's origin:

$x' = (x * \cos A) - (y * \sin A)$

The following algorithm computes the y-coordinate (y') for the point (x, y) that is rotated by the angle *A* in relation to the original values.

$y' = (x * \sin A) + (y * \cos A)$

Where y' is the new y-coordinate, y is the original y-coordinate, and Dy is the vertical distance to which the point was moved.

The two rotation transformations can be combined in a 2-by-2 matrix as follows:.

```
|x' y'| == |x y| * | cos A   sin A|
                   |-sin A   cos A|
```

The 2-by-2 matrix that produced the rotation contains the following values (for *A* = 30°):

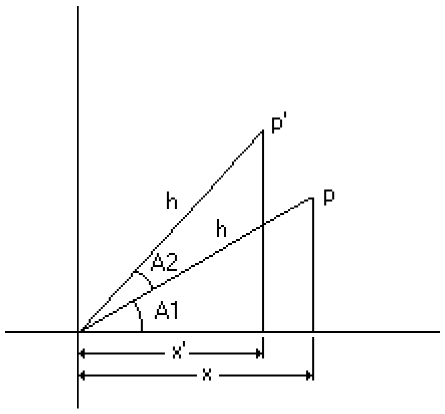
```
| 0.8660   0.5000|

|-0.5000   0.8660|
```

Rotation Algorithm Derivation

Rotation algorithms are based on trigonometry's addition theorem stating that the trigonometric function of a sum of two angles (*A1* and *A2*) can be expressed in terms of the trigonometric functions of the two angles.

```
sin(A1 + A2) = (sin A1 * cos A2) + (cos A1 * sin A2)
cos(A1 + A2) = (cos A1 * cos A2) - (sin A1 * sin A2)
```

The following illustration shows a point *p* rotated counterclockwise to a new position *p'*. In addition, it shows two triangles formed by a line drawn from the coordinate-space origin to each point and a line drawn from each point through the x-axis.



Using trigonometry, the x-coordinate of point *p* may be obtained by multiplying the length of the hypotenuse *h* by the cosine of *A1*.

$x = h * \cos A1$

The y-coordinate of point *p* may be obtained by multiplying the length of the hypotenuse *h* by the sine of *A1*.

$y = h * \sin A1$

Likewise, the x-coordinate of point *p'* may be obtained by multiplying the length of the hypotenuse *h* by the cosine of (*A1* + *A2*).

$x' = h * \cos (A1 + A2)$

Finally, the y-coordinate of point *p'* may be obtained by multiplying the length of the hypotenuse *h* by the sine of (*A1* + *A2*).

$y' = h * \sin (A1 + A2)$

Using the addition theorem, the preceding algorithms become the following:

$x' = (h * \cos A1 * \cos A2) - (h * \sin A1 * \sin A2)$
$y' = (h * \cos A1 * \sin A2) + (h * \sin A1 * \cos A2)$

The rotation algorithms for a given point rotated by angle *A2* may be obtained by substituting x for each occurrence of (*h* * cos *A1*) and by substituting y for each occurrence of (*h* * sin *A1*).

```
x' = (x * cos A2) - (y * sin A2)
y' = (x * sin A2) + (y * cos A2)
```

Example (C++).

```
// Example shows how to rotate coordiante system
    // by 30 degrees in counterclockwise direction

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_CS_Rotate(pContent, 30.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.1.4 PXC_CS_Scale

PXC_CS_Scale

Top Previous Next

PXC_CS_Scale function scales the coordinate system.

```
HRESULT  PXC_CS_Scale(
    _PXCContent* content,
    double sx,
    double sy
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

sx

[in] *sx* specifies the horizontal scaling component.

sy

[in] *sy* specifies the vertical scaling component.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Vertical scaling is represented by the following algorithm.

```
y' = y * Sy,
```

where `y'` is the new length, `y` is the original length, and `Sy` is the vertical scaling factor.
Horizontal scaling is represented by the following algorithm.

```
x' = x * Sx,
```

where `x'` is the new length, `x` is the original length, and `Sx` is the horizontal scaling factor.
The vertical and horizontal scaling transformations can be combined into a single operation by using a 2-by-2 matrix.

```
|x' y'|  =  |Sx   0|  *  |x y|
            |0   Sy|
```

The 2-by-2 matrix that is produced by the scaling transformation contains the following values (*sx* = 1.0, *sy* = 2.0).

```
|1.0   0.0|
|0.0   2.0|
```

Example (C++).

```
// Example shows how to scale coordiante system

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_CS_Scale(pContent, 1.0, 2.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.1.5 PXC_CS_Skew

PXC_CS_Skew

Top Previous Next

PXC_CS_Skew skews the **x** axis of the coordinate system by an angle *alpha*, and the **y** axis by an angle *beta*.

```
HRESULT  PXC_CS_Skew(
    _PXCContent* content,
    double alpha,
    double beta
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

alpha

[in] Specifies the angle, in degrees, by which the **x** axis should be skewed.

beta

[in] Specifies the angle, in degrees, by which the **y** axis should be skewed.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to skew the X axis
// by 30 degrees in counterclockwise direction
// and Y axis in 25 degrees in counterclockwise direction

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_CS_Skew(pContent, 30.0, 25.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.1.6 PXC_CS_Translate

PXC_CS_Translate

Top Previous Next

PXC_CS_Translate translates (shifts) the coordinate system.

```
HRESULT  PXC_CS_Translate(
    _PXCContent* content,
    double dx,
    double dy
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

dx

[in] Specifies the horizontal translation component, in points.

dy

[in] Specifies the vertical translation component, in points.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Horizontal translation may be represented by the following algorithm:

```
x' = x + Dx,
```

where `x'` is the new x-coordinate, `x` is the original x-coordinate, and `Dx` is the horizontal distance moved.
Vertical translation can be represented by the following algorithm:

```
y' = y + Dy,
```

where `y'` is the new y-coordinate, `y` is the original y-coordinate, and `Dy` is the vertical distance moved.
The horizontal and vertical translation transformations can be combined into a single operation by using a 3-by-3 matrix.

```
                     |1   0   0|
|x' y' 1| = |x y 1| * |0   1   0|
                     |Dx  Dy  1|
```

(The rules of matrix multiplication state that the number of rows in one matrix must equal the number of columns in the other. The integer `1` in the matrix `|x y 1|` is a placeholder added to meet this requirement.)

The 3-by-3 matrix that produced the illustrated translation transformation contains the following values.

```
|1     0    0|
|0     1    0|
|10.0  0.0  1|
```

Example (C++).

```
// Example shows how to translate the coordinate system along the X axis

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_CS_Translate(pContent, 10.0, 0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.2 Text and Fonts

Text and Fonts

Top Previous Next

Please read this section for information on font selection within the PDF specification:

- Font selection details

These functions create or modify text content:

- PXC_AddFontA
- PXC_AddFontFromFileA
- PXC_AddFontFromFileW
- PXC_AddFontW
- PXC_ClearNoEmbeddList
- PXC_DrawTextExW
- PXC_GetCurrentFont
- PXC_GetFontInfo
- PXC_GetStringWidthA
- PXC_GetStringWidthW
- PXC_GetTextOptions
- PXC_SetCharSpacing
- PXC_SetCurrentFont
- PXC_SetEmbeddingOptions
- PXC_SetFontEmbeddA
- PXC_SetFontEmbeddW
- PXC_SetTextLeading
- PXC_SetTextOptions
- PXC_SetTextRise
- PXC_SetTextRMode
- PXC_SetTextScaling
- PXC_SetWordSpacing
- PXC_TCS_Get
- PXC_TCS_Transform

- PXC_TextOutA
- PXC_TextOutW

2.3.2.1 Font selection details

Font selection details

Top Previous Next

The fiunctions **PXC_AddFontA** and **PXC_AddFontW** select the fonts to be used. If the designated font is not available, the **PDF-XChange** library attempts to match the font most closely resembling the requested font in terms of its characteristics.

In the example below, if the font named "Arial" with a weight of 300 is added (`FW_LIGHT`) with italic outlines, to the document, the following logic is applied:

The primary criteria used to match fonts within the document is the font name, in the example below, usually four fonts would be available (in a typical installation):

1. **Arial** (weight 400, non-italic)

2. **Arial** Italic (weight 400, italic)

3. **Arial** Bold (weight 700, non-italic)

4. **Arial** Bold Italic (weight 700, italic)

(Hence 4 fonts match the initial criteria parameters - not one!)

The next criteria is the font specified: *italic*. The library will therefore match fonts **2** and **4** - both being italic.

The Third (and last) criteria is font weight - again the closest match possible is selected, in this case the selected font would be #**2**, even it doesn't match exactly - however it is the closest match available.

This is so because many fonts have differing weights applied for normal or bold outlines (i.e. for bold fonts a typical weight is 700, however some fonts specify a weight of only 600 for this property). Furthermore, many fonts have have more than just the two variants detailed: normal (regular) and bold. A significant benefit of this approach is that an application need not know exact information about installed fonts on a user system. However to be reliable it is necessary to be sure that at least one variant of the required font is installed. Of course this does not *guarantee* a match will be found, but does allow PDF files to be generated in a wide set circumstances and, provided more exotic fonts are avoided or are supplied with your application, it should provide a generally reliable and robust means of ensuring consistent output for a broad range of installation sites.

Standard font weight table

Constant	Value
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600

FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_HEAVY	900
FW_BLACK	900

2.3.2.2 PXC_AddFontA

PXC_AddFontA

Top Previous Next

PXC_AddFontA adds an already installed font to the PDF object.

```
HRESULT  PXC_AddFontA(
    _PXCDocument* pdf,
    DWORD dwWeight,
    BOOL bItalic,
    LPCSTR fName,
    DWORD* font
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

dwWeight

[in] *dwWeight* specifies the weight of the font in a range 0 through 1000. For example, 400 is normal and 700 is bold. If this value is zero, a default weight is used. See the **Standard font weight table** in Font selection for details.

bItalic

[in] *bItalic* specifies font is italic or not.

fName

[in] *fName* specifies the font name.

Note: This may be either the font **Family name** or the **Face name**.

font

[out] *font* specifies a pointer to the variable which receives the font id.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

If the font with the specified parameters has been previously added to the pdf object, the function will return its identifier.

Supported font types: TrueType®, Postscript® Type1®, OpenType® with Postscript® or TrueType® outlines

Remarks

The function **PXC_AddFontA** has one limitation: it cannot be used to add fonts where the font name contains UNICODE characters or characters outside the `ANSI` code page (for example, some Chinese fonts). It is therefore recommended to use **PXC_AddFontW** in place of **PXC_AddFontA**.

Example (C++).

```
_PXCDocument*        pdf;

...

// Add bold 'Tahoma' font

DWORD fontID;

HRESULT res = PXC_AddFontA(pdf, FW_BOLD, FALSE, "Tahoma", &fontID);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.3.2.3 PXC_AddFontFromFileA

PXC_AddFontFromFileA

Top Previous Next

PXC_AddFontFromFileA adds a font from the specified file(s) to the PDF object.

```
HRESULT  PXC_AddFontFromFileA(
    _PXCDocument* pdf,
    LPCSTR fName,
    LPCSTR aName,
    DWORD* font
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

fName

[in] *fName* specifies the font file name.

Note: For Type1® fonts use the name of the **.pfa**/**.pfb** file.

aName

[in] *aName* specifies another font file name. For most fonts this parameter will be either `NULL` or an empty string.

Note: For Type1® fonts use the name of the corresponding **.pfm** file.

font

[out] *font* specifies pointer to the variable which receives the font id.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

If a specified font has previously been added to the pdf object, the function will return its identifier. However, it is strongly recommended to avoid adding such fonts numderous times to the same pdf object - as this will decrease program performance.

Supported fonts types: TrueType®, Postscript® Type1®, OpenType® with Postscript® or TrueType® outlines

Example (C++).

```
    _PXCDocument*        pdf;

    ...

    // Add font from file

    DWORD fontID;

    HRESULT res = PXC_AddFontFromFileA(pdf, "C:\\WINDOWS\\Fonts\\lsansi.ttf",
NULL, &fontID);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.3.2.4 PXC_AddFontFromFileW

PXC_AddFontFromFileW

Top Previous Next

PXC_AddFontFromFileW adds a font from the specified file(s) to the PDF object.

```
HRESULT  PXC_AddFontFromFileW(
     _PXCDocument* pdf,
     LPCWSTR fName,
     LPCWSTR aName,
     DWORD* font
);
```

Parameters

pdf

[in] Specifies the PDF object previously created by the function **PXC_NewDocument**.

fName

[in] *fName* specifies the font file name.

Note: For Type1® fonts and should be the name of the **.pfa**/**.pfb** file.

aName

[in] *aName* specifies another font file name. For most fonts this parameter should be either `NULL` or an empty string.

Note: For Type1® fonts this should be the name of the corresponding **.pfm** file.

font

[out] *font* specifies a pointer to the variable which receives the font id.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

If the specified font has been already added to the pdf object, the function will return its identifier. However, adding such fonts many times to the same pdf object is not recommended - this will decrease program performance significantly.

Supported fonts types: TrueType®, Postscript® Type1®, OpenType® with Postscript® or TrueType® outlines

Example (C++).

```
      _PXCDocument*        pdf;

   ...

   // Add font from file

   DWORD fontID;

   HRESULT res = PXC_AddFontFromFileW(pdf, L"C:\\WINDOWS\\Fonts\\lsansi.ttf",
NULL, &fontID);
   if (IS_DS_FAILED(res))
   {
       // Handle error
   }
   ...
```

2.3.2.5 PXC_AddFontW

PXC_AddFontW

Top Previous Next

PXC_AddFontW adds an already installed font to the PDF object.

```
HRESULT  PXC_AddFontW(
    _PXCDocument* pdf,
    DWORD dwWeight,
    BOOL bItalic,
    LPCWSTR fName,
    DWORD* font
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

dwWeight

[in] *dwWeight* specifies the weight of the font in the range 0 through 1000. For example, 400 is normal and 700 is bold. If this value is zero, a default weight is used. See **Standard font weight table** in Font selection for details.

bItalic

[in] *bItalic* specifies whether the font is italic or not.

fName

[in] *fName* specifies the font name.

Note: This may be either the font **Family name** or **Face name**.

font

[out] *font* specifies a pointer to the variable which receives the font id.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

If the font with the specified parameters has already been added to the pdf object, the function will return its identifier.

Supported fonts types: TrueType®, Postscript® Type1®, OpenType® with Postscript® or TrueType® outlines

Examples

Example (C++).

```
    _PXCDocument*        pdf;

    ...

    // Add bold 'Tahoma' font

    DWORD fontID;

    HRESULT res = PXC_AddFontW(pdf, FW_BOLD, FALSE, L"Tahoma", &fontID);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.3.2.6 PXC_ClearNoEmbeddList

PXC_ClearNoEmbeddList

Top Previous Next

PXC_ClearNoEmbeddList clears the list of fonts both required and not required for embedded inclusion within the document. See **PXC_SetFontEmbeddA** for more information.

```
HRESULT  PXC_ClearNoEmbeddList(
    _PXCDocument* pdf
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
_PXCDocument*        pdf;
    ...

    // Clear fonts list

    HRESULT res = PXC_ClearNoEmbeddList(pdf);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.3.2.7 PXC_DrawTextExW

PXC_DrawTextExW

Top Previous Next

PXC_DrawTextExW draws a portion of the text, formatted according to the parameters passed, in the specified rectangle.

```
HRESULT  PXC_DrawTextExW(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCWSTR str,
    LONG sPos,
    LONG len,
    DWORD flags,
    LPPXC_DrawTextStruct lpOptions
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect

[in] *rect* specifies the rectangle, inside of which the text will be drawn.

str

[in] *str* specifies the beginning of the text buffer.

sPos

[in] *sPos* specifies the position of the first symbol, which will be drawn in the rectangle *rect*, in the buffer *str*.

len

[in] *len* specifies total length of the text buffer *str*, including first *sPos* symbols, which will not be drawn.

Note:

The length of the buffer may be equal to `-1` - in this case it must contain the string which ends with the NULL.
It's recommended to set the length of the text in the buffer if known.

flags

[in] *flags* specifies text drawing flags. They specify horizontal and vertical text alignment and give some additional capabilities. See tables below.

Horizontal alignment flags

Constant	Value	Description
`DTF_Align_Left`	`0x0000`	Left aligned lines
`DTF_Align_Center`	`0x0001`	Centered lines
`DTF_Align_Right`	`0x0002`	Right aligned lines
`DTF_Align_Justify`	`0x0003`	All lines except last in the paragraph will be justified to have same width as destination rectangle by correcting word and character spacing separately for each line.
`DTF_Align_FullJustify`	`0x0007`	Same as `DTF_Align_Justify`, but the last line will be justified too.

Vertical alignment flags

Constant	Value	Description
DTF_Align_Top	0x0000	Text block will be aligned to the top of destination rectangle
DTF_Align_VCenter	0x0010	Text block will be centered in destination rectangle
DTF_Align_Bottom	0x0020	Text block will be aligned to the bottom of destination rectangle

Additional flags

Constant	Value	Description
DTF_CalcOnly	0x1000	If this flag is specified there will be no text output produced, but *endY* and *usedChars* fields of passed **PXC_DrawTextStruct** will be calculated and filled. See **PXC_DrawTextStruct** for details.

Note: The value of this parameter should be the combination of one of horizontal alignment flags, one of vertical alignment flags and any set of the additional flags.

lpOptions

[in/out] *lpOptions* specifies the pointer to **PXC_DrawTextStruct** structure. If `NULL` is passed as *lpOptions*, the previous text formatting settings will be used, but in this case you will not know how many symbols and till what position are drawn.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Function **PXC_DrawTextExW** was developed specially for drawing large amounts of text on one or a few pages. It formats the text according to the parameters, set in **PXC_DrawTextStruct** (see *lpOptions* description) and reports how many symbols fit into the specified rectangle and the vertical position of the last drawn symbol. It allows easily and effectively draw large amounts of text on the pages without analyzing length and position of every single string.

Remarks

Function **PXC_DrawTextExW** saves such previous text formatting settings like *CharacterSpacing*, *WordSpacing* and *TextScaling*, but it doesn't save current font name and font size.

Example (C++).

```
// Example shows how to print text in the specified rectangle
// with the desired justification and other parameters

    _PXCContent*         pContent;

    ...

    // Rectangle box where the text will be placed

    PXC_RectF rect;
    rect.left = I2L(1);
    rect.right = I2L(5);
    rect.top = I2L(11);
    rect.bottom = I2L(7);

    // Text to be output

    LPCWSTR        pwText = L"This is sample text that will be printed...";

    HRESULT hr = PXC_DrawTextExW(pContent, &rect, pwText, 0, ::lstrlenW
(pwText), DTF_Align_Justify | DTF_Align_Top, NULL);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.2.8 PXC_GetCurrentFont

PXC_GetCurrentFont

Top Previous Next

PXC_GetCurrentFont returns information relating to current font and size settings.

```
HRESULT  PXC_GetCurrentFont(
    _PXCContent* content,
    DWORD* fontID,
    double* fSize
);
```

Parameters

*_PXCContent**
: [in] Parameter `content` specifies the identifier of the page content to which the function will be applied.

fontID
: [in, out] *fontID* specifies a pointer to a variable containing the id of the current font. *fontID* may be equal to `NULL` if the id of the current font is not required.

fSize
: [in, out] *fontID* specifies a pointer to a variable containing the id of the current font size. *fontID* may be equal to `NULL` if the id of the current font size is not required.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
        _PXCContent*        pContent;

    ...

    // Get current font information

    DWORD        fontID;
    double        fontSize;

    HRESULT hr = PXC_GetCurrentFont(pContent, &fontID, &fontSize);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.2.9 PXC_GetFontInfo

PXC_GetFontInfo

Top Previous Next

PXC_GetFontInfo retrieves retrieves text metrics for current font.

```
HRESULT  PXC_GetFontInfo(
    _PXCContent* content,
    PXC_FontInfo* fInfo
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

options
: [in] Pointer to the **PXC_FontInfo** structure which receives the font metrics information.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
   PXC_FontInfo finfo;
   HRESULT hr = PXC_OK;
   memset(&finfo, 0, sizeof(PXC_FontInfo));
   finto.cbSize = sizeof(PXC_FontInfo);
   hr = PXC_GetFontInfo(page, &finfo);
   . . .
```

2.3.2.10 PXC_GetStringWidthA

PXC_GetStringWidthA

Top Previous Next

PXC_GetStringWidthA calculates and returns the width of the specified ASCII string in points using the current font and font size. Returned widths are calculated without consideration of character or word spacing, and without horizontal scaling.

```
HRESULT  PXC_GetStringWidthA(
    _PXCContent* content,
    LPCSTR lpszText,
    LONG cbLen,
    double* width
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

lpszText
: [in] *lpszText* points to an ASCII string.

cbLen
: [in] *cbLen* specifies the length (number of chars) of the text. If this value is 0, the string is assumed

to be null-terminated and the length is calculated automatically.

width

[out] *width* specifies the pointer to the variable that receives the string length in points.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
        _PXCContent* pContent;

    ...

    // Retrieve string width

    double width;

    HRESULT res = PXC_GetStringWidthA(pContent, "Sample text string", -1,
&width);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.3.2.11 PXC_GetStringWidthW

PXC_GetStringWidthW

Top Previous Next

PXC_GetStringWidthW calculates and returns the width of the specified UNICODE string in points using the current font and font size. Returned widths are calculated without consideration to character or word spacing, and without horizontal scaling.

```
HRESULT  PXC_GetStringWidthW(
    _PXCContent* content,
    LPCWSTR lpwszText,
    LONG cbLen,
    double* width
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

lpwszText

[in] *lpwszText* points to a UNICODE string.

cbLen

[in] *cbLen* specifies the length (number of chars) of the text. If this value is 0, the string is assumed to be null-terminated and the length is calculated automatically.

width

[out] *width* specifies a pointer to the variable that receives the string length in points.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
_PXCContent*        pContent;

    ...

    // Retrive string width

    double width;

    HRESULT res = PXC_GetStringWidthW(pContent, L"Sample text string", -1,
&width);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.3.2.12 PXC_GetTextOptions

PXC_GetTextOptions

Top Previous Next

PXC_GetTextOptions retrieves current settings for text output.

```
HRESULT  PXC_GetTextOptions(
    const _PXCContent* content,
    PXC_TextOptions* options
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

options

[in] Pointer to the **PXC_TextOptions** structure which receives the text options.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    _PXCContent* pContent;

  ...

  // Get current text settings for text output

  PXC_TextOptions tOpts;

  // Set correct structure size

  tOpts.cbSize = sizeof(PXC_TextOptions);

  // Get current settings

  HRESULT hr = PXC_GetTextOptions(pContent, &tOpts);
  if (IS_DS_FAILED(hr))
  {
      // Handle error
      ...
  }

  // Now 'tOpts' contains current settings for text output
  ...
```

2.3.2.13 PXC_SetCharSpacing

PXC_SetCharSpacing

Top Previous Next

PXC_SetCharSpacing sets additional spacing between characters when drawing text.

```
HRESULT  PXC_SetCharSpacing(
    _PXCContent* content,
    double cs,
    double* oldcs
);
```

Parameters

content

[in] Parameter *content* specifies identifier of page content to which the function will be applied.

cs

[in] *cs* specifies new character spacing in unscaled text units (see **Comments** for details).

oldcs

[out] *oldcs* specifies a pointer to a variable which receives the previous value for the character spacing.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Character spacing is set in unscaled text units. This means that it is not dependent on the font size, so when you set character spacing to 2 (for example) you will get an additional 2 points (1/72 of inch) of space between consecutive characters. However this is only true if no change to the text or graphics transformation matrix has taken place, otherwise the transformation will be the same as it is for text.

Example (C++).

```
    _PXCContent*        pContent;

  ...

  // Set new character spacing parameter and remember the old one

  double oldCS;
  PXC_SetCharSpacing(pContent, 10, &oldCS);

  // Do some text out, i.e.

  PXC_PointF origin = { I2L(2), I2L(1.5) };
  PXC_TextOutW(pContent, &origin, L"Some text", -1);

  // Return to original character spacing

  PXC_SetCharSpacing(pContent, oldCS, NULL);
```

2.3.2.14 PXC_SetCurrentFont

PXC_SetCurrentFont

Top Previous Next

PXC_SetCurrentFont specifies the font index and font size for the specified content and all subsequent text drawing until reset.

```
HRESULT  PXC_SetCurrentFont(
    _PXCContent* content,
    DWORD fontID,
    double fSize
);
```

Parameters

content

[in] Parameter *content* specifies identifier of page content to which the function will be applied.

fontID

[in] Specifies the font identifier, as previously returned during add font (for example, with **PXC_AddFontA** function), to be set as the current font.
If *fontID* has a value `-1` (`0xFFFFFFFF`), current font will not be changed.

fSize

[in] Specifies size in points of the current font.
If this value is `0`, current font size will not be changed.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to print one word with the double font size
// then return to the original font size

    _PXCContent*        pContent;

    ...

    // Get current font information

    DWORD        fontID;
    double        fontSize;

    HRESULT hr = PXC_GetCurrentFont(pContent, &fontID, &fontSize);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    fontSize *= 2.0;
    hr = PXC_SetCurrentFont(pContent, fontID, fontSize);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Out some text
    PXC_PointF origin = { I2L(2), I2L(1.5) };
    PXC_TextOutW(pContent, &origin, L"Word", -1);

    // Return to original font size

    fontSize /= 2.0;
    hr = PXC_SetCurrentFont(pContent, fontID, fontSize);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.2.15 PXC_SetEmbeddingOptions

PXC_SetEmbeddingOptions

Top Previous Next

PXC_SetEmbeddingOptions sets the embedding parameters for fonts used in the document.

```
HRESULT  PXC_SetEmbeddingOptions(
    _PXCDocument* pdf,
    BOOL bAllowEmbedding,
    BOOL bForceEmbedding,
    BOOL bToUnicode
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

bAllowEmbedding

[in] If this parameter is `TRUE`, all fonts (except those added to the list of fonts for exclusion from embedding parameters; see **PXC_SetFontEmbeddA**) will be embedded into the file. This means that any party viewing the document does not need these fonts to be resident on their own PCs as the information required for formatting is contained within the document for these fonts.
Font embedding does however make files larger.

bForceEmbedding

[in] Some TrueType fonts are protected from embedding (sometimes for copyright reasons). If the *bForceEmbedding* parameter is `TRUE`, protection information held by TrueType fonts will be ignored and these fonts will be embedded (if needed) into PDF file. Otherwise such fonts will not be embedded.

bToUnicode

[in] If this parameter is `TRUE`, the special information for embedded Unicode fonts will be added into the PDF document. This information allows the viewer to correctly extract the text from the PDF document.

However including this information increases file size.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

If using this feature, it is the responsibility of the PDF document author to ensure they are licensed to do so!

Example (C++).

```
_PXCDocument*        pdf;
 ...

 // Set next embedding options:
 // Embed all fonts (including protected fonts)
 // Embed special UNICODE fonts information

 // Set options

 HRESULT res = PXC_SetEmbeddingOptions(pdf, TRUE, TRUE, TRUE);
 if (IS_DS_FAILED(res))
 {
     // Handle error
 }
 ...
```

2.3.2.16 PXC_SetFontEmbeddA

PXC_SetFontEmbeddA

Top Previous Next

PXC_SetFontEmbeddA adds a font either into the list of fonts to be excluded from embedding or into the list of the fonts which will always be embedded (if the specified font is used in the document created).

```
HRESULT  PXC_SetFontEmbeddA(
    _PXCDocument* pdf,
    LPCSTR lpszFontName,
    PXC_EmbeddType bEmbedd
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

lpszFontName

[in] Pointer to null-terminated string specifying font name.

bEmbedd

[in] *bEmbedd* specifies the list to which the font will be added, and can be one of the following values:

Value	Description
EmbeddType_NeverEmbedd	The font *lpszFontName* will be added to the list of fonts, excluded from embedding.
EmbeddType_ForceEmbedd	The font *lpszFontName* will be added to the list of fonts, to be embedded into a document.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function has a UNICODE equivalent function **PXC_SetFontEmbeddW**.

Example (C++).

```
    _PXCDocument*        pdf;
...

// Embed standard 'Courier' font

// Set font embedding

HRESULT res = PXC_SetFontEmbeddA(pdf, "Courier", EmbeddType_ForceEmbedd);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.3.2.17 PXC_SetFontEmbeddW

PXC_SetFontEmbeddW

Top Previous Next

PXC_SetFontEmbeddW adds the specified font either into the list of the fonts to be excluded from embedding or into the list of the fonts which will be embedded (if the specified font is used in a document).

```
HRESULT  PXC_SetFontEmbeddW(
    _PXCDocument* pdf,
    LPCWSTR lpwszFontName,
    PXC_EmbeddType bEmbedd
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

lpwszFontName

[in] Pointer to a null-terminated string specifying a font name.

bEmbedd

[in] *bEmbedd* specifies the list to which font will be added, and can be one of the following values:

Value	**Description**
EmbeddType_NeverEmbedd	The font *lpwszFontName* will be added to the list of fonts for exclusion from embedding.
EmbeddType_ForceEmbedd	The font *lpwszFontName* will be added to the list of fonts, to be embedded into a document.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function has an ASCII equivalent function **PXC_SetFontEmbeddA**.

Example (C++).

```
    _PXCDocument*        pdf;
...

// Embed standard 'Courier' font

// Set font embedding

HRESULT res = PXC_SetFontEmbeddW(pdf, L"Courier", EmbeddType_ForceEmbedd);
if (IS_DS_FAILED(res))
{
    // Handle error
}
...
```

2.3.2.18 PXC_SetTextLeading

PXC_SetTextLeading

Top Previous Next

The **PXC_SetTextLeading** function sets the text *leading* value.

```
HRESULT  PXC_SetTextLeading(
    _PXCContent* content,
    double leading,
    double* oldleading
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

leading

[in] Specifies the text leading value.

oldleading

[out] Pointer to a variable which will contain the previous text leading value after the function return. This pointer may be `NULL` if you are not interested in this value.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Leading specifies the vertical distance between the baselines of adjacent lines of text and is measured in unscaled text space units .

Example (C++).

```
    _PXCContent*        pContent;

  ...

  // Set new leading value and remember the old one

  double oldLeading;
  PXC_SetTextLeading(pContent, 150, &oldLeading);

  // Do some text out, i.e.

  PXC_PointF origin = { I2L(2), I2L(1.5) };
  PXC_TextOutW(pContent, &origin, L"Some text", -1);

  // Return to original leading value

  PXC_SetTextLeading(pContent, oldLeading, NULL);
```

2.3.2.19 PXC_SetTextOptions

PXC_SetTextOptions

Top Previous Next

PXC_SetTextOptions sets the desired text options.

```
HRESULT  PXC_SetTextOptions(
    _PXCContent* content,
    const PXC_TextOptions* options
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

options
: [in] Pointer to **PXC_TextOptions** structure which specifies text options.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to print one word with the double font size
// then return to the original font size

    _PXCContent*        pContent;

    ...

    // Get current text settings for text output

    PXC_TextOptions tOpts, tOldOpts;

    // Set correct structure size

    tOpts.cbSize = sizeof(PXC_TextOptions);

    // Get current settings

    HRESULT hr = PXC_GetTextOptions(pContent, &tOpts);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Store current settings

    memcpy(&tOldOpts, &tOpts, sizeof(PXC_TextOptions));

    // Now change font size

    tOpts.fontSize *= 2;
    // Set this options

    hr = PXC_SetTextOptions(pContent, &tOpts);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Out some text

    PXC_PointF origin = { I2L(2), I2L(1.5) };
    PXC_TextOutW(pContent, &origin, L"Word", -1);

    // Return to original text output settings

    hr = PXC_SetTextOptions(pContent, &tOldOpts);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.2.20 PXC_SetTextRise

PXC_SetTextRise

Top Previous Next

The **PXC_SetTextRise** function sets the text rise.

```
HRESULT  PXC_SetTextRise(
    _PXCContent* content,
    double rise,
    double* oldrise
);
```

Parameters

content
[in] Parameter *content* specifies identifier of the page content to which the function will be applied.

rise
[in] Specifies text rise in points.

oldrise
[out] Pointer to a variable which will contain the previous text rise value after the function return.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Text rise specifies the distance to move the baseline up or down from its default location. Positive values move the baseline up. Adjustments to the baseline are useful for drawing superscripts or subscripts. The default location of the baseline can be restored by setting the text rise value to 0.

Example (C++).

```
      _PXCContent*        pContent;

    ...

    // Set new text rise parameter and remember the old one

    double oldRise;
    PXC_SetTextRise(pContent, P2L(10), &oldRise);

    // Do some text out, i.e.

    PXC_PointF origin = { I2L(2), I2L(1.5) };
    PXC_TextOutW(pContent, &origin, L"Some text", -1);

    // Return to original text rise

    PXC_SetTextRise(pContent, oldRise, NULL);
```

2.3.2.21 PXC_SetTextRMode

PXC_SetTextRMode

Top Previous Next

The **PXC_SetTextRMode** function sets the text rendering mode.

```
HRESULT  PXC_SetTextRMode(
    _PXCContent* content,
    PXC_TextRenderingMode mode,
    PXC_TextRenderingMode* oldmode
);
```

Parameters

content

[in] Parameter *content* specifies identifier of the page content to which the function will be applied.

mode

[in] Specifies text rendering mode. The possible modes are enumerated in **`PXC_TextRenderingMode`**. *(See comments for possible values)*

oldmode

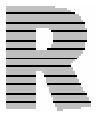
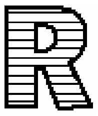
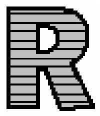
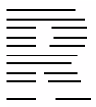
[out] Pointer to a variable which will contain the previous text rendering mode after the function return. *(See comments for possible values)*

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Possible values of **`PXC_TextRenderingMode`** are:

Value	Appearance	Meaning
`TextRenderingMode_Fill`		Fill text.
`TextRenderingMode_Stroke`		Stroke text.
`TextRenderingMode_FillStroke`		Fill, then stroke, text.
`TextRenderingMode_None`		Neither fill nor stroke text (invisible).
`TextRenderingMode_Clip_Fill`		Fill text and add to path for clipping.
`TextRenderingMode_Clip_Stroke`		Stroke text and add to path for clipping.
`TextRenderingMode_Clip_FillStroke`		Fill, then stroke, text and add to path for clipping.
`TextRenderingMode_Clip`		Add text to path for clipping.

Example (C++).

```
        _PXCContent*        pContent;

    ...

    // Set fill rendering mode and store the old mode
    PXC_TextRenderingMode oldmode;
    PXC_SetTextRMode(pContent, TextRenderingMode_Fill, &oldmode);

    // Do some text out, i.e.

    PXC_PointF origin = { I2L(2), I2L(1.5) };
    PXC_TextOutW(pContent, &origin, L"Some text", -1);

    // Return to old rendering mode

    PXC_SetTextRMode(pContent, oldmode, NULL);
```

2.3.2.22 PXC_SetTextScaling

PXC_SetTextScaling

Top Previous Next

PXC_SetTextScaling sets the horizontal scaling for text drawing.

```
HRESULT  PXC_SetTextScaling(
     _PXCContent* content,
     double scale,
     double* oldscale
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

scale
: [in] Specifies the text scaling level.

oldscale
: [out] Pointer to a variable which will contain the previous text scaling level after the function return.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
        _PXCContent*        pContent;

    ...

    // Set new scalling and remember the old one

    double oldScale;
    PXC_SetTextScaling(pContent, 150, &oldScale);

    // Do some text out, i.e.

    PXC_PointF origin = { I2L(2), I2L(1.5) };
    PXC_TextOutW(pContent, &origin, L"Some text", -1);

    // Return to original scalling

    PXC_SetTextScaling(pContent, oldScale, NULL);
```

2.3.2.23 PXC_SetWordSpacing

PXC_SetWordSpacing

Top Previous Next

The **PXC_SetWordSpacing** function sets additional spacing between words when drawing text.

```
HRESULT  PXC_SetWordSpacing(
    _PXCContent* content,
    double ws,
    double* oldws
);
```

Parameters

content

[in] Parameter *content* Parameter *content* specifies identifier for the page content to which the function will be applied.

ws

[in] *ws* specifies new word spacing in unscaled text units (see **Comments** for details).

oldws

[out] *oldws* specifies a pointer to variable which will receive the previous value of word spacing.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Word spacing is set in unscaled text units and is not dependant on font size, when you set word spacing to 10 (for example) there will be an additional 10 points (1/72 of inch) of space between consecutive words. However this is correct only when you do not change the text or graphics transformation matrix, otherwise it will be transformed with the same values as text.

Example (C++).

```
   _PXCContent*        pContent;
```

```
    ...

    // Set new word spacing parameter and remember the old one

    double oldWS;
    PXC_SetWordSpacing(pContent, -P2L(10), &oldWS);

    // Do some text out, i.e.

    PXC_PointF origin = { I2L(2), I2L(1.5) };
    PXC_TextOutW(pContent, &origin, L"Some text", -1);

    // Return to original word spacing

    PXC_SetWordSpacing(pContent, oldWS, NULL);
```

2.3.2.24 PXC_TCS_Get

PXC_TCS_Get

Top Previous Next

PXC_TCS_Get returns the current text matrix of the coordinate system transformation.

```
HRESULT  PXC_TCS_Get(
    const _PXCContent* content,
    LPPXC_Matrix m
);
```

Parameters

content
[in] Parameter *content* specifies the identifier of the page content to which function will be applied.

m
[out] Pointer to the structure of the **PXC_Matrix** type, which contains the current text matrix for the transformation.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to retrive current text transformation matrix

    _PXCContent*        pContent;

    ...
    PXC_Matrix CurTxtMatrix;

    HRESULT hr = PXC_TCS_Get(pContent, &CurTxtMatrix);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

...
```

2.3.2.25 PXC_TCS_Transform

PXC_TCS_Transform

Top Previous Next

The **PXC_TCS_Transform** function sets the text transformation matrix.

```
HRESULT  PXC_TCS_Transform(
    _PXCContent* content,
    LPCPXC_Matrix m
);
```

Parameters

content
[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

m
[in] Pointer to an **PXC_Matrix** structure that contains the transformation data.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how rotate text coordinate system by specified angle

    // Mathematic functions are required (for sin and cos calculations)
    #include <math.h>

    // Define 'Pi' constant if not defined
    #ifndef M_PI
        #define M_PI       3.14159265358979323846
    #endif

    // Rotate text matrix by 'angle'
```

```
HRESULT Rotate_TCS(_PXCPage* page, double angle)
{
    double a, sina, cosa;
    PXC_Matrix TxtMatrix;

    // Some intermediate calculations

    a = angle * M_PI / 180.0;
    sina = sin(a);
    cosa = cos(a);

    // Set matrix elements

    TxtMatrix.a = cosa;
    TxtMatrix.b = sina;
    TxtMatrix.c = -sina;
    TxtMatrix.d = cosa;
    TxtMatrix.e = 0.0;
    TxtMatrix.f = 0.0;

    // Do transformation

    return PXC_TCS_Transform((_PXCContent*)page, &TxtMatrix);
}
```

2.3.2.26 PXC_TextOutA

PXC_TextOutA

Top Previous Next

PXC_TextOutA outputs a text string starting from the specified position using the current text formatting settings.

```
HRESULT  PXC_TextOutA(
    _PXCContent* content,
    LPCPXC_PointF origin,
    LPCSTR lpszText,
    LONG cbLen
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

origin

[in] *origin* specifies the coordinates of the point, from which the string output will begin.

lpszText

[in] *lpszText* specifies a pointer to the buffer, containing the string.

cbLen

[in] *cbLen* specifies the string length in symbols.

Note: The length of the buffer may be equal to `-1` - in this case it must contain the string which ending with NULL. It is also recommended to specify the text length in the buffer - if known.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Function **PXC_TextOutA** outputs the string without taking into the consideration its length and will carry-over additional content to the next line. To output longer strings and/or strings with hyphens, the **PXC_DrawTextExW** function should be used.

Remarks

Strings generated using this function must be `ANSI` strings. To output strings, which cannot be converted to `ANSI`, the **PXC_TextOutW** function should be used.

Example (C++).

```
_PXCContent*        page;

    ...

    // Start point for text out

    PXC_PointF or;

    or.x = I2L(2);
    or.y = I2L(1.5);

    // Out some text to page

    HRESULT res = PXC_TextOutA(page, &or, "Some text", -1);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.3.2.27 PXC_TextOutW

PXC_TextOutW

Top Previous Next

PXC_TextOutW outputs the text string starting from the specified position using the current text formatting settings.

```
HRESULT  PXC_TextOutW(
    _PXCContent* content,
    LPCPXC_PointF origin,
    LPCWSTR lpwszText,
    LONG cbLen
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content for which the function will be applied.

origin

[in] *origin* specifies the coordinates of the point, from which the string output will start.

lpszText

[in] *lpwszText* specifies a pointer to the buffer, which contains the string.

cbLen

[in] *cbLen* specifies the string length in symbols.

Note: The length of the buffer may be equal to `-1` - in this case it must contain a string which ends with NULL.
It is also recommended to specify the text length in the buffer if known.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Function **PXC_TextOutW** outputs the string without taking into the consideration its length and will carry-over additional content to the next line. To output longer strings and/or strings with hyphens, the **PXC_DrawTextExW** function should be used.

Example (C++).

```
_PXCContent*        page;

    ...

    // Start point for text out

    PXC_PointF or;

    or.x = I2L(2);
    or.y = I2L(1.5);

    // Out some text to page

    HRESULT res = PXC_TextOutW(page, &or, L"Some text", -1);
    if (IS_DS_FAILED(res))
    {
        // Handle error
    }
    ...
```

2.3.3 Images

Images

Top Previous Next

These functions add and modify both 2D and 3D images to the PDF document:

- PXC_Add3DAnnotationA
- PXC_Add3DAnnotationW
- PXC_AddEnhMetafile
- PXC_AddImageA
- PXC_AddImageExA
- PXC_AddImageExW
- PXC_AddImageFromHBITMAP
- PXC_AddImageFromImageXChangePage
- PXC_AddImageFromIStream
- PXC_AddImageFromIStreamEx
- PXC_AddImageFromMemory
- PXC_AddImagePattern
- PXC_AddImageW
- PXC_AddStdMetafile
- PXC_AddU3DStream
- PXC_AddU3DViewToStream
- PXC_CloseImage
- PXC_CropImage
- PXC_GetImageColors
- PXC_GetImageDimension
- PXC_GetImageDPI
- PXC_MakeImageGrayscale
- PXC_MarkImageAsMask
- PXC_PlaceImage
- PXC_ReduceImageColors
- PXC_ScaleImage
- PXC_SetImageMask
- PXC_SetImageTransColor

2.3.3.1 PXC_Add3DAnnotationA

PXC_Add3DAnnotationA

Top Previous Next

Function **PXC_Add3DAnnotationA** adds to the content a *3D annotation*. A 3D annotation is used to provide a *virtual camera* for 3D objects added to the document.

```
HRESULT  PXC_Add3DAnnotationA(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCSTR pszTitle,
    DWORD dwAnnotOption,
    _PXCImage* AltImage,
    DWORD dw3DStreamID,
    const PXC_3DView* def_view,
    LONG def_view_id,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect

[in] Pointer to a `PXC_RectF` structure that specifying the bounding rectangle of the annotation.

pszTitle

[in] Pointer to a null-terminated string specifying the title of the annotation's pop-up window.

dwAnnotOption

[in] Combination of flags (enumerated within `PXC_Annot3DOptions` enum) which specifies the activation/deactivation options for the 3D view.

Constant	Value	Meaning
ActivateOn_Explicit	0x0000	The annotation should remain inactive until explicitly activated by a user action.
ActivateOn_OnPageOpen	0x0001	The annotation should be activated as soon as the page containing the annotation is opened.
ActivateOn_OnPageVisible	0x0002	The annotation should be activated as soon as any part of the page containing the annotation becomes visible.
ActivationEff_Live	0x0000	Real-time script-driven animations are enabled if present; if not, the artwork is instantiated.
ActivationEff_Loaded	0x0010	The artwork is instantiated, but real-time script-driven animations are disabled.
DeactivateOn_OnPageInvisible	0x0000	The annotation should be deactivated as soon as the page containing the annotation becomes invisible.
DeactivateOn_OnPageClose	0x0100	The annotation should be deactivated as soon as the page is closed.
DeactivateOn_Explicit	0x0200	The annotation should remain active until explicitly deactivated by a user action.
DeactivationEff_Unloaded	0x0000	The artwork will be unloaded (uninstantiated) upon deactivation of the annotation.
DeactivationEff_Loaded	0x1000	The artwork will be instantiated upon deactivation of the annotation, but real-time script-driven animations are disabled.
DeactivationEff_Live	0x2000	The artwork will be instantiated upon deactivation of the annotation, and real-time script-driven animations

are enabled.

AltImage

[in] Specifies the image handle (previously added to the document by the any relevant image function), to be used as the default appearance for the 3D artwork. This appearance is used by the viewer application to display 3D artwork while uninstantiated.

dw3DStreamID

[in] Specifies the handle of the U3D stream, previously added by the function **PXC_AddU3DStream**, from which the 3D artwork will be associated with the annotation.

def_view

[in] Pointer to a **PXC_3DView** structure that describes the default 3D view to be used with the annotation. If this paramter is `NULL`, parameter *def_view_id* should define the index of the view (added by the function **PXC_AddU3DViewToStream**), which should be used as the default view.

def_view_id

[in] Specifies the index of a U3D stream view which should be used as the default view. This parameter is ignored when *def_view* is `NULL`.

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the documents global settings will be used (for more information see **PXC_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function has a UNICODE equivalent - **PXC_Add3DAnnotationW**.

Example (C++)

```
// Example shows how to add 3D View annotation

   HRESULT TstU3D(_PXCDocument* pdf, LPCSTR U3D_FileName)
   {
       // Load U3D stream from specified file
       // and store it into buffer

       HANDLE f = CreateFile(U3D_FileName, GENERIC_READ, FILE_SHARE_READ,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
       if (f == INVALID_HANDLE_VALUE)
           return GetLastError();
       DWORD fsz = GetFileSize(f, NULL);
       LPBYTE buf = new BYTE[fsz];
       DWORD numreaded;
       ReadFile(f, buf, fsz, &numreaded, NULL);
       CloseHandle(f);

       // Add U3D stream to the document
```

```
        LONG u3d = PXC_AddU3DStream(pdf, buf, fsz);
        if (IS_DS_FAILED(u3d))
            return u3d;

        // Add A4 format page to the document

        _PXCPage* page = NULL;
        _PXCContent* cont = NULL;
        HRESULT res = PXC_AddPage(pdf, I2L(8.5), I2L(11), &page);
        if (IS_DS_FAILED(res))
            return res;
        cont = (_PXCContent*)page;

        // Fill rectangle for the annotation

        PXC_RectF r;
        r.left = I2L(1);
        r.right = I2L(5);
        r.top = I2L(11);
        r.bottom = I2L(7);

        // Fill common annotation information structure

        PXC_CommonAnnotInfo cai;
        memset(&cai, 0, sizeof(cai));
        cai.m_Flags = 0x44;
        cai.m_Color = 0xFF000000;

        // Fill 3D View structure

        PXC_3DView view = {sizeof(PXC_3DView)};
        lstrcpyW(view.m_ExtName, L"Default");
        view.m_CO = 1300.32;
        view.m_FOV = 30.0;
        view.m_C2W[0] = view.m_C2W[7] = 1.0;
        view.m_C2W[5] = -1.0;
        view.m_C2W[9] = -0.0893402;
        view.m_C2W[10] = -1300.32;
        view.m_C2W[11] = 20.335;

        // Add 3D View to stream

        PXC_AddU3DViewToStream(pdf, u3d, &view);

        // Add annotation

        res = PXC_Add3DAnnotationA(cont, &r, "Sample U3D Content",
ActivateOn_OnPageOpen | DeactivateOn_OnPageInvisible |
               ActivationEff_Live | DeactivationEff_Loaded, 0, u3d, &view, 0,
&cai);
```

```
        return res;
    }
```

2.3.3.2 PXC_Add3DAnnotationW

PXC_Add3DAnnotationW

Top Previous Next

PXC_Add3DAnnotationW adds to the content a *3D annotation*. A 3D annotation is used to provide a *virtual camera* for 3D objects that have been added to the document.

This function is the UNICODE equivalent of the funtion **PXC_Add3DAnnotationA**.

```
HRESULT  PXC_Add3DAnnotationW(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCWSTR pszTitle,
    DWORD dwAnnotOption,
    _PXCImage* AltImage,
    DWORD dw3DStreamID,
    const PXC_3DView* def_view,
    LONG def_view_id,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

content
[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect
[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

pszTitle
[in] Pointer to a null-terminated string that specifies the title of the annotation's pop-up window.

dwAnnotOption
[in] Combination of flags (enumerated as `PXC_Annot3DOptions` enum) which specifies the activation/deactivation options for the 3D view.

Constant	Value	Meaning
ActivateOn_Explicit	0x0000	The annotation should remain inactive until explicitly activated by a user action.
ActivateOn_OnPageOpen	0x0001	The annotation should be activated as soon as the page containing the annotation is opened.
ActivateOn_OnPageVisible	0x0002	The annotation should be activated as soon as any part of the page containing the annotation becomes visible.
ActivationEff_Live	0x0000	Real-time script-driven animations are enabled if present; if not, the artwork is instantiated.
ActivationEff_Loaded	0x0010	The artwork is instantiated, but real-time script-driven animations are disabled.
DeactivateOn_OnPageInvisi	0x0000	The annotation should be deactivated as soon as the

ble		page containing the annotation becomes invisible.
DeactivateOn_OnPageClose	0x0100	The annotation should be deactivated as soon as the page is closed.
DeactivateOn_Explicit	0x0200	The annotation should remain active until explicitly deactivated by a user action.
DeactivationEff_Unloaded	0x0000	The artwork will be unloaded (uninstantiated) upon deactivation of the annotation.
DeactivationEff_Loaded	0x1000	The artwork will be instantiated upon deactivation of the annotation, but real-time script-driven animations are disabled.
DeactivationEff_Live	0x2000	The artwork will be instantiated upon deactivation of the annotation, and real-time script-driven animations are enabled.

AltImage

[in] Specifies the image handle (previously added to the document by the relevant image function), which should be used as the default appearance for the 3D artwork. This appearence property is used by the viewer application to display 3D artwork whilst it is uninstantiated.

dw3DStreamID

[in] Specifies the handle of the U3D stream, previously added by the function **PXC_AddU3DStream** , from which 3D artwork will be associated with annotation.

def_view

[in] Pointer to a **PXC_3DView** structure describing the default 3D view used with an annotation. If this paramter is `NULL`, parameter *def_view_id* should define the index of the view (added by the function **PXC_AddU3DViewToStream**), to be used as the default view.

def_view_id

[in] Specifies the index of U3D stream's view which should be used as a default view. This paramter is ignored when *def_view* is `NULL`.

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the documents global settings will be used (for more information see **PXC_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function has an ASCII equivalent - **PXC_Add3DAnnotationA**.

Example (C++)

```
// Example shows how to add 3D View annotation

    HRESULT TstU3D(_PXCDocument* pdf, LPCSTR U3D_FileName)
    {
        // Load U3D stream from specified file
        // and store it into buffer
        HANDLE f = CreateFile(U3D_FileName, GENERIC_READ, FILE_SHARE_READ,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (f == INVALID_HANDLE_VALUE)
            return GetLastError();
        DWORD fsz = GetFileSize(f, NULL);
        LPBYTE buf = new BYTE[fsz];
        DWORD numreaded;
        ReadFile(f, buf, fsz, &numreaded, NULL);
        CloseHandle(f);

        // Add U3D stream to the document

        LONG u3d = PXC_AddU3DStream(pdf, buf, fsz);
        if (IS_DS_FAILED(u3d))
            return u3d;

        // Add A4 format page to the document

        _PXCPage* page = NULL;
        _PXCContent* cont = NULL;
        HRESULT res = PXC_AddPage(pdf, I2L(8.5), I2L(11), &page);
        if (IS_DS_FAILED(res))
            return res;
        cont = (_PXCContent*)page;

        // Fill rectangle for the annotation

        PXC_RectF r;
        r.left = I2L(1);
        r.right = I2L(5);
        r.top = I2L(11);
        r.bottom = I2L(7);

        // Fill common annotation information structure

        PXC_CommonAnnotInfo cai;
        memset(&cai, 0, sizeof(cai));
        cai.m_Flags = 0x44;
        cai.m_Color = 0xFF000000;

        // Fill 3D View structure

        PXC_3DView view = {sizeof(PXC_3DView)};
        lstrcpyW(view.m_ExtName, L"Default");
        view.m_CO = 1300.32;
        view.m_FOV = 30.0;
        view.m_C2W[0] = view.m_C2W[7] = 1.0;
        view.m_C2W[5] = -1.0;
        view.m_C2W[9] = -0.0893402;
        view.m_C2W[10] = -1300.32;
        view.m_C2W[11] = 20.335;

        // Add 3D View to stream

        PXC_AddU3DViewToStream(pdf, u3d, &view);

        // Add annotation

        res = PXC_Add3DAnnotationW(cont, &r, L"Sample U3D Content",
ActivateOn_OnPageOpen | DeactivateOn_OnPageInvisible |
                ActivationEff_Live | DeactivationEff_Loaded, 0, u3d, &view, 0,
&cai);
        return res;
    }
```

2.3.3.3 PXC_AddEnhMetafile

PXC_AddEnhMetafile

Top Previous Next

PXC_AddEnhMetafile adds an image, represented by enhanced metafile (HENHMETAFILE), to the PDF document.

```
HRESULT  PXC_AddEnhMetafile(
    _PXCDocument* pdf,
    HENHMETAFILE metafile,
    _PXCImage** image
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object, previously created by the function **PXC_NewDocument**.

metafile

[in] *metafile* specifies the Windows enhanced metafile object, which represents the image that will be added to the document.

image

[out] *image* specifies the pointer to a variable of `_PXCImage*` type, which adds an image in the *pdf* document.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

To have the image shown on the page, it's necessary to 'place' it using the **PXC_PlaceImage** function. Otherwise the image is considered to be orphaned and will be deleted from the document when saved to (disk) file.

Example (C++)

```
// Example shows how to add the image specified by the HMETAFILE handle

    _PXCDocument*        pdf;
    HENHMETAFILE                hEnhMetaFile;

    ...

    _PXCImage*        pImage = NULL;

    HRESULT hr = PXC_AddEnhMetafile(pdf, hEnhMetaFile, &pImage);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now 'pImage' could be used for further image operations
    ...
```

2.3.3.4 PXC_AddImageA

PXC_AddImageA

Top Previous Next

PXC_AddImageA adds an image, located in a specified file, specified as an ASCII string, to the PDF document.

```
HRESULT  PXC_AddImageA(
    _PXCDocument* pdf,
    LPCSTR filename,
    _PXCImage** image
);
```

Parameters

pdf
[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

filename
[in] *filename* Pointer to a null-terminated ASCII string which contains the full path and name of the required image file.

image
[out] *image* specifies a pointer to the variable of the `_PXCImage*` type, which will represent an image in the *pdf* document.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Use the **PXC_CloseImage** function after completing the image manipulation to reduce memory usage .

Remarks

For the image to be displayed on the page, it is necessary to place it there using the **PXC_PlaceImage** function. If not 'placed' - the image is considered orphaned and will be deleted from the document once written to disk.

Example (C++).

```
// Example shows how to add image specified by it's name to the document

    _PXCDocument*        pdf;

    ...

    _PXCImage*        pImage = NULL;
    LPCSTR                ImageFileName = "C:\\SomeImageName.jpg";

    HRESULT hr = PXC_AddImageA(pdf, ImageFileName, &pImage);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now 'pImage' could be used for further image operations
    ...
```

2.3.3.5 PXC_AddImageExA

PXC_AddImageExA

Top Previous Next

PXC_AddImageExA adds the images, located in the specified file, the name of which is specified as an ASCII string, to the PDF document.

```
HRESULT  PXC_AddImageExA(
    _PXCDocument* pdf,
    LPCSTR filename,
    _PXCImage** image,
    DWORD pages
);
```

Parameters

pdf
[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

filename
[in] *filename* Pointer to a null-terminated ASCII string which contains the full path and name to the image file.

image

[in, out] *image* specifies a pointer to an array of variables for the `_PXCImage*` type, in which objects, representing separate pages in a multipage image, will be returned. See comments.

pages

[in] *pages* specifies the number of elements in the array *image*. Number of elements may be smaller than total the number of the pages in the image.

Return Values

If the function succeeds, the return value is non-negative integer representing the number of pages in the specified image file when *image* is `NULL`, or *pages* is `0`, otherwise the function returns the number of pages which were loaded.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Use the **PXC_CloseImage** function after completing the image manipulation to reduce memory usage.

Remarks

For the image to be shown on the page, it is necessary to 'place' it there using the **PXC_PlaceImage** function. Otherwise the image is considered orphaned and will be deleted from the document when the document is saved.

Example (C++).

```
// Example shows how to add all image pages from multypage image file

   _PXCDocument*        pdf;

   ...

   _PXCImage*        pImages = NULL;
   LPCSTR                ImageFileName = "C:\\SomeImageName.jpg";

   // First retrive number of image pages in the file

   HRESULT hr = PXC_AddImageExA(pdf, ImageFileName, NULL, 0);
   if (IS_DS_FAILED(hr))
   {
       // Handle error
       ...
   }

   DWORD PageCount = hr;

   // Allocate necessary array

   pImages = new _PXCImage[PageCount];

   if (!pImages)
   {
       // Handle error
       ...
```

```
    }

    // Load all image pages from file and store their handles in the
    // allocated array

    hr = PXC_AddImageExA(pdf, ImageFileName, &pImages, PageCount);
    if (IS_DS_FAILED(hr))
    {
        delete[] pImages;
        // Handle error
        ...
    }

    // Now 'pImages' contains all image page handles that may be used in
    // further image operations

    ...
```

2.3.3.6 PXC_AddImageExW

PXC_AddImageExW

Top Previous Next

PXC_AddImageExW adds the images, located in a specified file within a UNICODE string, to the PDF document.

```
HRESULT  PXC_AddImageExW(
    _PXCDocument* pdf,
    LPCWSTR filename,
    _PXCImage** image,
    DWORD pages
);
```

Parameters

pdf
: [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

filename
: [in] *filename* Pointer to a null-terminated UNICODE string containing the full path and name of the image file.

image
: [in, out] *image* specifies a pointer to an array of variables for the `_PXCImage*` type, in which objects, representing separate pages in a multipage image, will be returned. See comments.

pages
: [in] *pages* specifies the number of elements in the array *image*. The Number of elements may be smaller than total number of the pages in the image.

Return Values

If the function succeeds, the return value is non-negative integer representing the number of pages in the

specified image file when *image* is `NULL`, or *pages* is `0`, otherwise the function returns the number of pages which were loaded.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Use the **PXC_CloseImage** function after completing the image manipulation to reduce memory usage.

Remarks

For the image shown on the page, it is necessary to 'place' it there using the **PXC_PlaceImage** function. Otherwise the image is considered orphaned and will be deleted from the file when the created document is saved.

Example (C++).

```
// Example shows how to add all image pages from multypage image file

    _PXCDocument*        pdf;

    ...

    _PXCImage*        pImages = NULL;
    LPCWSTR                ImageFileName = L"C:\\SomeImageName.jpg";

    // First retrive number of image pages in the file

    HRESULT hr = PXC_AddImageExW(pdf, ImageFileName, NULL, 0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    DWORD PageCount = hr;

    // Allocate necessary array

    pImages = new _PXCImage[PageCount];

    if (!pImages)
    {
        // Handle error
        ...
    }

    // Load all image pages from file and store their handles in the
    // allocated array

    hr = PXC_AddImageExW(pdf, ImageFileName, &pImages, PageCount);
    if (IS_DS_FAILED(hr))
    {
        delete[] pImages;
        // Handle error
        ...
    }

    // Now 'pImages' contains all image page handles that may be used in
    // further image operations

    ...
```

2.3.3.7 PXC_AddImageFromHBITMAP

PXC_AddImageFromHBITMAP

Top Previous Next

PXC_AddImageFromHBITMAP adds an image to the PDF document, as represented by the GDI object HBITMAP.

```
HRESULT  PXC_AddImageFromHBITMAP(
    _PXCDocument* pdf,
    HBITMAP hbm,
    HPALETTE hpal,
    _PXCImage** img
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

hbm

[in] *hbm* specifies the Windows HBITMAP GDI object, which represents the image to be added to the document.

hpal

[in] *hpal* specifies the Windows HPALETTE GDI object, representing the image palette. This parameter can be `NULL`

img

[out] *img* specifies a pointer to the variable of the `_PXCImage*` type, which will represent the image in the *pdf* document.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

For the image shown on the page, it is necessary to 'place' it there using the **PXC_PlaceImage** function. Otherwise the image is considered orphaned and will be deleted from the document when the file is saved.

Example (C++).

```
// Example shows how to add image specified by HBITMAP handle

    _PXCDocument*        pdf;
    HBITMAP                hBitmap;
    HPALETTE                hPalette;

    ...

    _PXCImage*        pImage = NULL;

    HRESULT hr = PXC_AddImageFromHBITMAP(pdf, hBitmap, hPalette, &pImage);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now 'pImage' could be used for further image operations
    ...
```

2.3.3.8 PXC_AddImageFromImageXChangePage

PXC_AddImageFromImageXChangePage

Top Previous Next

PXC_AddImageFromImageXChangePage adds an image, represented by the Image XChange page object, to the PDF document.

```
HRESULT  PXC_AddImageFromImageXChangePage(
    _PXCDocument* pdf,
    void* page,
    BOOL bClone,
    _PXCImage** img
);
```

Parameters

pdf
    [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

page
    [in] *page* specifies the Image XChange image page object.

bClone
    [in] *bClone* specifies whether to make a copy of the image.

img
    [out] *img* specifies a pointer to the variable of the `_PXCImage*` type, That will represent the image in the *pdf* document.

Return Values

If the function succeeds, the return value is non-negative integer.

If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Use the **PXC_CloseImage** function after the completion of image manipulation to reduce memory usage .

Remarks

For the image shown on the page, it is necessary to place it there using the **PXC_PlaceImage** function. If not 'placed' - the image is considered orphaned at the time of saving the file and will be deleted.

Example (C++).

```
// Example shows how to add image specified by Image XChange page object

    _PXCDocument*          pdf;
    void*                  pXCPage;

    ...

    _PXCImage*          pImage = NULL;

    HRESULT hr = PXC_AddImageFromImageXChangePage(pdf, pXCPage, TRUE,
&pImage);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now 'pImage' could be used for further image operations
    ...
```

2.3.3.9 PXC_AddImageFromIStream

PXC_AddImageFromIStream

Top Previous Next

PXC_AddImageFromIStream adds the image located in a specified `IStream` object to the PDF document.

```
HRESULT  PXC_AddImageFromIStream(
    _PXCDocument* pdf,
    IStream* pStream,
    _PXCImage** image
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

pStream

[in] *pStream* Pointer to a `IStream` object that contains image.

image

[out] *image* specifies a pointer to the variable of the `_PXCImage*` type, which will represent the image in the *pdf* document.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Use the **PXC_CloseImage** function after completing the image manipulation to reduce memory usage.

Note:

Please note, that if image, specified by *pStream* contains more than one page, only first page of the image will be added.

Remarks

For the image shown on the page, it is necessary to 'place' it there using the **PXC_PlaceImage** function. Otherwise the image is considered orphaned and will be deleted from the file when the created document is saved.

Example (C++).

```
// Example shows how to add image stored in the IStream object

    _PXCDocument*        pdf;
    IStream*                pStream;

    ...

    _PXCImage*        pImage = NULL;

    HRESULT hr = PXC_AddImageFromIStream(pdf, pStream, &pImage);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now 'pImage' could be used for further image operations
    ...
```

2.3.3.10 PXC_AddImageFromIStreamEx

PXC_AddImageFromIStreamEx

PXC_AddImageFromIStreamEx adds the images located in a specified `IStream` object to the PDF document.

```
HRESULT  PXC_AddImageFromIStreamEx(
    _PXCDocument* pdf,
    IStream* pStream,
    _PXCImage** image,
    DWORD pages
);
```

Parameters

pdf
: [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

pStream
: [in] *pStream* Pointer to a `IStream` object that contains image.

image
: [in, out] *image* specifies a pointer to an array of variables for the `_PXCImage*` type, in which objects, representing separate pages in a multipage image, will be returned. If this parameter is `NULL`, function returns number of pages in the specified image.

pages
: [in] *pages* specifies the number of elements in the array *image*. The Number of elements may be smaller than total number of the pages in the image. If this parameter is `0` (`(zero)`) function returns number of pages in the specified image.

Return Values

If the function succeeds, the return value is non-negative integer representing the number of pages in the specified image file when *image* is `NULL`, or *pages* is `0`, otherwise the function returns the number of pages which were loaded.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Use the **PXC_CloseImage** function after completing the image manipulation to reduce memory usage.

Remarks

For the image shown on the page, it is necessary to 'place' it there using the **PXC_PlaceImage** function. Otherwise the image is considered orphaned and will be deleted from the file when the created document is saved.

Example (C++).

```
// Example shows how to add all image pages from multypage image
// stored in IStream object

    _PXCDocument*        pdf;
    IStream*                pStream;
```

```
...

_PXCImage*        pImages = NULL;

// First retrive number of pages in the image

HRESULT hr = PXC_AddImageFromIStreamEx(pdf, pStream, NULL, 0);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

DWORD PageCount = hr;

// Allocate necessary array

pImages = new _PXCImage[PageCount];

if (!pImages)
{
    // Handle error
    ...
}

// Load all image pages from stream and store their handles in the
// allocated array

hr = PXC_AddImageFromIStreamEx(pdf, pStream, &pImages, PageCount);
if (IS_DS_FAILED(hr))
{
    delete[] pImages;
    // Handle error
    ...
}

// Now 'pImages' contains all image page handles that may be used in
// further image operations

...
```

2.3.3.11 PXC_AddImageFromMemory

PXC_AddImageFromMemory

Top Previous Next

PXC_AddImageFromMemory adds an image type to the PDF documents and returns its identifier for later use.

```
HRESULT  PXC_AddImageFromMemory(
    _PXCDocument* pdf,
    PXC_MemImageType type,
    LPBYTE data,
    LONG lDelta,
    const RGBQUAD* pal,
    DWORD palcount,
    DWORD width,
    DWORD height,
    DWORD datasize,
    _PXCImage** image
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

type

[in] Specifies the image type. May be one of the following values:

Constant	Value	Meaning
MemType_1bpp	1	Monochrome image (1 bit per pixel), and *pal* (which must be specified) contains up to 2 entries. Each bit in the image array represents a pixel.
MemType_4bpp	2	The image has a maximum of 16 colors, and *pal* (which must be specified) contains up to 16 entries. Each pixel in the image is represented by a 4-bit index into this palette.
MemType_8bpp	3	The image has a maximum of 256 colors, and the *pal* (which must be valid) contains up to 256 entries. Each pixel in the image is represented by one byte, the value of which is an index in the palette.
MemType_16bpp	4	TrueColor image (16 bits per pixel).
MemType_24bpp	5	TrueColor image (24 bits per pixel).
MemType_32bpp	6	TrueColor image (32 bits per pixel).
MemType_4RLE	7	The same as `MemType_4bpp`, but the image data contained in the *data* array is compressed by the RLE method. For this type of image, if *lDelta* is negative, *data* represents a horizontally flipped image (such as in a BMP file), otherwise it represents a normally oriented image.
MemType_8RLE	8	The same as `MemType_8bpp`, but the image data contained in the *data* array is compressed by the RLE method. For this type of image, if *lDelta* is negative, *data* represents a horizontally flipped image (such as in a BMP file), otherwise it represents a normally oriented image.

data

[in] Specifies image data. This parameter may not be `NULL`. The length of this buffer (specified by *datasize*) must be sufficient to contain all the image data, and depends on the image type, width and height.

lDelta

[in] This parameter specifies the length in bytes of one row of the image. I.e., it specifies what is

required to add it to the *data* pointer to seek to the next image's row. For all image's types, except `MemType_4RLE` and `MemType_8RLE`, this parameter cannot be `0`. If the *data* is a pointer to the last image's row (image is bottom-top type, for example, .BMP), *lDelta* must be negative. This parameter cannot be 0 for all image types, with the exception of `MemType_4RLE` and `MemType_8RLE` types. **PDF-XChange Library** during adding image increments to the *data* pointer with a *lDelta* value to seek to the next image's line. So, in this instance this parameter can be negative.

pal

[in] *pal* specifies a pointer to the array of the image palette. The palette consists of the *palcount* `RGBQUAD` elements. The length of palette depends on the image *type*. For some image types this parameter can be `NULL`.

palcount

[in] Specifies length of the *pal* array. If *pal* is `NULL`, *palcount* must be set to a value of 0.

width

[in] Specifies the **real** width of the image in pixels. This value must be positive.

height

[in] Specifies the **real** height of the image in pixels. This value must be positive.

datasize

[in] Specifies size if the buffer pointed by *data* parameter. This value cannot be **0**.

image

[out] *data* specifies a pointer to the variable of the `_PXCImage*` type, which will represent an image in the *pdf* document.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Use the **PXC_CloseImage** function after completing the image manipulation to reduce memory usage .

Remarks

For the image to be displayed on the page, it is necessary to place it there using the **PXC_PlaceImage** function. If not 'placed' - the image is considered orphaned and will be deleted from the document once written to disk.

Example (C++).

```
// Example shows how to add image stored in memory to the document
// It is assumed that the image is 24 bit and image data passed to the
function is correct

    HRESULT AddImage(_PXCDocument* pdf, _PXCContent* pContent, _PXCImage**
ppImage, LPBYTE ImageData, DWORD DataSize, DWORD Width, DWORD Height)
    {
        // Calculate offset for the next row
        // we assume that each row in ImageData is aligned on DWORD

        LONG delta = (Width * 3) / 4;

        // Add image to file
        return PXC_AddImageFromMemory(pdf, MemType_24bpp, ImageData, delta,
NULL, 0, Width, Height, DataSize, ppImage);
    }
```

2.3.3.12 PXC_AddImagePattern

PXC_AddImagePattern

Top Previous Next

PXC_AddImagePattern adds a pattern, based on a specified image, to the pdf object and returns its identifier.

```
HRESULT  PXC_AddImagePattern(
    _PXCDocument* pdf,
    _PXCImage* image
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image

[in] *image* specifies the image identifier to use as a pattern.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Use the added pattern to fill or to stroke paths. To do so,apply this pattern (using **PXC_ApplyPattern**), and fill (using **PXC_FillPath**), or stroke (using **PXC_StrokePath**) the current path.

While calling functions above, the returned identifier should be used

Example (C++).

```
// Example shows how to add image pattern to the document

    _PXCDocument*         pdf;
    _PXCImage*         pImage;

    ...

    // Add image pattern to the document

    DWORD pat = PXC_AddImagePattern(pdf, pImage);
    if (IS_DS_FAILED(pat))
    {
        // Handle error
        ...
    }

    // Now this pattern could be used for further operations
    ...
```

2.3.3.13 PXC_AddImageW

PXC_AddImageW

Top Previous Next

PXC_AddImageW adds an image, located in the specified file, the name of which is specified as a UNICODE string, to the PDF document.

```
HRESULT  PXC_AddImageW(
    _PXCDocument* pdf,
    LPCWSTR filename,
    _PXCImage** image
);
```

Parameters

pdf
: [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

filename
: [in] *filename* Pointer to a null-terminated UNICODE string containing the full path and name to the image file.

image
: [out] *image* specifies a pointer to the variable of the `_PXCImage*` type, which will represent the image in the *pdf* document.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Use the **PXC_CloseImage** function after completing the image manipulation to reduce memory usage.

Remarks

For the image shown on the page, it is necessary to place it there by using **PXC_PlaceImage** function. If not done - the image will become orphaned within the document and deleted when written to disk.

Example (C++).

```
// Example shows how to add image specified by it's name to the document

    _PXCDocument*        pdf;
```

```
...

_PXCImage*          pImage = NULL;
LPCWSTR                  ImageFileName = L"C:\\SomeImageName.jpg";

HRESULT hr = PXC_AddImageW(pdf, ImageFileName, &pImage);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

// Now 'pImage' could be used for further image operations
...
```

2.3.3.14 PXC_AddStdMetafile

PXC_AddStdMetafile

Top Previous Next

PXC_AddStdMetafile adds an image, defined by the metafile (HMETAFILE), to the PDF document.

```
HRESULT  PXC_AddStdMetafile(
    _PXCDocument* pdf,
    HMETAFILE metafile,
    _PXCImage** image
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

metafile

[in] *metafile* specifies the Windows metafile object, which represents an image that will be added to the document.

image

[out] *image* specifies pointer to a variable for the `_PXCImage*` type, representing the image in *pdf* document.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

For the image to be shown on the page, it's necessary to place it there using the **PXC_PlaceImage** function. An image not 'placed' is considered orphaned and deleted when the document is saved to a (disk) file.

Example (C++).

```
// Example shows how to add image specified by HMETAFILE handle

    _PXCDocument*        pdf;
    HMETAFILE                hMetaFile;

    ...

    _PXCImage*        pImage = NULL;

    HRESULT hr = PXC_AddStdMetafile(pdf, hMetaFile, &pImage);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now 'pImage' could be used for further image operations
    ...
```

2.3.3.15 PXC_AddU3DStream

PXC_AddU3DStream

Top Previous Next

PXC_AddU3DStream adds a *U3D stream* to the PDF document and returns its identifier for later use.

For more information about U3D content with a PDF document please refer to the PDF Specification V1.5.

N.B. This function makes no checks to validate the contents of the specified buffer in terms of the U3D stream!

```
HRESULT  PXC_AddU3DStream(
    _PXCDocument* pdf,
    LPBYTE lpBuf,
    UINT nBufSize
);
```

Parameters

pdf
  [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

lpBuf
  [in] Pointer to the buffer where the U3D stream is located. This buffer may be freed after the function call.

nBufSize
  [in] Specifies the size in bytes of the buffer specified by *lpBuf*.

Return Values

If the function succeeds, the return value represents the U3D stream handle, which may then be used with other relevant library functionality.
If the function fails, the return value is an **error code**.

To determine if the function was successful use the defined macro's as described here: Error Handling.

Remarks

For the U3D object to be displayed on the page, it is necessary to add a 3D annotation to the page using the PXC_Add3DAnnotationA (or PXC_Add3DAnnotationW) function. If not 'placed' - the image is considered orphaned and will be deleted from the document once written to disk.

Example (C++).

```
// Example shows how to add U3D stream from file to the PDF document

    LONG Add3DStreamFromFile(_PXCDocument* pdf, LPCSTR FileName)
    {
        // Load U3D stream from specified file
        // and store it into buffer

        HANDLE f = CreateFile(FileName, GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (f == INVALID_HANDLE_VALUE)
            return GetLastError();

        // The length of the file

        DWORD fsz = GetFileSize(f, NULL);

        // Allocate buffer for stream data

        LPBYTE buf = new BYTE[fsz];
        DWORD numreaded;

        // Read stream data into the buffer

        ReadFile(f, buf, fsz, &numreaded, NULL);
        CloseHandle(f);

        // Add U3D stream to the document

        LONG u3d = PXC_AddU3DStream(pdf, buf, fsz);

        return u3d;
    }
```

2.3.3.16 PXC_AddU3DViewToStream

PXC_AddU3DViewToStream

Top Previous Next

PXC_AddU3DViewToStream adds a 3D view for the U3D stream added to the PDF document. A U3D stream may have an unlimited view's associated.

For more information about U3D content within a PDF please refer to the PDF Specification V1.5.

```
HRESULT  PXC_AddU3DViewToStream(
    _PXCDocument* pdf,
    DWORD dwU3DStream,
    const PXC_3DView* view
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

dwU3DStream

[in] Specifies the handle of the U3D stream previously added to the *pdf* document using the function **PXC_AddU3DStream**.

view

[in] Pointer to a **PXC_3DView** structure that describes attributes of the 3D view. This parameter may not be `NULL`.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// U3D stream handle

   LONG u3d;

   // document's handle

   _PXCDocument*        pdf;

   ...

   // Fill 3D View structure

   PXC_3DView view = {sizeof(PXC_3DView)};
   lstrcpyW(view.m_ExtName, L"Default");
   view.m_CO = 1300.32;
   view.m_FOV = 30.0;
   view.m_C2W[0] = view.m_C2W[7] = 1.0;
   view.m_C2W[5] = -1.0;
   view.m_C2W[9] = -0.0893402;
   view.m_C2W[10] = -1300.32;
   view.m_C2W[11] = 20.335;

   // Add 3D View to stream
```

```
    HRESULT hr = PXC_AddU3DViewToStream(pdf, u3d, &view);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // done
```

2.3.3.17 PXC_CloseImage

PXC_CloseImage

Top Previous Next

PXC_CloseImage function "closes" the specified image. It is recommended to use this function after all image manipulations to reduce memory usage.

```
HRESULT  PXC_CloseImage(
    _PXCDocument* pdf,
    _PXCImage* image
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image

[in] *image* specifies the pointer to the image which should be *Closed*.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

It is only possible to place the closed image on the page (see **PXC_PlaceImage**). All other operations related to an image will return an error.

Note:

On calling the **PXC_PlaceImage** function, the **PXC_CloseImage** function will be called automatically if the specified image is not closed.

Example (C++).

```
// Example shows how to 'close' the image to reduce memory usage

    _PXCDocument*        pdf;
    _PXCImage*        pImage;
```

```
...

// When no operations with image will not be performed
// (except, PXC_PlaceImage) it is recomended to 'close' the image

HRESULT hr = PXC_CloseImage(pdf, pImage);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

// Now 'pImage' could only be placed by 'PXC_PlaceImage' function
...
```

2.3.3.18 PXC_CropImage

PXC_CropImage

Top Previous Next

PXC_CropImage crops the image previously added to the PDF document.

```
HRESULT  PXC_CropImage(
    const _PXCDocument* pdf,
    _PXCImage* image,
    LPCRECT croprect
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image

[in] *image* specifies the image identifier.

croprect

[in] *croprect* pointer to the variable for the `RECT` type, which specifies the rectangle.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to crop image to specified rectangle
```

```
    _PXCDocument*        pdf;
    _PXCImage*        image;

    ...

    RECT                CropRect;

    // Set cropping rectangle to be 100 x 200 pixels

    CropRect.left        = 0;
    CropRect.top        = 0;
    CropRect.right        = 100;
    CropRect.bottom        = 200;

    // Crop image

    HRESULT hr = PXC_CropImage(pdf, image, &CropRect);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

2.3.3.19 PXC_GetImageColors

PXC_GetImageColors

Top Previous Next

PXC_GetImageColors retrieves the number of colors present within the image.

```
HRESULT  PXC_GetImageColors(
    const _PXCDocument* pdf,
    const _PXCImage* image
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image

[in] *image* specifies the image identifier.

Return Values

If the function succeeds, it returns the number of colors.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Please note, that when *image* represents handle of the metafile image (returned, for example, by function **PXC_AddEnhMetafile**), funtion will return error code.

Example (C++).

```
    // Example shows how to get the number of image colors

_PXCDocument*         pdf;
_PXCImage*         image;

...

DWORD                 ColorNumber = 0;

ColorNumber = PXC_GetImageColors(pdf, image);
if (IS_DS_FAILED(ColorNumber))
{
    // Handle error
    ...
}
...
```

2.3.3.20 PXC_GetImageDimension

PXC_GetImageDimension

Top Previous Next

PXC_GetImageDimension returns the height and width of the image, previously added to the PDF document.

```
HRESULT  PXC_GetImageDimension(
    const _PXCDocument* pdf,
    const _PXCImage* image,
    double* width,
    double* height
);
```

Parameters

pdf
: [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image
: [in] *image* specifies the image identifier.

width
: [out] *width* specifies the address of the variable to receive the image's width in points.

height
: [out] *height* specifies the address of the variable to receive the image's height in points.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    // Example shows how to get image dimension (width and height)
    information

_PXCDocument*        pdf;
_PXCImage*          image;

...

double width, height;

HRESULT hr = PXC_GetImageDimension(pdf, image, &width, &height);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}
...
```

2.3.3.21 PXC_GetImageDPI

PXC_GetImageDPI

Top Previous Next

PXC_GetImageDPI retrieves an images's resolution in DPI (Dots Per inch)

```
HRESULT  PXC_GetImageDPI(
    const _PXCDocument* pdf,
    const _PXCImage* image,
    DWORD* xdpi,
    DWORD* ydpi
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image

[in] *image* specifies the image identifier.

xdpi

[out] *xdpi* specifies the pointer to the variable which recieves the X (horizontal) resolution of the image.

ydpi

[out] *ydpi* specifies the pointer to the variable which recieves the `Y` (vertical) resolution of the image.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    // Example shows how to get image DPI information

_PXCDocument*        pdf;
_PXCImage*          image;

...

DWORD Xdpi, Ydpi;

HRESULT hr = PXC_GetImageDPI(pdf, image, &Xdpi, &Ydpi);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}
...
```

2.3.3.22 PXC_MakeImageGrayscale

PXC_MakeImageGrayscale

Top Previous Next

PXC_MakeImageGrayscale converts the specified image to grayscale.

```
HRESULT  PXC_MakeImageGrayscale(
    _PXCDocument* pdf,
    _PXCImage* image
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image

[in] *image* specifies the image identifier.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to convert an image to grayscale

    _PXCDocument*        pdf;
    _PXCImage*        image;

    ...

    HRESULT hr = PXC_MakeImageGrayscale(pdf, image);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

2.3.3.23 PXC_MarkImageAsMask

PXC_MarkImageAsMask

Top Previous Next

PXC_MarkImageAsMask marks the image as a mask.

```
HRESULT  PXC_MarkImageAsMask(
    const _PXCDocument* pdf,
    _PXCImage* image,
    BOOL bMask
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by function **PXC_NewDocument**.

image

[in] *image* specifies the image identifier.

bMask

[in] If *bMask* is `TRUE` the image will be treated as mask.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

If the image is defined as a Mask it will act as a masking image that fills the entire page using the current **FillColor**. Otherwise, while placement occurs an image will be treated as a standard image.

Example (C++).

```
// Example shows how to mark an image as a mask

    _PXCDocument*        pdf;
    _PXCImage*        image;

    ...

    HRESULT hr = PXC_MarkImageAsMask(pdf, image, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

2.3.3.24 PXC_PlaceImage

PXC_PlaceImage

Top Previous Next

PXC_PlaceImage places the image on the specified page at the specified coordinates. This function automatically calls **PXC_CloseImage** when completed.

```
HRESULT  PXC_PlaceImage(
    _PXCContent* content,
    _PXCImage* image,
    double x,
    double y,
    double width,
    double height
);
```

Parameters

content
[in] Parameter *content* specifies the identifier for the page content of where the image should be placed.

image
[in] *image* specifies the image identifier.

x
[in] *x* specifies the X coordinate on the page where left top corner of the image should be located.

y
[in] *y* specifies the Y coordinate of the page where left top corner of the image should be located.

width
[in] *width* specifies the width of the image in points.

height
[in] *height* specifies the height of the image in points.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to add and place an image specified by its name to the
document
// keeping original size of the image

    HRESULT PlaceImage(_PXCDocument* pdf, _PXCContent* pContent, LPCWSTR
ImageFileName)
    {
        _PXCImage*          pImage = NULL;
        HRESULT          hr = DS_OK;

        // Add image from file

        hr = PXC_AddImageW(pdf, ImageFileName, &pImage);
        if (IS_DS_FAILED(hr))
        {
            return hr;
        }

        // Retrive image width and height

        double width, height;

        hr = PXC_GetImageDimension(pdf, pImage, &width, &height);
        if (IS_DS_FAILED(hr))
        {
            return hr;
        }

        // Place image

        return PXC_PlaceImage(pContent, pImage, 0, height, width, height);
    }
```

2.3.3.25 PXC_ReduceImageColors

PXC_ReduceImageColors

Top Previous Next

PXC_ReduceImageColors optimizes an image, previously added to the pdf document, by reducing the number of colors within the image.

```
HRESULT  PXC_ReduceImageColors(
    const _PXCDocument* pdf,
    _PXCImage* image,
    DWORD depth,
    BOOL bGrayscale,
    BOOL dither,
    BOOL optimal
);
```

Parameters

pdf
: [in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image
: [in] *image* specifies the image identifier.

depth
: [in] Specifies the new color count in the image. This value must be in the range `2` to `65536`.

bGrayscale
: [in] If the *bGrayscale* parameter is `TRUE`, during reduction the image will be converted to grayscale.

dither
: [in] If the *dither* parameter is `TRUE`, dithering method will be used during color reduction.

optimal
: [in] This parameter is reserved and must be `0`.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to reduce the number of colors in the image to 256

    _PXCDocument*        pdf;
    _PXCImage*          image;

    ...

    HRESULT hr = PXC_ReduceImageColors(pdf, image, 256, FALSE, TRUE, 0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

2.3.3.26 PXC_ScaleImage

PXC_ScaleImage

PXC_ScaleImage scales an image previously added to the document.

```
HRESULT  PXC_ScaleImage(
    const _PXCDocument* pdf,
    _PXCImage* image,
    DWORD width,
    DWORD height,
    BOOL bProp,
    DWORD method
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image

[in] *image* specifies the image identifier.

width

[in] *width* specifies the new width of the image. Width is specified in `pixels`.

height

[in] *height* specifies the new height of the image. Height is specified in `pixels`.

bProp

[in] If this parameter is `TRUE`, the image will be scaled proportionally to fit into a rectangle which has `width` and `height` dimensions.

method

[in] *method* specifies the scaling method to be used to scale the image. This parameter can be any one of the following values:

Constant	Value	Definition
ScaleImage_Linear	0	The Image will be scaled using linear filtration. This has the benefit of being fast in processing terms, but quality may be compromised.
ScaleImage_Bilinear	1	The Image will be scaled using bilinear filtration. This is slower than the `ScaleMethod_Linear` method, but produces a higher quality result.
ScaleImage_Bicubic	2	The Image will be scaled using bicubic filtration. This method is the slowest but it yields the best results in terms of quality.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

If the specified image has been previously closed, this function will fail.

Example (C++).

```
// Example shows how to scale an image to fit a specified width and height
// while keeping the original proportions

    _PXCDocument*        pdf;
    _PXCImage*        image;

    ...

    // Scale image

    HRESULT hr = PXC_ScaleImage(pdf, image, 100, 200, TRUE,
ScaleImage_Bicubic);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

2.3.3.27 PXC_SetImageMask

PXC_SetImageMask

Top Previous Next

PXC_SetImageMask specifies one image to be used as the mask for another.

```
HRESULT  PXC_SetImageMask(
    const _PXCDocument* pdf,
    _PXCImage* image,
    _PXCImage* mask
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image

[in] *image* specifies the image identifier for the image to be masked as previously returned from one of the valid AddImages functions.

mask

[in] *mask* specifies the identifier of the image which will be used as the mask for another *image*. This image must be a monochrome 1bpp image, and must have the same dimensions as the `image` to be masked, otherwise the function call will fail.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.

To determine if the function was successful use the defined macro's as described here: Error Handling.

Example (C++).

```
// Example shows how to set one image as mask for the other image

    _PXCDocument*        pdf;
    _PXCImage*        image;
    _PXCImage*        imageMask;

    ...

    HRESULT hr = PXC_SetImageMask(pdf, image, imageMask);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

2.3.3.28 PXC_SetImageTransColor

PXC_SetImageTransColor

Top Previous Next

PXC_SetImageTransColor specifies the transparent color for the specified image.

```
HRESULT  PXC_SetImageTransColor(
    const _PXCDocument* pdf,
    _PXCImage* image,
    COLORREF color
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

image

[in] *image* specifies the image identifier.

color

[in] *color* specifies the transparent color value.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: Error Handling.

Example (C++).

```
// Example shows how to set red color as transparent for image

    _PXCDocument*        pdf;
    _PXCImage*          image;

    ...

    HRESULT hr = PXC_SetImageTransColor(pdf, image, RGB(255, 0, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

2.3.4 Drawing

Drawing

Top Previous Next

Drawing functions add discrete graphical objects, such as shapes and colors, to the PDF content.

The Path Construction And Drawing topic looks at how drawing is done using the **graphical path** paradigm.

The Graphics State Stack Operations topic explains how to save and restore graphical parameters using the **PXC_GetStateLevel**, **PXC_RestoreState**, and **PXC_SaveState** functions.

The graphical functions for PDF content generation are:

- **PXC_ApplyPattern**
- **PXC_Arc**
- **PXC_ArcN**
- **PXC_Chord**
- **PXC_ChordEx**
- **PXC_Circle**
- **PXC_ClipPath**
- **PXC_ClosePath**
- **PXC_CurveTo**
- **PXC_Ellipse**
- **PXC_EllipseArc**
- **PXC_EllipseArcEx**
- **PXC_EndPath**
- **PXC_FillPath**
- **PXC_GetContentDC**
- **PXC_GetLineInfo**
- **PXC_GradientFill**
- **PXC_LineTo**
- **PXC_MoveTo**
- **PXC_NoDash**
- **PXC_Pie**

- PXC_PieEx
- PXC_PolyCurve
- PXC_Polygon
- PXC_Rect
- PXC_ReleaseContentDC
- PXC_SetBlendMode
- PXC_SetDash
- PXC_SetDrawingColor
- PXC_SetDrawingGray
- PXC_SetFillColor
- PXC_SetFillGray
- PXC_SetFlat
- PXC_SetLineCap
- PXC_SetLineJoin
- PXC_SetLineWidth
- PXC_SetMiterLimit
- PXC_SetPolyDash
- PXC_SetStrokeAdjust
- PXC_SetStrokeColor
- PXC_SetStrokeGray
- PXC_SetTransparency
- PXC_StrokePath

2.3.4.1 Path Construction And Drawing

Path Construction And Drawing

Top Previous Next

Paths define shapes, trajectories, and regions of all sorts. They are used to draw lines, define the shape of filled areas and specify boundaries for clipping other graphics. The graphics state includes a current *clipping path* that defines the clipping boundary for the current page. At the beginning of each page, the clipping path is initialized to include the entire page.

A path may be composed of both straight and curved line segments, which may connect to one another or may be disconnected. A pair of segments are said to *connect* only if they are defined consecutively, with the second segment starting where the first one ends. Thus the order in which the segments of a path are defined is significant. Nonconsecutive segments that meet or intersect fortuitously are not considered to connect.

A path is made up of one or more disconnected *subpaths*, each comprising a sequence of connected segments. The topology of the path is unrestricted: it may be concave or convex, may contain multiple subpaths representing disjointed areas, and may intersect itself in arbitrary ways. Thes function PXC_EndPath, explicitly connects the end of a subpath back to its starting point; such a subpath is said to be *closed*. A subpath that has not been explicitly closed is *open*.

Path Construction

A page description begins with an empty path and builds up its definition by invoking one or more path construction operators to add segments to it. The path construction functions may be invoked in any sequence, but the first invoked always begins a new subpath. The path definition concludes with the application of a path painting function such as **PXC_StrokePath** or **PXC_FillPath**; this may optionally be preceded by one of the clipping path functions, for example **PXC_ClipPath**. Note that the path construction functions in themselves do not place any content on the page; only painting functions have this capability. A path definition is not complete until a path painting function has been applied.

The path currently under construction is called the *current path*. In PDF document construction, the current path is *not* part of the graphics state and is not saved and restored along with the other graphics state parameters. Once a path has been painted, it is no longer defined; there is therefore no current path until a new one is started with the use of the path construction functions available.

The trailing endpoint of the segment most recently added to the current path is referred to as the *current point*.

Path-Painting Functions

The path-painting functions terminate a path object, causing it to be painted on the page in the manner that the operations specify. The principal path painting functions are **PXC_StrokePath** (for stroking) and **PXC_FillPath** (for filling). These functions have parameters which allow for the combined stroking and filling in a single operation or the application of different rules for determining properties of the area to be filled.

2.3.4.2 Graphics State Stack Operations

Graphics State Stack Operations

Top Previous Next

A well-structured PDF document typically contains many graphical elements that are essentially independent of each other and sometimes nested to multiple levels. The *graphics state stack* allows these elements to make local changes to the graphics state without disturbing the graphics state of the surrounding environment. The stack is a `LIFO` (last in, first out) data structure in which the contents of the graphics state can be saved and later restored using the following functions:

- **PXC_SaveState** pushes a copy of the entire graphic state onto the stack;
- **PXC_RestoreState** restores the entire graphic state to its former value by popping it from the stack.

Example (C++).

```
// Example shows how to save the current graphics state, perform some changes,
// and then return to the original graphics state

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_SaveState(pContent);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Do some changes of CS, drawing etc.

    ...

    // Then return to original one

    hr = PXC_RestoreState(pContent);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now CS is original one
    ...
```

2.3.4.2.1 PXC_GetStateLevel

PXC_GetStateLevel

Top Previous Next

PXC_GetStateLevel returns the current save level of the *graphics state stack*. For more information please see functions **PXC_SaveState**, **PXC_RestoreState**.

```
HRESULT  PXC_GetStateLevel(
    const _PXCContent* content,
    LONG* level
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

level

[out] Pointer to a `LONG` variable that receives the current save level of the graphics state stack.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to retrieve current state level

    _PXCContent*        pContent;

    ...

    LONG CurLevel = 0;

    HRESULT hr = PXC_GetStateLevel(pContent, &CurLevel);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.2.2 PXC_RestoreState

PXC_RestoreState

Top Previous Next

PXC_RestoreState restrores the entire graphic state to its former value by poping it from the stack.

```
HRESULT  PXC_RestoreState(
    _PXCContent* content
);
```

Parameters

content
[in] Parameter *content* specifies the identifier of page content to which the function will be applied.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.3.4.2.3 PXC_SaveState

PXC_SaveState

Top Previous Next

PXC_SaveState pushes a copy of the entire graphic state onto the stack;

```
HRESULT  PXC_SaveState(
    _PXCContent* content
);
```

Parameters

content
[in] Parameter *content* specifies identifier of the page content to which the function will be applied.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

The **PXC_SaveState** function may be used any number of times to save any number of page content's state.

2.3.4.3 PXC_ApplyPattern

PXC_ApplyPattern

Top Previous Next

PXC_ApplyPattern applies the specified pattern before path filling or stroking takes place.

```
HRESULT  PXC_ApplyPattern(
    _PXCContent* content,
    DWORD patID,
    BOOL bForStroke,
    COLORREF patColor
);
```

Parameters

content
[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

patID
[in] *patID* specifies the pattern identifier.

bForStroke
[in] If *bForStroke* is `TRUE`, the specified pattern will be used for stroking operations (see **PXC_StrokePath**). Otherwise it will be used for filling (see **PXC_FillPath**).

patColor
[in] *patColor* specifies the colorto be used to draw the pattern. This parameter is used only for hatched patterns, and is ignored for patterns based on images.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Pattern identifier *patID* should be obtained by calling **PXC_AddImagePattern**

Example (C++).

```
// Example shows haw to apply add and use pattern

    _PXCContent*         pContent;
    _PXCDocument*         pdf;

    ...

    // Add horizontal hatch pattern to the document

    DWORD pat = PXC_AddHatchPattern(pdf, HatchType_Horizontal);
    if (IS_DS_FAILED(pat))
    {
        // Handle error
        ...
    }
```

```
// Use this pattern for filling operations

HRESULT hr = PXC_ApplyPattern(pContent, pat, FALSE, RGB(128, 128, 0));
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

...
```

2.3.4.4 PXC_Arc

PXC_Arc

Top Previous Next

PXC_Arc adds a circle's arc to the current path.

```
HRESULT  PXC_Arc(
    _PXCContent* content,
    LPCPXC_PointF center,
    double radius,
    double alpha,
    double beta
);
```

Parameters

content
[in] Parameter *content* specifies the identifier of page content to which the function will be applied.

center
[in] Pointer to **PXC_PointF** structure, containing coordinates of the center.

radius
[in] Specifies the radius of arc.

alpha
[in] Specifies the starting angle in degrees.

beta
[in] Specifies the ending angle in degrees.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

An Arc is drawn by default from *alpha* to *beta*. If *alpha* is less than *beta*, the arc will be drawn counterclockwise; if *alpha* is greater than *beta*, the arc will be drawn clockwise, from *alpha* to *beta*.

Example (C++).

```
// Example shows how to draw circle's arc with the blue color

    _PXCContent*        pContent;

    ...

    // Set stroke color to blue

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 0, 255));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Add circle's arc

    PXC_PointF ptCenter;

    ptCenter.x = 100.0;
    ptCenter.y = 200.0;

    hr = PXC_Arc(pContent, &ptCenter, 50, -90.0, +90.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // And stroke it

    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.5 PXC_ArcN

PXC_ArcN

Top Previous Next

PXC_ArcN adds a circle's arc to the current path similar to the **PXC_Arc** function, but in the opposite direction, from *beta* to *alpha*. This function is equivalent to **PXC_Arc**(*pdf*, *center*, *r*, *beta*, *alpha*).

```
HRESULT  PXC_ArcN(
    _PXCContent* content,
    LPCPXC_PointF center,
    double radius,
    double alpha,
    double beta
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

center
: [in] Pointer to **PXC_PointF** which contains (X,Y) coordinates for the center.

radius
: [in] Specifies the radius of arc.

alpha
: [in] Specifies the starting angle in degrees.

beta
: [in] Specifies ending angle in degrees.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to draw circle's arc with the blue color

   _PXCContent*        pContent;

   ...

   // Set stroke color to blue

   HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 0, 255));
   if (IS_DS_FAILED(hr))
   {
       // Handle error
       ...
   }

   // Add circle's arc

   PXC_PointF ptCenter;

   ptCenter.x = 100.0;
   ptCenter.y = 200.0;
```

```
    hr = PXC_ArcN(pContent, &ptCenter, 50, -45.0, +45.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // And stroke it

    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.6 PXC_Chord

PXC_Chord

Top Previous Next

PXC_Chord adds to the current path a chord (a region bounded by the intersection of an ellipse and a line segment, called a secant).

```
HRESULT  PXC_Chord(
    _PXCContent* content,
    LPCPXC_RectF rect,
    double alpha,
    double beta
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which function will be applied.

rect

[in] Pointer to **PXC_RectF** structure containing the coordinates of the bounding rectangle.

alpha

[in] Specifies the starting angle in degrees.

beta

[in] Specifies the ending angle in degrees.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to draw chord with the green color

    _PXCContent*        pContent;

    ...

    // Set stroke color to green

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Add chord

    PXC_RectF rect;

    rect.left = 20.0;
    rect.top = 200.0;
    rect.right = 320.0;
    rect.bottom = 500.0;


    hr = PXC_Chord(pContent, &rect, -90.0, +90.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // And stroke it

    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.7 PXC_ChordEx

PXC_ChordEx

Top Previous Next

PXC_ChordEx adds to the current path a chord (a region bounded by the intersection of an ellipse and a line segment, called a secant).

Similar to the **PXC_Chord** function, using a different method to specify angles.

```
HRESULT  PXC_ChordEx(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCPXC_PointF pnt1,
    LPCPXC_PointF pnt2
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect

[in] Pointer to **PXC_RectF** containing the coordinates of the bounding rectangle.

pnt1

[in] Pointer to **PXC_PointF** containing the coordinates of the end point of the radial line defining the start point of the arc.

pnt2

[in] Pointer to **PXC_PointF** containing the coordinates of the end point of the radial line defining the end point of the arc.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to draw chord with the green color

    _PXCContent*         pContent;

    ...

    // Set stroke color to green
```

```
HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

// Add chord

PXC_RectF rect;

rect.left = 20.0;
rect.top = 200.0;
rect.right = 320.0;
rect.bottom = 500.0;

PXC_PointF        ptStart;
PXC_PointF        ptEnd;

ptStart.x = 0.0;
ptStart.y = 0.0;

ptEnd.x = 150.0;
ptEnd.y = 450.0;


hr = PXC_ChordEx(pContent, &rect, &ptStart, &ptEnd);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

// And stroke it

hr = PXC_StrokePath(pContent, TRUE);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

...
```

2.3.4.8 PXC_Circle

PXC_Circle

Top Previous Next

PXC_Circle adds a circle to the current path.

```
HRESULT  PXC_Circle(
    _PXCContent* content,
    LPCPXC_PointF center,
    double r
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

center

[in] Pointer to **PXC_PointF** structure containing the coordinates of the center of the circle.

r

[in] Specifies the radius of the circle.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to draw circle with the green color

    _PXCContent*        pContent;

    ...

    // Set stroke color to green

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Add circle

    PXC_PointF ptCenter;

    ptCenter.x = 100.0;
    ptCenter.y = 200.0;

    hr = PXC_Circle(pContent, &ptCenter, 20.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // And stroke it

    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.9 PXC_ClipPath

PXC_ClipPath

Top Previous Next

PXC_ClipPath specifies that the current path will be used for clipping.

```
HRESULT  PXC_ClipPath(
    _PXCContent* content,
    PXC_FillRule mode
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

mode

[in] Specifies the type of action for the path. The possible actions are enumerated in `PXC_FillRule`. *(See comments for possible values)*

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Possible values of `PXC_FillRule` for are:

Fill Mode	Value	Meaning
`FillRule_Winding`	0	Fill using winding mode (fills any region with a nonzero winding value).
`FillRule_EvenOdd`	1	Fill using even-odd mode (fills the area between odd-numbered and even-numbered polygon sides on each scan line).

Example (C++).
```
// Example shows how to specify that current path will be used for clipping

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_ClipPath(pContent, FillRule_Winding);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.10 PXC_ClosePath

PXC_ClosePath

Top Previous Next

PXC_ClosePath closes the current path. You can fill (**PXC_FillPath**), stroke (**PXC_StrokePath**) the closed path, or use it for clipping (**PXC_ClipPath**).

```
HRESULT  PXC_ClosePath(
     _PXCContent* content
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error**

Handling.

Example (C++).

```
// Example shows how to close path

    _PXCContent*         pContent;

    ...

    HRESULT hr = PXC_ClosePath(pContent);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.11 PXC_CurveTo

PXC_CurveTo

Top Previous Next

PXC_CurveTo adds a segment of a Bezier curve to the current path. A Bezier curve uses the current point (See **PXC_MoveTo**) as its first point. After adding the curve, the end of the curve (point with coordinates `pnt3`) will become the current point.

```
HRESULT  PXC_CurveTo(
    _PXCContent* content,
    LPCPXC_PointF pnt1,
    LPCPXC_PointF pnt2,
    LPCPXC_PointF pnt3
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

pnt1

[in] Pointer to **PXC_PointF** specifies the first control point of the Bezier curve.

pnt2

[in] Pointer to **PXC_PointF** specifies the second control point of the Bezier curve.

pnt3

[in] Pointer to **PXC_PointF** specifies the end point of the Bezier curve.

Return Values

If the function succeeds, the return value is non-negative integer.

If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to draw Bezier curve with the green color

    _PXCContent*        pContent;

    ...

    // Set stroke color to green

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Add Bezier curve

    PXC_PointF pt1;
    PXC_PointF pt2;
    PXC_PointF pt3;

    pt1.x = 25.0;
    pt1.y = 25.0;

    pt2.x = 150.0;
    pt2.y = 450.0;

    pt3.x = 150.0;
    pt3.y = 450.0;

    hr = PXC_MoveTo(pContent, 100, 100);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    hr = PXC_CurveTo(pContent, &pt1, &pt2, &pt3);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // And stroke it
    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.12 PXC_Ellipse

PXC_Ellipse

Top Previous Next

PXC_Ellipse adds an ellipse to the current path.

```
HRESULT  PXC_Ellipse(
    _PXCContent* content,
    LPCPXC_RectF rect
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect

[in] Pointer to **PXC_RectF** which contains the coordinates of the enclosing rectangle.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to draw a green ellipse

    _PXCContent*        pContent;

    ...

    // Set stroke color to green

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Add ellipse

    PXC_RectF rect;

    rect.left = 20.0;
    rect.top = 200.0;
    rect.right = 320.0;
    rect.bottom = 500.0;


    hr = PXC_Ellipse(pContent, &rect);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // And stroke it

    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.13 PXC_EllipseArc

PXC_EllipseArc

Top Previous Next

The **PXC_EllipseArc** adds an elliptical arc to the current path.

```
HRESULT  PXC_EllipseArc(
    _PXCContent* content,
    LPCPXC_RectF rect,
    double alpha,
    double beta
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect

[in] Pointer to **PXC_RectF** which contains the coordinates of the enclosing rectangle.

alpha

[in] Specifies the starting angle in degrees.

beta

[in] Specifies the ending angle in degrees.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

The elliptical arc is drawn from *alpha* to *beta*. If *alpha* is less than *beta*, the arc is drawn counterclockwise; if *alpha* greater than *beta*, the arc is drawn clockwise, from *alpha* to *beta*.

Example (C++).

```
// Example shows how to draw a green elliptical arc

    _PXCContent*        pContent;

    ...

    // Set stroke color to green

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Add elliptical arc

    PXC_RectF rect;

    rect.left = 20.0;
    rect.top = 200.0;
    rect.right = 320.0;
    rect.bottom = 500.0;
    hr = PXC_EllipseArc(pContent, &rect, -90.0, +90.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // And stroke it

    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.14 PXC_EllipseArcEx

PXC_EllipseArcEx

Top Previous Next

PXC_EllipseArcEx adds an elliptical arc to the current path. It is similar to **PXC_EllipseArc**, but uses a different method of specifying angles.

```
HRESULT  PXC_EllipseArcEx(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCPXC_PointF pnt1,
    LPCPXC_PointF pnt2
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect

[in] Pointer to **PXC_RectF** which contains the coordinates of rectangle.

pnt1

[in] Pointer to **PXC_PointF** which contains the coordinates of the ending point of the radial line defining the starting point of the arc.

pnt2

[in] Pointer to **PXC_PointF** which contains the coordinates of the ending point of the radial line defining the ending point of the arc.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to draw a green elliptical arc

   _PXCContent*        pContent;

   ...

   // Set stroke color to green

   HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
   if (IS_DS_FAILED(hr))
   {
       // Handle error
       ...
   }

   // Add elliptical arc

   PXC_RectF rect;

   rect.left = 20.0;
   rect.top = 200.0;
   rect.right = 320.0;
   rect.bottom = 500.0;

   PXC_PointF        ptStart;
   PXC_PointF        ptEnd;

   ptStart.x = 0.0;
   ptStart.y = 0.0;

   ptEnd.x = 150.0;
   ptEnd.y = 450.0;


   hr = PXC_EllipseArcEx(pContent, &rect, &ptStart, &ptEnd);
   if (IS_DS_FAILED(hr))
   {
       // Handle error
       ...
   }
```

```
    // And stroke it

    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.15 PXC_EndPath

PXC_EndPath

Top Previous Next

The **PXC_EndPath** ends the current path.

```
HRESULT  PXC_EndPath(
    _PXCContent* content
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This is a "no-op" path painting function and causes no marks to be placed on the page and may be used with a clipping path function to establish a new clipping path. Therefore, after a path has been constructed, the sequence:

```
PXC_ClipPath(content, PXC_FillRule);
PXC_EndPath(content);
```

will intersect that path with the current clipping path to establish a new clipping path.

Example (C++).

```
// Example shows how to end path

    _PXCContent*        pContent;
    ...

    HRESULT hr = PXC_EndPath(pContent);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.16 PXC_FillPath

PXC_FillPath

Top Previous Next

PXC_FillPath fills the current path with the current fill color, or pattern.

```
HRESULT  PXC_FillPath(
    _PXCContent* content,
    BOOL bClose,
    BOOL bStroke,
    PXC_FillRule mode
);
```

Parameters

content

[in] Parameter *content* specifies identifier of page content to which function will be applied.

bClose

[in] Specifies close or not the current path before fill.

bStroke

[in] Specifies stoke or not the current path. If `TRUE` the path will be stroked with the current stroke color. (See **PXC_StrokePath**).

mode

[in] Specifies fill mode. Can be one of following values:

Value	Definition
`FillRule_Winding`	Fill using winding mode (fills any region with a nonzero winding value).
`FillRule_EvenOdd`	Fill using even-odd mode (fills the area between odd-numbered and even-numbered polygon sides on each scan line).

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.

To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

In general, the modes differ only in cases where a complex, overlapping path must be filled (for example, a five-sided polygon that forms a five-pointed star with a pentagon in the center). In such cases, `FillRule_EvenOdd` mode fills every other enclosed region within the polygon (that is, the points of the star), but `FillRule_Winding` mode fills all regions (that is, the points and the pentagon).

When the fill mode is `FillRule_EvenOdd` the function fills the area between odd-numbered and even-numbered polygon sides on each scan line. i.e. the function fills the area between the first and second side, between the third and fourth side, and so on.

When the fill mode is `FillRule_Winding`, the function fills any region that has a nonzero winding value. This value is defined as the number of times a pen used to draw the path would go around the region. The direction of each edge of the path is important.

Example (C++).

```
// Example shows how to draw rectangle filled with the color green

    _PXCContent*        pContent;

    ...

    // Set fill color to green

    HRESULT hr = PXC_SetFillColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Add rectangle

    hr = PXC_Rect(pContent, 20.0, 200.0, 320.0, 500.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // And stroke it

    hr = PXC_FillPath(pContent, TRUE, FALSE, FillRule_Winding);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

```
   ...
```

2.3.4.17 PXC_GetContentDC

PXC_GetContentDC

Top Previous Next

PXC_GetContentDC allows the creation of a context device, connected with any active content *content* and compatible with *refDC*. This is then utilised with the active content and the help of standard Windows API GDI functions.

```
HRESULT  PXC_GetContentDC(
     _PXCContent* content,
     HDC refDC,
     LPRECT drawRect,
     LPCPXC_RectF crect,
     HDC* cdc
);
```

Parameters

content
[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

refDC
[in] Identifier of any existing DC. If this parameter is equal to `NULL`, the graphic context, compatible with the desktop, will be created.

drawRect
[in] Defines the rectangle, within which all operations will take place.

crect
[in] Defines the rectangle in the coordinate system of the active document, for the created context to be linked to.

cdc
[in] Pointer to the identifier of the created context.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to get page DC of the rectangle area
// for making drawings

    _PXCContent*        pContent;
```

```
...

// Drawing rectangle

RECT drawRect;
drawRect.left = 0;
drawRect.top = 0;
drawRect.right = 100;
drawRect.bottom = 100;

// Document rectangle

PXC_RectF                pageRect;

pageRect.left = I2L(1);
pageRect.right = I2L(2);
pageRect.top = I2L(1);
pageRect.bottom = I2L(3);

// hdc for drawing

HDC hdc;

HRESULT hr = PXC_GetContentDC(pContent, NULL, &drawRect, &pageRect, &hdc);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

// Now hdc could be used for GDI operations
...

// After all drawings are done hdc must be released
PXC_ReleaseContentDC(pContent, FALSE);
```

2.3.4.18 PXC_GetLineInfo

PXC_GetLineInfo

Top Previous Next

PXC_GetLineInfo returns current properties of the line representation for stroke operations.

```
HRESULT  PXC_GetLineInfo(
    const _PXCContent* content,
    double* width,
    PXC_LineJoin* join,
    PXC_LineCap* cap,
    double* mlimit
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

width
: [out] Pointer to a `double` variable that receives the current width of lines. If this parameter is `NULL`, the width of the line is not returned.

join
: [out] Pointer to a `PXC_LineJoin` variable that receives the current line join style (**PXC_SetLineJoin**). If this parameter is `NULL` a corresponding value is not returned.

cap
: [out] Pointer to a `PXC_LineCap` variable that receives the current line end cap style (**PXC_SetLineCap**). This parameter may be `NULL`.

mlimit
: [out] Pointer to a `double` variable that receives the current value of the miter limit (see **PXC_SetMiterLimit**). This parameter may be `NULL`.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to set new line width, do some drawing operations,
// and then return to the original line width

   _PXCContent*        pContent;

   // Get current line width

   double oldwidth;
   HRESULT hr = PXC_GetLineInfo(pContent, &oldwidth, NULL, NULL, NULL);
   if (IS_DS_FAILED(hr))
   {
       // Handle error
       ...
   }

   // Set new line width

   hr = PXC_SetLineWidth(pContent, 3.0);
   if (IS_DS_FAILED(hr))
   {
       // Handle error
       ...
```

```
    }

    // Some drawing here using new line width
    ...

    // Set original line width

    hr = PXC_SetLineWidth(pContent, oldvalue);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.19 PXC_GradientFill

PXC_GradientFill

Top Previous Next

PXC_GradientFill fills rectangular and triangular structures.

```
HRESULT  PXC_GradientFill(
    _PXCContent* content,
    LPPXC_TRIVERTEX pVertex,
    ULONG dwNumVertex,
    PVOID pMesh,
    ULONG dwNumMesh,
    PXC_GradientMode dwMode
);
```

Parameters

content
[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

pVertex
[in] Pointer to an array of **PXC_TRIVERTEX** structures that each define a triangle vertex.

dwNumVertex
[in] The number of vertices in *pVertex* array.

pMesh
[in] Array of **GRADIENT_TRIANGLE** structures in triangle mode, or an array of **GRADIENT_RECT** structures in rectangle mode.

dwNumMesh
[in] The number of elements (triangles or rectangles) in *pMesh*.

dwMode
[in] Specifies gradient fill mode. This parameter can be one of the following values.

Mode	Definition

`Gradient_Rect_H`	In this mode, two endpoints describe a rectangle. The rectangle is defined to have a constant color (specified by the **PXC_TRIVERTEX** structure) for the left and right edges. The color will be interpolated from the left to right edge and fills the interior.
`Gradient_Rect_V`	In this mode, two endpoints describe a rectangle. The rectangle is defined to have a constant color (specified by the **PXC_TRIVERTEX** structure) for the top and bottom edges. The color will be interpolated from the top to bottom edge and fills the interior.
`Gradient_Triangle`	In this mode, an array of **PXC_TRIVERTEX** structures is passed along with a list of array indexes that describe separate triangles. Linear interpolation will be performed between triangle vertices and fills the interior.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

This function is similar to the Windows API **GradientFill** function.
See Microsoft MSDN Library for more detail about the **GradientFill** function and related structures.

It is not necessary to call the **PXC_FillPath** or **PXC_StrokePath** function's after a **PXC_GradientFill**.

Example (C++).

```
// Example shows how to do gradient filling

    _PXCContent*        pContent;

    ...

    PXC_RectF                        rc;
    PXC_TRIVERTEX                vert [4];
    GRADIENT_TRIANGLE        gTri[3];

    // Fill structures

    rc.left = 100;
    rc.top = 500;
    rc.right = 300;
    rc.bottom = 200;

    vert[0].x               =  rc.left;
    vert[0].y               =  rc.top;
    vert[0].color       =  0x000000;

    vert[1].x               =  rc.right;
    vert[1].y               =  rc.top;
    vert[1].color       =  RGB(0, 0, 255);
    vert[2].x                   =  rc.right;
    vert[2].y                   =  rc.bottom;
    vert[2].color           =  RGB(0, 255, 0);

    vert[3].x                   =  rc.left;
    vert[3].y                   =  rc.bottom;
    vert[3].color           =  RGB(255, 255, 255);

    gTri[0].Vertex1         = 0;
    gTri[0].Vertex2         = 1;
    gTri[0].Vertex3         = 2;

    gTri[1].Vertex1         = 1;
    gTri[1].Vertex2         = 2;
    gTri[1].Vertex3         = 3;

    gTri[2].Vertex1         = 2;
    gTri[2].Vertex2         = 3;
    gTri[2].Vertex3         = 0;

    HRESULT hr = PXC_GradientFill(pContent, vert, 4, (PVOID)gTri, 3,
Gradient_Triangle);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.20 PXC_LineTo

PXC_LineTo

Top Previous Next

PXC_LineTo adds a line segment from the current point to a specified point in the current path. The current point will be moved to a point designated by the `(x, y)` coordinates.

```
HRESULT  PXC_LineTo(
    _PXCContent* content,
    double x,
    double y
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

x

[in] Specifies the x-coordinate of the line's ending point.

y

[in] Specifies the y-coordinate of the line's ending point.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to draw a green rectangle

    _PXCContent*        pContent;

    ...

    // Set stroke color to green

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Create new path

    hr = PXC_MoveTo(pContent, 100, 200);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    hr = PXC_LineTo(pContent, 300, 200);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    hr = PXC_LineTo(pContent, 300, 500);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    hr = PXC_LineTo(pContent, 100, 500);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // NOTE: we do not close the path,
    // in the next call it will be done automatically!

    // Stroke the path

    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.21 PXC_MoveTo

PXC_MoveTo

Top Previous Next

PXC_MoveTo begins a new path (or subpath), and sets its starting position to a specified point.

```
HRESULT  PXC_MoveTo(
    _PXCContent* content,
    double x,
    double y
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

x

[in] Specifies the x-coordinate of the new position.

y

[in] Specifies the y-coordinate of the new position.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to set the point as a new starting point
// for a new path or subpath

    _PXCContent*        pContent;

    ...

    // Set new point as a start for the new path

    hr = PXC_MoveTo(pContent, 100, 100);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.22 PXC_NoDash

PXC_NoDash

Top Previous Next

PXC_NoDash sets a solid line style for stroke operations.

```
HRESULT  PXC_NoDash(
    _PXCContent* content
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

This function is equivalent to: `PXC_SetDash(content, 0, 0, 0);`

Example (C++).

```
// Example shows how to 'switch off' dashed line style
// and set line style to 'solid'

    _PXCContent*        pContent;
    ...

    HRESULT hr = PXC_NoDash(pContent);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.23 PXC_Pie

PXC_Pie

Top Previous Next

PXC_Pie adds to the current path a pie-shaped wedge bounded by the intersection of an ellipse and two radials.

See **PXC_PieEx** for an alternative method.

```
HRESULT  PXC_Pie(
    _PXCContent* content,
    LPCPXC_RectF rect,
    double alpha,
    double beta
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

rect
: [in] Pointer to the **PXC_RectF** structure which contains the coordinates of the bounding rectangle.

alpha
: [in] Specifies starting angle in degrees.

beta
: [in] Specifies ending angle in degrees.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to draw a green pie wedge

    _PXCContent*        pContent;
```

```
...

// Set stroke color to green

HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

// Add pie

PXC_RectF rect;

rect.left = 20.0;
rect.top = 200.0;
rect.right = 320.0;
rect.bottom = 500.0;


hr = PXC_Pie(pContent, &rect, -90.0, +90.0);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

// And stroke it

hr = PXC_StrokePath(pContent, TRUE);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

...
```

2.3.4.24 PXC_PieEx

PXC_PieEx

Top Previous Next

The **PXC_PieEx** adds to the current path a pie-shaped wedge bounded by the intersection of an ellipse and two radials. It is similar to **PXC_Pie**, but uses a different method to specify angles.

```
HRESULT  PXC_PieEx(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCPXC_PointF pnt1,
    LPCPXC_PointF pnt2
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

rect

[in] Pointer to **PXC_RectF** structure which contains the coordinates of the bounding rectangle.

pnt1

[in] Pointer to **PXC_PointF** which contains the coordinates of the ending point of the radial line defining the starting point of the arc.

pnt2

[in] Pointer to **PXC_PointF** which contains the coordinates of the ending point of the radial line defining the ending point of the arc.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to draw a green pie wedge

    _PXCContent*         pContent;

    ...

    // Set stroke color to green

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Add pie

    PXC_RectF rect;

    rect.left = 20.0;
    rect.top = 200.0;
    rect.right = 320.0;
    rect.bottom = 500.0;

    PXC_PointF          ptStart;
    PXC_PointF          ptEnd;

    ptStart.x = 0.0;
    ptStart.y = 0.0;

    ptEnd.x = 150.0;
    ptEnd.y = 450.0;


    hr = PXC_PieEx(pContent, &rect, &ptStart, &ptEnd);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // And stroke it

    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.25 PXC_PolyCurve

PXC_PolyCurve

Top Previous Next

PXC_PolyCurve adds one or more Bezier curves to the current path.

```
HRESULT  PXC_PolyCurve(
    _PXCContent* content,
    LPCPXC_PointF points,
    UINT pntCount
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

points

[in] Pointer to an array of **PXC_PointF** which sets (X,Y) coordinates of endpoints and are the control point of the curve(s).

pntCount

[in] Specifies the number of items in the array pointed to by *points*.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

The **PXC_PolyCurve** function adds cubic Bezier curves by using the endpoints and control points specified by *points*. The first curve is drawn from the current point to the third point by using the first and second points as control points. Each subsequent curve in the sequence needs exactly three more points: the ending point of the previous curve is used as the starting point of the next, the next two points in the sequence are control points, and the third is the ending point.

Example (C++).

```
// Example show how to draw star with the specified number of points
// using Bezier curves

   void DrawStar(_PXCContent* page, double x, double y, double r, int
BeamCount)
   {
       // Allocate the array of points
       PXC_PointF* pxy = new PXC_PointF[BeamCount];
       if (!pxy)
           return;

       // Construct the array

       double a = -90;
       for (int i = 0; i < BeamCount; i++)
       {
           pxy[i].x = x + r * cos(a * PI / 180.0);
           pxy[i].y = y - r * sin(a * PI / 180.0);
           a += 2.0 * (360.0 / BeamCount);
       }

       // Add curve to the path

       HRESULT hr = PXC_PolyCurve(page, pxy, BeamCount);
       if (IS_DS_FAILED(hr))
       {
           delete[] pxy;
           return;
       }
```

```
        // Stroke it

        hr = PXC_StrokePath(pContent, TRUE);
        if (IS_DS_FAILED(hr))
        {
            delete[] pxy;
            return;
        }
    }
```

2.3.4.26 PXC_Polygon

PXC_Polygon

Top Previous Next

PXC_Polygon adds one or more line segments to the current path.

```
HRESULT  PXC_Polygon(
    _PXCContent* content,
    LPCPXC_PointF points,
    UINT pntCount,
    BOOL bMove
);
```

Parameters

content

[in] Parameter *content* specifies identifier for the page content to which the function will be applied.

points

[in] Pointer to an array of **PXC_PointF** which sets the (X, Y) coordinates of the endpoints of a line segment.

pntCount

[in] Specifies the number of items in the array pointed to by *points*.

bMove

[in] If this parameter is TRUE, then the first array item specifies the coordinates of the starting point, from which the subsequent line segments will be drawn. Otherwise the starting point will be current position.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example show how to draw star with the specified number of points

    void DrawStar(_PXCContent* page, double x, double y, double r, int
BeamCount)
    {
        // Allocate the array of points
        PXC_PointF* pxy = new PXC_PointF[BeamCount];
        if (!pxy)
            return;

        // Construct the array

        double a = -90;
        for (int i = 0; i < BeamCount; i++)
        {
            pxy[i].x = x + r * cos(a * PI / 180.0);
            pxy[i].y = y - r * sin(a * PI / 180.0);
            a += 2.0 * (360.0 / BeamCount);
        }

        // Add Polygon to the path

        HRESULT hr = PXC_Polygon(page, pxy, BeamCount, TRUE);
        if (IS_DS_FAILED(hr))
        {
            delete[] pxy;
            return;
        }

        // Stroke it

        hr = PXC_StrokePath(pContent, TRUE);
        if (IS_DS_FAILED(hr))
        {
            delete[] pxy;
            return;
        }
    }
```

2.3.4.27 PXC_Rect

PXC_Rect

Top Previous Next

PXC_Rect adds a rectangle to the current path.

```
HRESULT  PXC_Rect(
    _PXCContent* content,
    double left,
    double top,
    double right,
    double bottom
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

left
: [in] Specifies the x-coordinate of the upper-left corner of a rectangle.

top
: [in] Specifies the y-coordinate of the upper-left corner of a rectangle.

right
: [in] Specifies the x-coordinate of the lower-right corner of a rectangle.

bottom
: [in] Specifies the y-coordinate of the lower-right corner of a rectangle.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

PXC_Rect automaticaly closes the path.

Example (C++).

```
// Example shows how to draw a green rectangle

    _PXCContent*        pContent;

    ...

    // Set stroke color to green

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Add rectangle

    hr = PXC_Rect(pContent, 20.0, 200.0, 320.0, 500.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    // And stroke it

    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.28 PXC_ReleaseContentDC

PXC_ReleaseContentDC

Top Previous Next

PXC_ReleaseContentDC releases the context, linked with the active content, which was previously created by the **PXC_GetContentDC** function.

```
HRESULT  PXC_ReleaseContentDC(
    _PXCContent* content,
    BOOL bCancel
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

bCancel

[in] If the parameter is `TRUE`, all the operations, performed with the graphical context, will be undone.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

If *bCancel* is not `TRUE`, all the operations, performed with the context, will be sent to the PDF document as a metafile. Context will be released in any event.

Example (C++).

```
// After all operations with the DC obtained by the 'PXC_GetContentDC'
// function are complete, the DC should be released

    _PXCContent*        pContent;
    ...

    // Get HDC, do some GDI operations
    ...

    // After all drawings are done hdc must be released
    PXC_ReleaseContentDC(pContent, FALSE);
```

2.3.4.29 PXC_SetBlendMode

PXC_SetBlendMode

Top Previous Next

PXC_SetBlendMode sets the blend mode for the image creation operations.

```
HRESULT  PXC_SetBlendMode(
    _PXCContent* content,
    PXC_BlendMode bMode
);
```

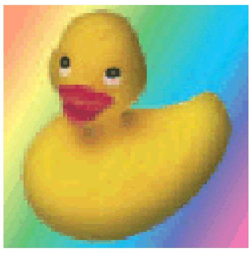
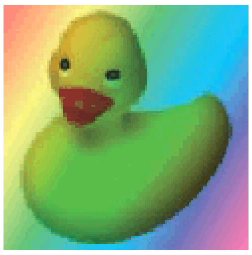
Parameters

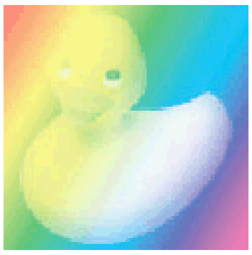
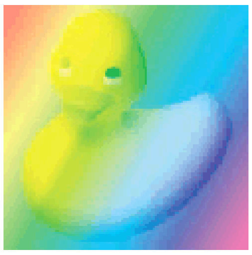
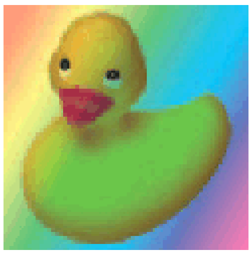
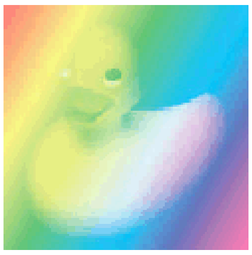
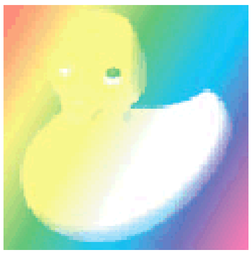
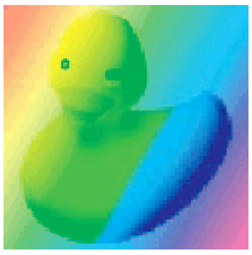
content

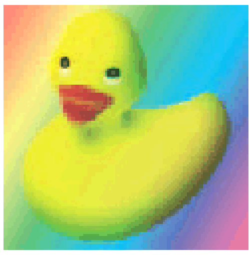
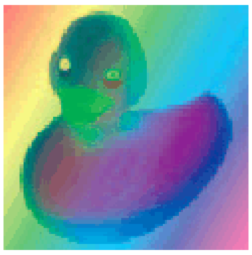
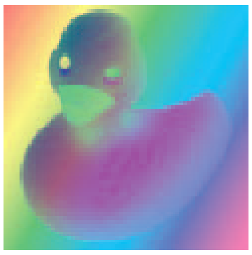
[in] Parameter *content* specifies identifier of the page content to which the function will be applied.

bMode

[in] Parameter *bMode* specifies the blend mode code. May be any one of the following values:

Constant	Definition	Sample
BlendMode_Normal	Selects the source color, ignoring the backdrop: B(Cb, Cs) = Cs	
BlendMode_Multiply	Multiplies the backdrop and source color values: B(Cb, Cs) = Cb * Cs The resulting color is always at least as dark as either of the two constituent colors. Multiplying any color with black produces black; multiplying with white leaves the original color unchanged. Painting successive overlapping objects with a color other than black or white produces progressively darker colors.	

`BlendMode_Screen`	Multiplies the complements of the backdrop and source color values, then complements the result: B(Cb, Cs) = 1 - [(1 - Cb) * (1 - Cs)] The resulting color is always at least as light as either of the two constituent colors. Screening any color with white produces white; screening with black leaves the original color unchanged. The effect is similar to projecting multiple photographic slides simultaneously onto a single screen.	
`BlendMode_Overlay`	Multiplies or screens the colors, depending on the backdrop color. Source colors overlay the backdrop while preserving its highlights and shadows. The backdrop color is not replaced, but is mixed with the source color to reflect the lightness or darkness of the backdrop.	
`BlendMode_Darken`	Selects the darker of the backdrop and source colors: B(Cb, Cs) = min(Cb, Cs) The backdrop is replaced with the source where the source is darker; otherwise it is left unchanged.	
`BlendMode_Lighten`	Selects the lighter of the backdrop and source colors: B(Cb, Cs) = max(Cb, Cs) The backdrop is replaced with the source where the source is lighter; otherwise it is left unchanged.	
`BlendMode_ColorDodge`	Brightens the backdrop color to reflect the source color. Painting with black produces no change.	
`BlendMode_ColorBurn`	Darkens the backdrop color to reflect the source color. Painting with white produces no change.	

`BlendMode_HardLight`	Multiplies or screens the colors, depending on the source color value. If the source color is lighter than 0.5, the backdrop is lightened, as if it was screened; this is useful for adding highlights to a scene. If the source color is darker than 0.5, the backdrop is darkened, as if it was multiplied; this is useful for adding shadows to a scene. The degree of lightening or darkening is proportional to the difference between the source color and 0.5; if it is equal to 0.5, the backdrop is unchanged. Painting with pure black or white produces pure black or white. The effect is similar to shining a harsh spotlight on the backdrop.	
`BlendMode_SoftLight`	Darkens or lightens the colors, depending on the source color value. If the source color is lighter than 0.5, the backdrop is lightened, as if it was dodged; this is useful for adding highlights to a scene. If the source color is darker than 0.5, the backdrop is darkened, as if it was burned in. The degree of lightening or darkening is proportional to the difference between the source color and 0.5; if it is equal to 0.5, the backdrop is unchanged. Painting with pure black or white produces a distinctly darker or lighter area, but does not result in pure black or white. The effect is similar to shining a diffused spotlight on the backdrop.	
`BlendMode_Difference`	Subtracts the darker of the two constituent colors from the lighter: B(Cb, Cs) = |Cb - Cs| Painting with white inverts the backdrop color; painting with black produces no change.	
`BlendMode_Exclusion`	Produces an effect similar to that of the Difference mode, but lower in contrast. Painting with white inverts the backdrop color; painting with black produces no change.	

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an error code.
To determine if the function was successful use the defined macro's as described here: Error Handling.

Comments

In principle, the blend function B(Cb, Cs), used in the composing formula to customize the blending operation, could be any function of the backdrop and source colors that yields another color, Cr, for the result.

A blend mode is termed separable if each component of the resulting color is completely determined by the corresponding components of the constituent backdrop and source colors—that is, if the blend mode function B is applied separately to each set of corresponding components:

Cr = B(Cb, Cs)

where the lowercase variables cr, cb, and cs denote corresponding components of the colors Cr, Cb, and Cs, expressed in additive form. (Theoretically, a blend mode could have a different function for each color component and still be separable; however, none of the standard PDF blend modes have this property.) A separable blend mode can be used with any color space, since it applies independently to any number of components. Only separable blend modes can be used for blending spot colors.

Example (C++).

```
// Set blend mode to 'Multiply'

    _PXCContent*        pContent;

    ...

    hr = PXC_SetBlendMode(pContent, BlendMode_Multiply);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.30 PXC_SetDash

PXC_SetDash

Top Previous Next

PXC_SetDash sets the dash style (dash pattern) used for stroke paths.

```
HRESULT  PXC_SetDash(
    _PXCContent* content,
    double b,
    double w,
    double offs
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for page content to which the function will be applied.

b

[in] Specifies length of the "black" portion of the dash.

w

[in] Specifies length of the "white" portion of the dash.

offs

[in] Specifies the distance into the dash pattern at which to start the dash.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

For more information please see **PXC_SetPolyDash**. The function **PXC_SetDash** is equivalent to the following code:

```
double darray[2];
darray[0] = b;
darray[1] = w;
PXC_SetPolyDash(content, darray, 2, offs);
```

Example (C++).

```
// Example shows how to set new dash style

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_SetDash(pContent, 3.0, 5.0, 0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.31 PXC_SetDrawingColor

PXC_SetDrawingColor

Top Previous Next

PXC_SetDrawingColor sets the graphics stroke and fill color.

```
HRESULT  PXC_SetDrawingColor(
    _PXCContent* content,
    COLORREF color
);
```

Parameters

content

[in] Parameter *content* specifies identifier of page content to which the function will be applied.

color

[in] Specifies color for the stroke and fill.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

The function is equivalent to:

```
PXC_SetStrokeColor(content, color);
PXC_SetFillColor(content, color);
```

Example (C++).

```
    _PXCContent*        pContent;

...

// Set new drawing color as blue

HRESULT hr = PXC_SetDrawingColor(pContent, RGB(0, 0, 255));
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}

...
```

2.3.4.32 PXC_SetDrawingGray

PXC_SetDrawingGray

Top Previous Next

PXC_SetDrawingGray sets the fill and stroke color as the level of gray.

```
HRESULT  PXC_SetDrawingGray(
    _PXCContent* content,
    BYTE gLevel
);
```

Parameters

content
[in] Parameter *content* specifies identifier of the page content to which the function will be applied.

gLevel
[in] Specifies the level of gray.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

The function is equivalent to

```
PXC_SetDrawingColor(content, RGB(gLevel, gLevel, gLevel));
```

Example (C++).

```
    _PXCContent*        pContent;

 ...

 // Set new drawing color as gray

 HRESULT hr = PXC_SetDrawingGray(pContent, 128);
 if (IS_DS_FAILED(hr))
 {
     // Handle error
     ...
 }

 ...
```

2.3.4.33 PXC_SetFillColor

PXC_SetFillColor

Top Previous Next

PXC_SetFillColor sets the fill color.

```
HRESULT  PXC_SetFillColor(
    _PXCContent* content,
    COLORREF color
);
```

Parameters

content
[in] Parameter *content* specifies identifier of page content to which function will be applied.

color
[in] Specifies fill color.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
   _PXCContent*        pContent;

    ...

    // Set new fill color as red

    HRESULT hr = PXC_SetFillColor(pContent, RGB(255, 0, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.34 PXC_SetFillGray

PXC_SetFillGray

Top Previous Next

PXC_SetFillGray sets the gray level of the fill color.

```
HRESULT  PXC_SetFillGray(
    _PXCContent* content,
    BYTE gLevel
);
```

Parameters

content

[in] Parameter *content* specifies identifier of the page content to which the function will be applied.

gLevel

[in] Specifies the level of gray.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

The function is equivalent to
`PXC_SetFillColor(content, RGB(gLevel, gLevel, gLevel));`

Example (C++).

```
   _PXCContent*        pContent;

    ...
    // Set new fill color as gray

    HRESULT hr = PXC_SetFillGray(pContent, 128);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.35 PXC_SetFlat

PXC_SetFlat

Top Previous Next

The **PXC_SetFlat** function sets the flatness tolerance.

```
HRESULT  PXC_SetFlat(
    _PXCContent* content,
    double flat_tolerance
);
```

Parameters

content
[in] Parameter *content* specifies identifier of page content to which function will be applied.

flat_tolerance
[in] Specifies flatness tolerance. The value must be in range between 0 and 100.0 .

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

The *flatness tolerance* controls the maximum permitted distance in device pixels between the mathematically correct path and an approximation constructed from straight line segments, as shown in the picture.

Note: Although the figure exaggerates the difference between the curved and flattened paths for the sake of clarity, the purpose of the flatness tolerance is to control the precision of curve rendering, not to draw inscribed polygons. If the parameter's value is large enough to cause visible straight line segments to appear, the result is unpredictable.

Example (C++).

```
// Example shows how to set the flatness tolerance

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_SetFlat(pContent, 10.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.36 PXC_SetLineCap

PXC_SetLineCap

Top Previous Next

PXC_SetLineCap specifies the shape to be used at the ends of open subpath (and dashes, if any) when they are stroked.

```
HRESULT  PXC_SetLineCap(
    _PXCContent* content,
    PXC_LineCap cap
);
```

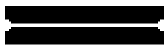
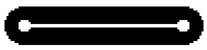
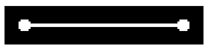
Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

cap

[in] End cap style. Can be any one of the following values:

Constant	Appearance	Meaning
LineCap_Butt		The stroke is squared off at the endpoint of the path. There is no projection beyond the end of the path.
LineCap_Round		A semicircular arc with a diameter equal to the line width is drawn around the endpoint and filled in.
LineCap_Square		The stroke continues beyond the endpoint of the path for a distance equal to half the line width and is then squared off.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to set round line ends

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_SetLineCap(pContent, LineCap_Round);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.37 PXC_SetLineJoin

PXC_SetLineJoin

Top Previous Next

The **PXC_SetLineJoin** function sets the line join type.

```
HRESULT  PXC_SetLineJoin(
    _PXCContent* content,
    PXC_LineJoin join
);
```

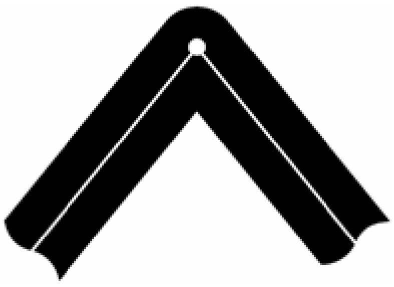
Parameters

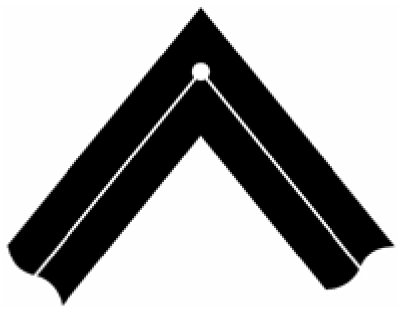
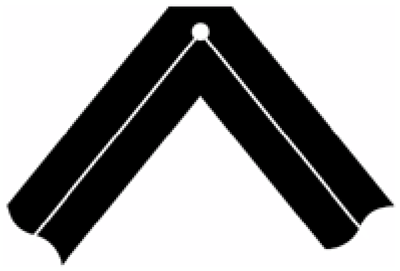
content

[in] Parameter *content* specifies the identifier of page content to which the function will be applied.

join

[in] Specifies the type of line join. May be any one of the following values:

Value	**Appearance**	**Meaning**
LineJoin_Miter		The outer edges of the strokes for the two segments are extended until they meet at some angle, as in a picture frame. If the segments meet at too sharp angle (as defined by the miter limit parameter — see PXC_SetMiterLimit), a bevel join is used instead.

`LineJoin_Round`		A circle with a diameter equal to the line width is drawn around the point where the two segments meet and is filled in, producing a rounded corner. **Note:** If the path segments shorter than the half of the line width meet at a sharp angle, an unintended "wrong side" of the circle may appear.
`LineJoin_Bevel`		The two segments are finished with butt caps (see **PXC_SetLineCap**) and the resulting notch beyond the ends of the segments is filled with a triangle.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to set 'round' line join

    _PXCContent*          pContent;

    ...

    HRESULT hr = PXC_SetLineJoin(pContent, LineJoin_Round);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.38 PXC_SetLineWidth

PXC_SetLineWidth

Top Previous Next

PXC_SetLineWidth sets the line width for stroke operations.

```
HRESULT  PXC_SetLineWidth(
    _PXCContent* content,
    double width
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier of page content to which the function will be applied.

width
: [in] Specifies line width. Line width is specified in points (1/72 of inch).

  Note: A line width of `0` denotes the thinnest line that can be rendered at device resolution: 1 device pixel wide. However, some devices cannot reproduce 1-pixel lines, and on high-resolution devices, they are nearly invisible. Since the results of rendering such "zero-width" lines are device-dependent, their use is not recommended.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Default line width value is `1.0` point.

Example (C++).

```
// Example shows how to set line width

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_SetLineWidth(pContent, 0.5);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.39 PXC_SetMiterLimit

PXC_SetMiterLimit

Top Previous Next

The **PXC_SetMiterLimit** function sets the miter limit for path drawing.

```
HRESULT  PXC_SetMiterLimit(
    _PXCContent* content,
    double mlimit
);
```

Parameters

content

[in] Parameter *content* specifies identifier of the page content to which the function will be applied.

mlimit

[in] Specifies miter limit (see **Comments**). This value may not exceed 10.0.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

When two line segments meet at a sharp angle and mitered joins have been specified as the line join style, it is possible for the miter to extend far beyond the thickness of the line stroking the path. The miter limit imposes a maximum on the ratio of the miter length to the line width (see picture above). When the limit is exceeded, the join is converted from a miter to a bevel.

The ratio of miter length to line width is directly related to the angle φ between the segments in user space by the formula:

$$\frac{miterLength}{lineWidth} = \frac{1}{\sin\left(\frac{\varphi}{2}\right)}$$

For example, a miter limit of 1.414 converts miters to bevels for φ less than 90 degrees, a limit of 2.0 converts them for φ less than 60 degrees, and a limit of 10.0 converts them for φ less than approximately 11.5 degrees.

Example (C++).

```
// Example shows how to set new miter limit for path drawing

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_SetMiterLimit(pContent, 3.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.40 PXC_SetPolyDash

PXC_SetPolyDash

Top Previous Next

PXC_SetPolyDash sets the dash style (dash pattern) used for stroke paths.

```
HRESULT  PXC_SetPolyDash(
    _PXCContent* content,
    double* darray,
    DWORD arCount,
    double offs
);
```

Parameters

content

[in] Parameter *content* specifies identifier of the page content to which the function will be applied.

darray

[in] Specifies an array of `double` pairs, which represents the dash pattern.

arCount

[in] Specifies the number of items in the *darray* array. This number must be even.

offs

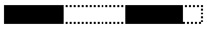
[in] Specifies the distance into the dash pattern at which to start the dash.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Before beginning to stroke a path, the dash array is cycled through, adding up the lengths of dashes and gaps. When the accumulated length becomes equal or greater than the value specified by dash phase (*offs*), stroking of the path begins, using the dash array (*darray*) cyclically from that point onward. The table below shows examples of line dash patterns. As one can see in the table, an empty dash array and zero phase can be used to restore the dash pattern to a solid line.

Appearance	Dash Array And Phase	Description
	{0, 0} 0	No dash; solid, unbroken lines.
	**{3, 3} 0**	3 points on, 3 points off, ...
	{2, 2} 1	1 on, 2 off, 2 on, 2 off, ...
	{2, 1} 0	2 on, 1 off, 2 on, 1 off, ...

Example (C++).

```
// Example shows how to set new dash style

   _PXCContent*        pContent;

   ...

   double DashArray[4] = { 3.0, 2.0, 4.0, 1.0 };

   HRESULT hr = PXC_SetPolyDash(pContent, DashArray, 4, 0);
   if (IS_DS_FAILED(hr))
   {
       // Handle error
```

```
        ...
    }
```

2.3.4.41 PXC_SetStrokeAdjust

PXC_SetStrokeAdjust

Top Previous Next

PXC_SetStrokeAdjust switches stroke adjustment on or off.

```
HRESULT  PXC_SetStrokeAdjust(
    _PXCContent* content,
    BOOL bAdjust
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

bAdjust

[in] If this value is `TRUE` the stroke adjustment will be turned on; otherwise it will be turned off.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

When a stroke is drawn along a path, the scan conversion algorithm may produce lines of non uniform thickness because of rasterization effects. In general, the line width and the coordinates of the endpoints, transformed into device space, are arbitrary real numbers which are not device pixels. Therefore a line of a given width can intersect with different numbers of device pixels, depending on where it is positioned.

For best results, it is important to compensate for the rasterization effects to produce strokes of uniform thickness. This is especially important in low-resolution display applications.

Example (C++).

```
// Example shows how to turn on stroke adjustment

    _PXCContent*        pContent;

    ...

    HRESULT hr = PXC_SetStrokeAdjust(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.42 PXC_SetStrokeColor

PXC_SetStrokeColor

Top Previous Next

PXC_SetStrokeColor sets the stroke color.

```
HRESULT  PXC_SetStrokeColor(
    _PXCContent* content,
    COLORREF color
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

color
: [in] Specifies stroke color.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
      _PXCContent*        pContent;

    ...

    // Set new stroke color as green

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.43 PXC_SetStrokeGray

PXC_SetStrokeGray

Top Previous Next

The **PXC_SetStrokeGray** function sets the gray level of the stroke color.

```
HRESULT  PXC_SetStrokeGray(
    _PXCContent* content,
    BYTE gLevel
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

gLevel

[in] Specifies the level of gray.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

The function is equivalent to `PXC_SetStrokeColor(content, RGB(gLevel, gLevel, gLevel));`

Example (C++).

```
        _PXCContent*        pContent;

    ...

    // Set new stroke color as gray

    HRESULT hr = PXC_SetStrokeGray(pContent, 128);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.3.4.44 PXC_SetTransparency

PXC_SetTransparency

Top Previous Next

PXC_SetTransparency sets the transparency level for the subsequent fill and stroke operations.

```
HRESULT  PXC_SetTransparency(
    _PXCContent* content,
    BYTE tFill,
    BYTE tStroke
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which the function will be applied.

tFill

[in] Parameter *tFill* specifies the transparency level for the subsequent fill operations.

tStroke

[in] Parameter *tStroke* specifies the transparency level for the subsequent stroke operations.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

The transparency level must be set within the range from 0 (opaque paint) to 255 (fully transparent paint).

Example (C++).

```
// Example shows how to set transparency level
// for Fill and Stroke operations.

    _PXCContent*         pContent;

    ...

    // Set new level of transparency

    hr = PXC_SetTransparency(pContent, 128, 128);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
```

2.3.4.45 PXC_StrokePath

PXC_StrokePath

The **PXC_StrokePath** function strokes the current path, applying the current stroke color, line join style etc.

```
HRESULT  PXC_StrokePath(
    _PXCContent* content,
    BOOL bClose
);
```

Parameters

content

[in] Parameter *content* specifies the identifier for the page content to which thefunction will be applied.

bClose

[in] Closes the current path before applying the current parameters for the stroke.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to stroke path,
    // i.e. draw rectangle and then stroke it with the green color

    _PXCContent*        pContent;

    ...

    // Set stroke color to green

    HRESULT hr = PXC_SetStrokeColor(pContent, RGB(0, 255, 0));
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Add rectangle

    hr = PXC_Rect(pContent, 20.0, 200.0, 320.0, 500.0);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // And stroke it
    hr = PXC_StrokePath(pContent, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

2.4 AcroForms

2.4.1 AcroForms Common Functions

AcroForms Common Functions

Top Previous Next

These functions apply to all AcroForm creation:

- PXC_AddJavaScript
- PXC_FormField_GetBorderInfo
- PXC_FormField_SetAppearanceInfo
- PXC_FormField_SetBorderInfo
- PXC_FormField_SetFlags
- PXC_FormField_SetReadOnly
- PXC_FormField_SetRequired
- PXC_FormField_SetTooltip

2.4.1.1 PXC_AddJavaScript

PXC_AddJavaScript

Top Previous Next

PXC_AddJavaScript adds a *JavaScript* code block to the PDF document. You can then refer to this code in PXC_FormField_AddJSAction.

```
HRESULT  PXC_AddJavaScript(
    _PXCDocument* doc,
    LPCSTR jsName,
    LPCSTR js
);
```

Parameters

page
[in] *doc* Specifies the page object.

jsName
[in] *jsName* - pointer to a null-terminated string containing the name of the **JavaScript** block: function or global variables.

js

[in] *js* - pointer to a null-terminated string containing **JavaScript** code in the *jsName* block.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.1.2 PXC_FormField_GetBorderInfo

PXC_FormField_GetBorderInfo

Top Previous Next

PXC_FormField_GetBorderInfo gets the control's current or default Border properties.

```
HRESULT  PXC_FormField_GetBorderInfo(
    _PXCFormControl* control,
    PXC_BorderInfo* border
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

border

[in] *border* specifies the control border information to be retrieved which is represented by **PXC_BorderInfo** type.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.1.3 PXC_FormField_SetAppearanceInfo

PXC_FormField_SetAppearanceInfo

Top Previous Next

PXC_FormField_SetAppearanceInfo sets the form field appearance properties.

```
HRESULT  PXC_FormField_SetAppearanceInfo(
    _PXCFormControl* control,
    const PXC_CommonFieldAppearance* appearance
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

appearance

[in] *appearance* - pointer to the **PXC_CommonFieldAppearance** containing information for the appearance properties.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.1.4 PXC_FormField_SetBorderInfo

PXC_FormField_SetBorderInfo

Top Previous Next

PXC_FormField_SetBorderInfo sets the control's Border properties.

```
HRESULT  PXC_FormField_SetBorderInfo(
    _PXCFormControl* control,
    const PXC_BorderInfo* border
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

border

[in] *border* specifies control border information as represented by **PXC_BorderInfo** type.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.1.5 PXC_FormField_SetFlags

PXC_FormField_SetFlags

Top Previous Next

PXC_FormField_SetFlags sets flags to the control.

```
HRESULT  PXC_FormField_SetFlags(
    _PXCFormControl* control,
    PXC_AnnotsFlags Flags
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

Flags

[in] *Flags* specifies the flags to be set as defined in **PXC_AnnotsFlags** type.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: PXC_AddPushButton, PXC_AddCheckBox, PXC_AddRadioButton, PXC_AddTextBox or PXC_AddChoice.

2.4.1.6 PXC_FormField_SetReadOnly

PXC_FormField_SetReadOnly

Top Previous Next

PXC_FormField_SetReadOnly marks the control as Read Only.

```
HRESULT  PXC_FormField_SetReadOnly(
    _PXCFormControl* control,
    BOOL bReadOnly
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

bReadOnly

[in] *bReadOnly* specifies whether to mark the control as Read Only.

Return Values

If the function succeeds, the return value is non-negative integer.

If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.1.7 PXC_FormField_SetRequired

PXC_FormField_SetRequired

Top Previous Next

PXC_FormField_SetRequired marks a control as Required.

```
HRESULT  PXC_FormField_SetRequired(
    _PXCFormControl* control,
    BOOL bRequired
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

bRequired

[in] *bRequired* specifies whether to mark the control as Required.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.1.8 PXC_FormField_SetTooltip

PXC_FormField_SetTooltip

Top Previous Next

PXC_FormField_SetTooltip sets a tooltip for the control.

```
HRESULT  PXC_FormField_SetTooltip(
    _PXCFormControl* control,
    LPCWSTR tooltip
);
```

Parameters

control

[in] *control* specifies the control object previously created by the according **Add function**, see remark.

tooltip

[in] *tooltip* - pointer to a null-terminated UNICODE string containing the tooltip of the control.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.2 AcroForms Check Box

AcroForms Check Box

Top Previous Next

AcroForm Check Box functions are:

- **PXC_AddCheckBox**
- **PXC_CheckBox_GetOptions**
- **PXC_CheckBox_SetOnStateValue**
- **PXC_CheckBox_SetOptions**
- **PXC_CheckBox_Options**

2.4.2.1 PXC_AddCheckBox

PXC_AddCheckBox

Top Previous Next

Function **PXC_AddCheckBox** adds a checkbox to the PDF page.

```
HRESULT  PXC_AddCheckBox(
    _PXCPage* page,
    const PXC_RectF* rect,
    LPCWSTR name,
    BOOL checked,
    _PXCCheckBox** pCheckBox
);
```

Parameters

page

[in] *page* specifies the page object.

rect
[in] *rect* - pointer to a rectangle that will hold new checkbox coordinates.

name
[in] *name* - pointer to a null-terminated UNICODE string containing the name of the checkbox.

checked
[in] *checked* is boolean value indicating whether the new checkbox will be checked by default.

pCheckBox
[out] *pCheckBox* specifies a pointer to a variable of the `_PXCPushButton*` type, which will represent the checkbox on the *page* page.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.2.2 PXC_CheckBox_GetOptions

PXC_CheckBox_GetOptions

Top Previous Next

PXC_CheckBox_GetOptions gets checkbox options.

```
HRESULT  PXC_CheckBox_GetOptions(
     _PXCCheckBox* checkbox,
     PXC_CheckBox_Options* options
);
```

Parameters

checkbox
[in] *checkbox* specifies the checkbox object previously created by the function **PXC_AddCheckBox**.

options
[in] *options* - pointer to the **PXC_CheckBox_Options** which will receive information about checkbox options.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.2.3 PXC_CheckBox_SetOnStateValue

PXC_CheckBox_SetOnStateValue

Top Previous Next

PXC_CheckBox_SetOnStateValue sets the checkbox state value to **On** by default. This is the "Yes" option according to the PDF specification.

```
HRESULT  PXC_CheckBox_SetOnStateValue(
    _PXCCheckBox* checkbox,
    LPCWSTR lpcwOnStateVal
);
```

Parameters

checkbox
: [in] *checkbox* specifies the checkbox object previously created by the function **PXC_AddCheckBox**.

lpcwOnStateVal
: [in] *lpcwOnStateVal* - pointer to a null-terminated UNICODE string containing the name of the **On** state value.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.2.4 PXC_CheckBox_SetOptions

PXC_CheckBox_SetOptions

Top Previous Next

PXC_CheckBox_SetOptions sets checkbox options.

```
HRESULT  PXC_CheckBox_SetOptions(
    _PXCCheckBox* checkbox,
    const PXC_CheckBox_Options* options
);
```

Parameters

checkbox
: [in] *checkbox* specifies the checkbox object previously created by the function **PXC_AddCheckBox**.

options
: [in] *options* - pointer to the **PXC_CheckBox_Options** which contains information about checkbox options.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.2.5 PXC_CheckBox_Options

PXC_CheckBox_Options

Top Previous Next

The **PXC_CheckBox_Options** structure specifies checkbox options.

```
typedef struct _PXC_CheckBox_Options {
   PXC_AnnotsFlags Flags;
   BOOL bReadOnly;
   BOOL bRequired;
   LPSTR lpszExportVal;
   BOOL bChecked;
   DWORD dwStyle;
} PXC_CheckBox_Options;
```

Members

Flags

specifies flags which may hold the following value:

Constant	Value	Description
AF_Invisible	0	If set, do not display the annotation if it does not belong to one of the standard annotation types and no annotation handler is available. If clear, display such an unknown annotation using an appearance stream specified by its appearance dictionary, if any
AF_Hidden	1	(PDF 1.2) If set, do not display or print the annotation or allow it to interact with the user, regardless of its annotation type or whether an annotation handler is available. In cases where screen space is limited, the ability to hide and show annotations selectively can be used in combination with appearance streams to display auxiliary pop-up information similar in function to online help systems
AF_Print	2	(PDF 1.2) If set, print the annotation when the page is printed. If clear, never print the annotation, regardless of whether it is displayed on the screen. This can be useful, for example, for annotations representing interactive pushbuttons, which would serve no meaningful purpose on the printed page
AF_NoZoom	3	(PDF 1.3) If set, do not scale the annotation’s appearance to match the magnification of the page. The location of the annotation on the page (defined by the upper-left corner of its annotation rectangle) remains fixed, regardless of the page magnification. See below for further discussion
AF_NoRotate	4	(PDF 1.3) If set, do not rotate the annotation’s appearance to match the rotation of the page. The upper-left corner of the annotation rectangle remains in a fixed location on the page, regardless of the page rotation
AF_NoView	5	(PDF 1.3) If set, do not display the annotation on the screen or allow it to interact with the user. The annotation may be printed (depending on the setting of the Print flag) but should be considered hidden for purposes of on-screen display and user interaction
AF_Locked	7	(PDF 1.4) If set, do not allow the annotation to be deleted or its properties (including position and size) to be modified by the user. However, this flag does not restrict changes to the annotation’s contents, such as the value of a form field
AF_ToggleNoView	8	(PDF 1.5) If set, invert the interpretation of the NoView flag for

certain events. A typical use is to have an annotation that appears only when a mouse cursor is held over it

bReadOnly
specifies whether a button will be ReadOnly

bRequired
specifies whether a button will be Required

lpszExportVal
specifies an export value. By default this value is "Yes" however you may set your own, with the exception of setting to "Off"

bChecked
specifies whether the checkbox will be checked by default

dwStyle
reserved, must be 0

2.4.3 AcroForms Choice

AcroForms Choice

Top Previous Next

AcroForm drop list functions are:

- PXC_AddChoice
- PXC_Choice_AddItem
- PXC_Choice_AddItems
- PXC_Choice_SelectItems
- PXC_Choice_SetOptions

2.4.3.1 PXC_AddChoice

PXC_AddChoice

Top Previous Next

PXC_AddChoice adds a choice to the PDF page.

```
HRESULT  PXC_AddChoice(
    _PXCPage* page,
    const PXC_RectF* rect,
    PXC_ChoiceKind kind,
    LPCWSTR name,
    _PXCChoice** pChoice
);
```

Parameters

page
[in] *page* Specifies the page object.

rect

[in] *rect* Pointer to a rectangle that will hold new choice coordinates.

kind

[in] *kind* specifies the kind of choice. Can be one of the following values:

Constant	Value	Meaning
`Ch_ListBox`	0	listbox field
`Ch_ComboBox`	1	combobox field

name

[in] *name* Pointer to a null-terminated UNICODE string containing the name of the choice control.

pChoice

[out] *pChoice* specifies a pointer to the variable of the `_PXCChoice*` type, which will represent the choice in the *page* page.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.3.2 PXC_Choice_AddItem

PXC_Choice_AddItem

Top Previous Next

PXC_Choice_AddItem adds one item with values to choice.

```
HRESULT  PXC_Choice_AddItems(
    _PXCChoice* choice,
    LPCWSTR lpcwItemText,
    LPCWSTR lpcwExportValue,
    BOOL bSelected
);
```

Parameters

checkbox

[in] *choice* specifies the checkbox object previously created by the function **PXC_AddCheckBox**.

lpcwItemTexts

[in] *lpcwItemText* - pointer to pointer to a null-terminated UNICODE string containing the text of choice item.

lpcwExportValue

[in] *lpcwExportValue* - pointer to pointer to a null-terminated UNICODE string containing the export value of choice item. Can be NULL.

bSelected

[in] *bSelected* specifies whether to select item by default.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.3.3 PXC_Choice_AddItems

PXC_Choice_AddItems

Top Previous Next

PXC_Choice_AddItems adds items with values to choice.

```
HRESULT  PXC_Choice_AddItems(
    _PXCChoice* choice,
    LPCWSTR* lpcwItemTexts,
    LPCWSTR* lpcwExportValues,
    DWORD dwCount,
    LONG nSelItem
);
```

Parameters

checkbox
[in] *choice* specifies the checkbox object previously created by the function **PXC_AddCheckBox**.

lpcwItemTexts
[in] *lpcwItemTexts* - pointer to pointer to a null-terminated UNICODE string containing the text of choice item.

lpcwExportValues
[in] *lpcwExportValues* - pointer to pointer to a null-terminated UNICODE string containing the export value of choice item. Can be NULL.

dwCount
[in] *dwCount* specifies number of items to add.

nSelItem
[in] *nSelItem* specifies the item index to be selected. Numeration of items is from 0 to count - 1. Possible value -1, in which case no item will be selected.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.3.4 PXC_Choice_SelectItems

PXC_Choice_SelectItems

Top Previous Next

PXC_Choice_SelectItems selects items which already exist in choice.

```
HRESULT  PXC_Choice_SelectItems(
    _PXCChoice* choice,
    DWORD* lpcwItemTexts,
    DWORD dwCount
);
```

Parameters

choice

[in] `choice` specifies the checkbox object previously created by the function **PXC_AddCheckBox**.

`lpcwItemTexts`

[in] *lpcwItemTexts* - pointer to DWORD which represent array of item's indexes to be selected.

dwCount

[in] *dwCount* specifies number of items in array *lpcwItemTexts*.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.3.5 PXC_Choice_SetOptions

PXC_Choice_SetOptions

Top Previous Next

PXC_Choice_SetOptions sets choice options.

```
HRESULT  PXC_Choice_SetOptions(
    _PXCChoice* choice,
    const PXC_ChoiceOptions* options
);
```

Parameters

choice

[in] *choice* specifies the choice object previously created by the function **PXC_AddChoice**.

options

[in] *options* - pointer to the **PXC_ChoiceOptions** which contain information about choice options.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.4 AcroForms Actions

AcroForms Actions

Top Previous Next

These functions add actions to specified AcroForm controls:

- **PXC_FormField_AddGoTo3DAction**
- **PXC_FormField_AddGoToAction**
- **PXC_FormField_AddJSAction**
- **PXC_FormField_AddLaunchAction**
- **PXC_FormField_AddNamedAction**
- **PXC_FormField_AddNamedActionEx**

- PXC_FormField_AddRemoteGoToAction
- PXC_FormField_AddResetFormAction
- PXC_FormField_AddSubmitFormAction
- PXC_FormField_AddURIAction

2.4.4.1 PXC_FormField_AddGoTo3DAction

PXC_FormField_AddGoTo3DAction

Top Previous Next

PXC_FormField_AddGoTo3DAction adds an action pointing to a 3D object in the document.

```
HRESULT  PXC_FormField_AddGoTo3DAction(
    _PXCFormControl* control,
    PXC_FF_TriggerEvent trigger,
    DWORD annot,
    BOOL bUseDefView,
    DWORD dwViewIndex
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

trigger

[in] *trigger* specifies an event upon which the action will begin. May be any one of the following values:

Constant	Value	Definition
FF_Trigger_EnterArea	0	(PDF 1.2) An action to be performed when the cursor enters the annotation's active area
FF_Trigger_ExitArea	1	(PDF 1.2) An action to be performed when the cursor exits the annotation's active area
FF_Trigger_MouseDown	2	(PDF 1.2) An action to be performed when the mouse button is pressed inside the annotation's active area
FF_Trigger_MouseUp	3	(PDF 1.2) An action to be performed when the mouse button is released inside the annotation's active area
FF_Trigger_ReceiveFocus	4	(PDF 1.2) An action to be performed when the annotation receives input focus
FF_Trigger_LoseFocus	5	(PDF 1.2) An action to be performed when the annotation loses input focus

annot

[in] *annot* specifies an index to the 3D annotation object previously added by function **PXC_Add3DAnnotationW**.

bUseDefView

[in] *bUseDefView* specifies whether to use default view after the action is complete.

dwViewIndex

[in] *dwViewIndex* specifies the index for the view to be displayed in a 3D object after the action is complete.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.4.2 PXC_FormField_AddGoToAction

PXC_FormField_AddGoToAction

Top Previous Next

PXC_FormField_AddGoToAction adds goto action to the control.

```
HRESULT  PXC_FormField_AddGoToAction(
    _PXCFormControl* control,
    PXC_FF_TriggerEvent trigger,
    LPCPXC_RectF rect,
    DWORD destPage,
    PXC_OutlineDestination mode
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

trigger

[in] *trigger* specifies an event upon which the action will begin. May be any one of the following values:

Constant	Value	Meaning
FF_Trigger_EnterArea	0	(PDF 1.2) An action to be performed when the cursor enters the annotation’s active area
FF_Trigger_ExitArea	1	(PDF 1.2) An action to be performed when the cursor exits the annotation’s active area
FF_Trigger_MouseDown	2	(PDF 1.2) An action to be performed when the mouse button is pressed inside the annotation’s active area
FF_Trigger_MouseUp	3	(PDF 1.2) An action to be performed when the mouse button is released inside the annotation’s active area
FF_Trigger_ReceiveFocus	4	(PDF 1.2) An action to be performed when the annotation receives input focus
FF_Trigger_LoseFocus	5	(PDF 1.2) An action to be performed when the annotation loses input focus

rect

[in] *rect* specifies the rectangle of the action.

destPage

[in] *destPage* specifies the destination page of the action.

mode

[in] *mode* specifies an outline destination mode as represented by **PXC_OutlineDestination** type.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.4.3 PXC_FormField_AddJSAction

PXC_FormField_AddJSAction

Top Previous Next

PXC_FormField_AddJSAction adds a JavaScript code to be executed by PDF viewer application.

```
HRESULT  PXC_FormField_AddJSAction(
    _PXCFormControl* control,
    PXC_FF_TriggerEvent trigger,
    LPCSTR js
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

trigger

[in] *trigger* specifies an event upon which the action will begin. May be any one of the following values:

Constant	Value	Meaning
FF_Trigger_EnterArea	0	(PDF 1.2) An action to be performed when the cursor enters the annotation's active area
FF_Trigger_ExitArea	1	(PDF 1.2) An action to be performed when the cursor exits the annotation's active area
FF_Trigger_MouseDown	2	(PDF 1.2) An action to be performed when the mouse button is pressed inside the annotation's active area
FF_Trigger_MouseUp	3	(PDF 1.2) An action to be performed when the mouse button is released inside the annotation's active area
FF_Trigger_ReceiveFocus	4	(PDF 1.2) An action to be performed when the annotation receives input focus
FF_Trigger_LoseFocus	5	(PDF 1.2) An action to be performed when the annotation loses input focus

js

[in] *js* specifies the JavaScript code.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.4.4 PXC_FormField_AddLaunchAction

PXC_FormField_AddLaunchAction

Top Previous Next

PXC_FormField_AddLaunchAction adds a launch action to the control.

```
HRESULT  PXC_FormField_AddLaunchAction(
    _PXCFormControl* control,
    PXC_FF_TriggerEvent trigger,
    LPCWSTR application,
    LPCWSTR directory,
    PXC_LaunchOperation operation,
    LPCWSTR parameters
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

trigger

[in] *trigger* specifies an event upon which the action will begin. May be any one of the following values:

Constant	Value	Meaning
FF_Trigger_EnterArea	0	(PDF 1.2) An action to be performed when the cursor enters the annotation’s active area
FF_Trigger_ExitArea	1	(PDF 1.2) An action to be performed when the cursor exits the annotation’s active area
FF_Trigger_MouseDown	2	(PDF 1.2) An action to be performed when the mouse button is pressed inside the annotation’s active area
FF_Trigger_MouseUp	3	(PDF 1.2) An action to be performed when the mouse button is released inside the annotation’s active area
FF_Trigger_ReceiveFocus	4	(PDF 1.2) An action to be performed when the annotation receives input focus
FF_Trigger_LoseFocus	5	(PDF 1.2) An action to be performed when the annotation

loses input focus

application
[in] *application* - pointer to a null-terminated UNICODE string containing the full path to the application.

directory
[in] *directory* - pointer to a null-terminated UNICODE string containing the full path to the directory.

operation
[in] *operation* specifies the launch operation as described by **PXC_LaunchOperation**.

parameters
[in] *parameters* - pointer to a null-terminated UNICODE string containing the parameters string list to be passed to the application.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.4.5 PXC_FormField_AddNamedAction

PXC_FormField_AddNamedAction

Top Previous Next

PXC_FormField_AddNamedAction adds a named action.

```
HRESULT  PXC_FormField_AddNamedAction(
    _PXCFormControl* control,
    PXC_FF_TriggerEvent trigger,
    PXC_FF_Named_Action actioninfo
);
```

Parameters

control
[in] *control* specifies the control object previously created by the related **Add function**, see remark.

trigger
[in] *trigger* specifies an event upon which the action will begin. May be any one of the following values:

Constant	Value	Definition
FF_Trigger_EnterArea	0	(PDF 1.2) An action to be performed when the cursor enters the annotation’s active area
FF_Trigger_ExitArea	1	(PDF 1.2) An action to be performed when the cursor exits the annotation’s active area

FF_Trigger_MouseDown	2	(PDF 1.2) An action to be performed when the mouse button is pressed inside the annotation’s active area
FF_Trigger_MouseUp	3	(PDF 1.2) An action to be performed when the mouse button is released inside the annotation’s active area
FF_Trigger_ReceiveFocus	4	(PDF 1.2) An action to be performed when the annotation receives input focus
FF_Trigger_LoseFocus	5	(PDF 1.2) An action to be performed when the annotation loses input focus

actioninfo

[in] *actioninfo* specifies action information which may be any one of the following value:

Constant	Value	Definition
AN_NextPage	0	go to the next page of the document
AN_PrevPage	1	go to the previous page of the document
AN_FirstPage	2	go to the first page of the document
AN_LastPage	3	go to the last page of the document

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.4.6 PXC_FormField_AddNamedActionEx

PXC_FormField_AddNamedActionEx

Top Previous Next

PXC_FormField_AddNamedActionEx adds a named action.

```
HRESULT  PXC_FormField_AddNamedActionEx(
    _PXCFormControl* control,
    PXC_FF_TriggerEvent trigger,
    LPCSTR name
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

trigger

[in] *trigger* specifies the event upon which the action will begin. May be any one of the following values:

Constant	Value	Definition

`FF_Trigger_EnterArea`	0	(PDF 1.2) An action to be performed when the cursor enters the annotation’s active area
`FF_Trigger_ExitArea`	1	(PDF 1.2) An action to be performed when the cursor exits the annotation’s active area
`FF_Trigger_MouseDown`	2	(PDF 1.2) An action to be performed when the mouse button is pressed inside the annotation’s active area
`FF_Trigger_MouseUp`	3	(PDF 1.2) An action to be performed when the mouse button is released inside the annotation’s active area
`FF_Trigger_ReceiveFocus`	4	(PDF 1.2) An action to be performed when the annotation receives input focus
`FF_Trigger_LoseFocus`	5	(PDF 1.2) An action to be performed when the annotation loses input focus

name

[in] *name* specifies an action name to be passed to the viewer application.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.4.7 PXC_FormField_AddRemoteGoToAction

PXC_FormField_AddRemoteGoToAction

Top Previous Next

PXC_FormField_AddRemoteGoToAction adds remote goto action to the control.

```
HRESULT  PXC_FormField_AddRemoteGoToAction(
    _PXCFormControl* control,
    PXC_FF_TriggerEvent trigger,
    LPCWSTR file,
    BOOL bNewWindow,
    LPCPXC_RectF rect,
    DWORD destPage,
    PXC_OutlineDestination mode
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related**Add function**, see remark.

trigger

[in] *trigger* specifies an event upon which the action should begin. May be any one of the following

values:

Constant	Value	Definition
FF_Trigger_EnterArea	0	(PDF 1.2) An action to be performed when the cursor enters the annotation’s active area
FF_Trigger_ExitArea	1	(PDF 1.2) An action to be performed when the cursor exits the annotation’s active area
FF_Trigger_MouseDown	2	(PDF 1.2) An action to be performed when the mouse button is pressed inside the annotation’s active area
FF_Trigger_MouseUp	3	(PDF 1.2) An action to be performed when the mouse button is released inside the annotation’s active area
FF_Trigger_ReceiveFocus	4	(PDF 1.2) An action to be performed when the annotation receives input focus
FF_Trigger_LoseFocus	5	(PDF 1.2) An action to be performed when the annotation loses input focus

file

[in] *file* - pointer to a null-terminated UNICODE string containing the full path name to the file.

bNewWindow

[in] *bNewWindow* specifies whether to open the destination document in a new window.

rect

[in] *trigger* specifies the rectangle of the action.

destPage

[in] *trigger* specifies the destination page of the action.

mode

[in] *trigger* specifies the outline destination mode as represented by **PXC_OutlineDestination** type.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.4.8 PXC_FormField_AddResetFormAction

PXC_FormField_AddResetFormAction

Top Previous Next

PXC_FormField_AddResetFormAction adds a reset form action to reset a form field.

```
HRESULT  PXC_FormField_AddResetFormAction(
    _PXCFormControl* control,
    PXC_FF_TriggerEvent trigger,
    _PXCFormControl** lppFields,
    DWORD dwFieldsCount
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

trigger

[in] *trigger* specifies an event upon which the action will begin. May be any one of the following values:

Constant	Value	Definition
FF_Trigger_EnterArea	0	(PDF 1.2) An action to be performed when the cursor enters the annotation’s active area
FF_Trigger_ExitArea	1	(PDF 1.2) An action to be performed when the cursor exits the annotation’s active area
FF_Trigger_MouseDown	2	(PDF 1.2) An action to be performed when the mouse button is pressed inside the annotation’s active area
FF_Trigger_MouseUp	3	(PDF 1.2) An action to be performed when the mouse button is released inside the annotation’s active area
FF_Trigger_ReceiveFocus	4	(PDF 1.2) An action to be performed when the annotation receives input focus
FF_Trigger_LoseFocus	5	(PDF 1.2) An action to be performed when the annotation loses input focus

lppFields

[in] *lppFields* specifies a controls list to be reset.

dwFieldsCount

[in] *dwFieldsCount* retrieves the number of form fields passed to *lppFields*

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.4.9 PXC_FormField_AddSubmitFormAction

PXC_FormField_AddSubmitFormAction

Top Previous Next

PXC_FormField_AddSubmitFormAction adds a submit form action.

```
HRESULT PXC_FormField_AddSubmitFormAction(
    _PXCFormControl* control,
    PXC_FF_TriggerEvent trigger,
    const PXC_FF_SubmitAction* sa
);
```

Parameters

control
[in] *control* specifies the control object previously created by the related **Add function**, see remark.

trigger
[in] *trigger* specifies an event upon which the action will begin. May be any one of the following values:

Constant	Value	Definition
FF_Trigger_EnterArea	0	(PDF 1.2) An action to be performed when the cursor enters the annotation's active area
FF_Trigger_ExitArea	1	(PDF 1.2) An action to be performed when the cursor exits the annotation's active area
FF_Trigger_MouseDown	2	(PDF 1.2) An action to be performed when the mouse button is pressed inside the annotation's active area
FF_Trigger_MouseUp	3	(PDF 1.2) An action to be performed when the mouse button is released inside the annotation's active area
FF_Trigger_ReceiveFocus	4	(PDF 1.2) An action to be performed when the annotation receives input focus
FF_Trigger_LoseFocus	5	(PDF 1.2) An action to be performed when the annotation loses input focus

sa
[in] *sa* specifies submit action properties as described in the **PXC_FF_SubmitAction** type.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.4.10 PXC_FormField_AddURIAction

PXC_FormField_AddURIAction

Top Previous Next

PXC_FormField_AddURIAction adds a URL action to the control.

```
HRESULT  PXC_FormField_AddURIAction(
    _PXCFormControl* control,
    PXC_FF_TriggerEvent trigger,
    LPCSTR uri
);
```

Parameters

control

[in] *control* specifies the control object previously created by the related **Add function**, see remark.

trigger

[in] *trigger* specifies an event upon which an action should act. May be any one of the following values:

Constant	Value	Definition
FF_Trigger_EnterArea	0	(PDF 1.2) An action to be performed when the cursor enters the annotation’s active area
FF_Trigger_ExitArea	1	(PDF 1.2) An action to be performed when the cursor exits the annotation’s active area
FF_Trigger_MouseDown	2	(PDF 1.2) An action to be performed when the mouse button is pressed inside the annotation’s active area
FF_Trigger_MouseUp	3	(PDF 1.2) An action to be performed when the mouse button is released inside the annotation’s active area
FF_Trigger_ReceiveFocus	4	(PDF 1.2) An action to be performed when the annotation receives input focus
FF_Trigger_LoseFocus	5	(PDF 1.2) An action to be performed when the annotation loses input focus

uri

[in] *uri* - pointer to a null-terminated UNICODE string containing the URI.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

Parameter *control* represents a control which may be retrieved by calling any one of the following functions: **PXC_AddPushButton**, **PXC_AddCheckBox**, **PXC_AddRadioButton**, **PXC_AddTextBox** or **PXC_AddChoice**.

2.4.5 AcroForms Push Button

AcroForms Push Button

Top Previous Next

These functions add AcroForm push buttons to the PDF content:

- **PXC_AddPushButton**
- **PXC_PushButton_SetCaption**
- **PXC_PushButton_SetIcon**
- **PXC_PushButton_SetIconEx**
- **PXC_PushButton_SetOptions**
- **PXC_PushButton_GetOptions**

2.4.5.1 PXC_AddPushButton

PXC_AddPushButton

Top Previous Next

PXC_AddPushButton adds a push button to the PDF document.

```
HRESULT  PXC_AddPushButton(
    _PXCPage* page,
    const PXC_RectF* rect,
    LPCWSTR name,
    _PXCPushButton** pButton
);
```

Parameters

page
[in] *page* Specifies the page object previously created by the function **PXC_AddPage**.

rect
[in] *rect* Pointer to a rectangle that will hold new push button coordinates.

name
[in] *name* Pointer to a null-terminated UNICODE string containing the name of the push button control.

pButton
[out] *pButton* specifies a pointer to the variable of the `_PXCPushButton*` type, which will represent the push button in the *page* page.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

For the image to be displayed on the page, it is necessary to place it there using the **PXC_PlaceImage** function. If not 'placed' - the image is considered orphaned and will be deleted from the document once written to disk.

2.4.5.2 PXC_PushButton_SetCaption

PXC_PushButton_SetCaption

Top Previous Next

PXC_PushButton_SetCaption sets a caption for a push button.

```
HRESULT  PXC_PushButton_SetCaption(
    _PXCPushButton* button,
    LPCWSTR lpcwCaption
);
```

Parameters

button
[in] *button* Specifies the push button object previously created by the function **PXC_AddPushButton**.

lpcwCaption
[in] *lpcwCaption* - pointer to a null-terminated UNICODE string containing the caption of the push button.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

For the image to be displayed on the page, it is necessary to place the image using the **PXC_PlaceImage** function. If not 'placed' - the image is considered orphaned and will be deleted from the document once written to disk.

2.4.5.3 PXC_PushButton_SetIcon

PXC_PushButton_SetIcon

Top Previous Next

PXC_PushButton_SetIcon sets an icon to a push button.

```
HRESULT  PXC_PushButton_SetIcon(
    _PXCPushButton* button,
    _PXCImage* img
);
```

Parameters

button
[in] *button* specifies the push button object previously created by the function **PXC_AddPushButton**.

img
[in] *img* - pointer to an image object returned by function such as **PXC_AddImageA**.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.5.4 PXC_PushButton_SetIconEx

PXC_PushButton_SetIconEx

Top Previous Next

PXC_PushButton_SetIconEx sets an icon to a push button.

```
HRESULT  PXC_PushButton_SetIconEx(
    _PXCPushButton* button,
    const PXC_FF_IconAppearance* iconinfo
);
```

Parameters

button

[in] *button* specifies the push button object previously created by the function **PXC_AddPushButton**.

iconinfo

[in] *iconinfo* - pointer to the **PXC_FF_IconAppearance** which contains information about icon and positioning.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.5.5 PXC_PushButton_SetOptions

PXC_PushButton_SetOptions

Top Previous Next

PXC_PushButton_SetOptions sets push button options.

```
HRESULT  PXC_PushButton_SetOptions(
    _PXCPushButton* button,
    const PXC_PushButton_Options* options
);
```

Parameters

button

[in] *button* specifies the push button object previously created by the function **PXC_AddPushButton**.

options

[in] *options* - pointer to the **PXC_PushButton_Options** which contains information about the push button options.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.5.6 PXC_PushButton_GetOptions

PXC_PushButton_GetOptions

Top Previous Next

PXC_PushButton_GetOptions retrieves push button options.

```
HRESULT  PXC_PushButton_GetOptions(
    _PXCPushButton* button,
    PXC_PushButton_Options* options
);
```

Parameters

button
: [in] *button* specifies the push button object previously created by the function **PXC_AddPushButton**.

options
: [in] *options* - pointer to the **PXC_PushButton_Options** structure which will receive information about push button options.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.6 AcroForms Radio Button

AcroForms Radio Button

Top Previous Next

These functions add multiple-choice Radio buttons to the AcroForm:

- **PXC_AddRadioButton**
- **PXC_AddradioButtonList**
- **PXC_RadioButton_SetCheck**
- **PXC_RadioButton_SetFlags**
- **PXC_RadioButton_SetOnStateValue**
- **PXC_RadioButtonList_GetOptions**
- **PXC_RadioButtonList_SetOptions**

2.4.6.1 PXC_AddradioButtonList

PXC_AddradioButtonList

Top Previous Next

PXC_AddradioButtonList adds a radio button list which will contain radio buttons.

```
HRESULT  PXC_AddradioButtonList(
    _PXCDocument* doc,
    LPCWSTR name,
    _PXCRadioButtonList** pRadioButtonGroup
);
```

Parameters

doc
: [in] *doc* specifies the PDF object previously created by the function **PXC_NewDocument**.

name
: [in] *name* - pointer to a null-terminated UNICODE string containing the name of the radio button list.

pRadioButtonGroup
: [out] *pRadioButtonGroup* specifies a pointer to the variable of the `_PXCRadioButtonList` type, which will represent the radio button list in the *doc* page.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.6.2 PXC_AddRadioButton

PXC_AddRadioButton

Top Previous Next

PXC_AddRadioButton adds a radio button to the PDF page.

```
HRESULT  PXC_AddRadioButton(
    _PXCPage* page,
    _PXCRadioButtonList* radiogroup,
    const PXC_RectF* rect,
    _PXCRadioButton** pRadioButton
);
```

Parameters

page
[in] *page* specifies the page object.

radiogroup
[in] *radiogroup* specifies the radio button list previously created by the function **PXC_AddRadioButtonList**.

rect
[in] *rect* - pointer to a rectangle that will hold new radio button coordinates.

pRadioButton
[out] *pRadioButton* specifies a pointer to the variable of the `_PXCRadioButton` type, which will represent the radio button in the *page* page.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.6.3 PXC_RadioButton_SetCheck

PXC_RadioButton_SetCheck

Top Previous Next

PXC_RadioButton_SetCheck sets radiobutton **Checked** state.

```
HRESULT  PXC_RadioButton_SetCheck(
    _PXCRadioButton* radio,
    BOOL bChecked
);
```

Parameters

radio
[in] *radio* specifies the radiobutton object previously created by the function **PXC_AddRadioButton**.

bChecked

[in] *bChecked* specifies whether radiobutton will be checked by default.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.6.4 PXC_RadioButton_SetFlags

PXC_RadioButton_SetFlags

Top Previous Next

PXC_RadioButton_SetFlags sets radiobutton flags.

```
HRESULT  PXC_RadioButton_SetFlags(
    _PXCRadioButton* radio,
    const PXC_AnnotsFlags Flags
);
```

Parameters

radio

[in] *radio* specifies the radiobutton object previously created by the function **PXC_AddRadioButton**.

Flags

[in] *Flags* specifies flags as described in **PXC_AnnotsFlags**.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.6.5 PXC_RadioButton_SetOnStateValue

PXC_RadioButton_SetOnStateValue

Top Previous Next

PXC_RadioButton_SetOnStateValue sets a radio button **On** state, the value of which by default is "Yes" according to the PDF specification.

```
HRESULT  PXC_RadioButton_SetOnStateValue(
    _PXCRadioButton* radio,
    LPCWSTR lpcwOnStateVal
);
```

Parameters

radio

[in] *radio* specifies the checkbox object previously created by the function **PXC_AddCheckBox**.

lpcwOnStateVal

[in] *lpcwOnStateVal* - pointer to a null-terminated UNICODE string containing the name of the **On** state value.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.6.6 PXC_RadioButtonList_GetOptions

PXC_RadioButtonList_GetOptions

Top Previous Next

PXC_RadioButtonList_GetOptions retrieves radio button group options.

```
HRESULT  PXC_RadioButtonList_GetOptions(
    _PXCRadioButtonList* radiogroup,
    PXC_RadioButton_Options* options
);
```

Parameters

radiogroup
[in] *radiogroup* specifies the radio button list object previously created by the function **PXC_AddRadioButtonList**.

options
[in] *options* - pointer to the **PXC_RadioButton_Options** which will receive information refarding radio button options.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.6.7 PXC_RadioButtonList_SetOptions

PXC_RadioButtonList_SetOptions

Top Previous Next

PXC_RadioButtonList_SetOptions sets radio button group options.

```
HRESULT  PXC_RadioButtonList_SetOptions(
    _PXCRadioButtonList* radiogroup,
    const PXC_RadioButton_Options* options
);
```

Parameters

radiogroup

[in] *radiogroup* specifies the radio button list object previously created by the function **PXC_AddRadioButtonList**.

options

[in] *options* - pointer to the **PXC_RadioButton_Options** which contains information regarding radio button options.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.7 AcroForms Text Box

AcroForms Text Box

Top Previous Next

These functions add an AcroForm Text Box to the PDF content:

- **PXC_AddTextBox**
- **PXC_TextBox_SetCaption**
- **PXC_TextBox_SetOptions**

2.4.7.1 PXC_AddTextBox

PXC_AddTextBox

Top Previous Next

PXC_AddTextBox adds a text box to the PDF page.

```
HRESULT  PXC_AddTextBox(
    _PXCPage* page,
    const PXC_RectF* rect,
    LPCWSTR name,
    _PXCTextBox** pTextBox
);
```

Parameters

page

[in] *page* Specifies the page object.

rect

[in] *rect* Pointer to a rectangle that will hold new push button coordinates.

name

[in] *name* Pointer to a null-terminated UNICODE string containing the name of the push button control.

pTextBox

[out] *pTextBox* specifies a pointer to the variable of the `_PXCPushButton*` type, which will represent the push button in the *page* page.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.

To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.4.7.2 PXC_TextBox_SetCaption

PXC_TextBox_SetCaption

Top Previous Next

PXC_TextBox_SetCaption sets a caption for a textbox.

```
HRESULT  PXC_TextBox_SetCaption(
    _PXCPushButton* textbox,
    LPCWSTR lpcwCaption
);
```

Parameters

textbox

[in] *textbox* Specifies the textbox object previously created by the function **PXC_AddTextBox**.

lpcwCaption

[in] *lpcwCaption* Pointer to a null-terminated UNICODE string containing the caption of the textbox.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Remarks

For the image to be displayed on the page, it is necessary for the image to be placed using the **PXC_PlaceImage** function. If not 'placed' - the image is considered orphaned and will be deleted from the document once written to disk.

2.4.7.3 PXC_TextBox_SetOptions

PXC_TextBox_SetOptions

Top Previous Next

PXC_TextBox_SetOptions sets textbox options.

```
HRESULT  PXC_TextBox_SetOptions(
    _PXCTextBox* textbox,
    const PXC_TextBox_Options* options
);
```

Parameters

textbox

[in] *textbox* specifies the textbox object previously created by the function **PXC_AddTextBox**.

options

[in] *options* - pointer to the **PXC_TextBox_Options** which contains information about textbox options.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

2.5 Memory Management

2.5.1 PXC_Alloc

PXC_Alloc

Top Previous Next

PXC_Alloc allocates a memory block of the specified size with the help of memory manager **PDF-XChange Library**.

```
void*  PXC_Alloc(
    UINT sz
);
```

Parameters

sz

[in] *sz* specifies the size of the memory block to be allocated.

Return Values

PXC_Alloc returns a void pointer to the allocated space, or **NULL** if there is insufficient memory available.

Note

Memory allocated with the **PXC_Alloc** must be freed using the function **PXC_Free**.

Example (C++).

```
// Allocate 100 byte memory block

    BYTE* pBlock = (BYTE*)PXC_Alloc(100);

    if (pBlock == NULL)
    {
        // Handle error
        ...
    }
    ...
```

2.5.2 PXC_Free

PXC_Free

Top Previous Next

PXC_Free frees the memory block previously allocated by the function **PXC_Alloc**.

```
void  PXC_Free(
    void* ptr
);
```

Parameters

ptr

[in] Previously allocated memory block to be freed.

Return Values

None.

Example (C++).

```
// Example shows that all memory allocated by 'PXC_Alloc' function
// should be freed by the PXC_Free function!

    // Allocate 100 byte memory block

    BYTE* pBlock = (BYTE*)PXC_Alloc(100);

    if (pBlock == NULL)
    {
        // Handle error
        ...
    }
    ...

    // When pBlock data is no more necessary then:

    PXC_Free(pBlock);
```

2.6 Viewing/Display Options

Viewing/Display Options

Top Previous Next

Please note the functions in this section do not relate to the PDF-XChange Viewer SDK specifically - but the way in which the PDF files you create should be presented when viewed in the PDF-XChange Viewer - or any other Viewing tool - such as Adobe Acrobat Reader when viewed by a user.

2.6.1 Common Structures

2.6.1.1 PXC_3DView (COPY)

PXC_3DView

Top Previous Next

The **PXC_3DView** structure specifies parameters for a U3D View. For more information please refer to the

PDF Specification V1.5.

```
typedef struct _PXC_3DView {
   DWORD m_cbSize;
   WCHAR m_ExtName[128];
   double m_C2W[12];
   double m_CO;
   double m_FOV;
   COLORREF m_BackColor;
} PXC_3DView;
```

Members

m_cbSize
: Specifies the size of the structure and is provided for compatibility with future versions of **PDF-XChange** where this structure may be modified. Should be equal to the `size of(PXC_3DView)`.

m_ExtName
: Specifies the external name of the 3D view (which will be displayed within the viewer application). If this field is empty (`m_ExtName[0] == 0`), default value **`Default`** will be used.

m_C2W
: Defines the matrix that specifies a position and orientation of the camera in world coordinates.

m_CO
: Specifies distance to the center of orbit.

m_FOV
: Specifies the angle of view of the camera (in degrees).

m_BackColor
: Defines the background color for U3D object.

2.6.1.2 PXC_BorderInfo

PXC_BorderInfo

Top Previous Next

The **PXC_BorderInfo** structure specifies an interactive form field border information.

```
typedef struct _PXC_BorderInfo {
   PXC_AnnotBorder AnnotBorder;
   COLORREF BorderColor;
} PXC_BorderInfo;
```

Members

AnnotBorder
: Specifies the **PXC_AnnotBorder** type.

BorderColor
: Specifies the color of the border.

2.6.1.3 PXC_ChoiceOptions

PXC_ChoiceOptions

Top Previous Next

The **PXC_ChoiceOptions** structure specifies checkbox options.

```
typedef struct _PXC_ChoiceOptions {
   BOOL bEdit;
   BOOL bSort;
   BOOL bMultiselect;
   BOOL bDoNotSpellCheck;
   BOOL bCommitOnSelChange;
} PXC_ChoiceOptions;
```

Members

bEdit
: If TRUE, the combo box includes an editable text box as well as a dropdown list, if FALSE, it includes only a drop-down list. This flag is meaningful only for combobox control

bSort
: If TRUE, the field's option items should be sorted alphabetically

bMultiselect
: (PDF 1.4) If TRUE, more than one of the field's option items may be selected simultaneously, if FALSE, no more than one item at any time may be selected

bDoNotSpellCheck
: (PDF 1.4) If TRUE, text entered in the field is not spell-checked. This flag is meaningful only for a Combobox and only if set as TRUE

bCommitOnSelChange
: (PDF 1.5) If TRUE, the new value is committed as soon as a selection is made with the pointing device. This option enables applications to perform an action once a selection is made, without requiring the user to exit the field. If FALSE, the new value is not committed until the user exits the field

2.6.1.4 PXC_CommonAnnotInfo (COPY)

PXC_CommonAnnotInfo

Top Previous Next

The **PXC_CommonAnnotInfo** structure specifies a matrix of coordinate system transformation.

```
typedef struct _PXC_CommonAnnotInfo {
   double m_Opacity;
   COLORREF m_Color;
   DWORD m_Flags;
   PXC_AnnotBorder m_Border;
} PXC_CommonAnnotInfo;
```

Members

m_Opacity

Defines the annotation opacity level in the document. Will be used only if the PDF specification version is `1.4` or higher (**PXC_SetSpecVersion**).

m_Color

Defines the annotation color. This color will be used as the color for the following:

- The background of the annotation's icon when closed
- The title bar of the annotation's pop-up window
- The border of the annotation link

m_Flags

A set of flags specifying various characteristics for the annotation. May be combination of the following flags:

Flag	Description
AF_Invisible	If set, do not display the annotation.
AF_Hidden	(*PDF 1.2*) If set, do not display or print the annotation or allow it to interact with the user, regardless of its annotation type.
AF_Print	(*PDF 1.2*) If set, print the annotation when the page is printed. If clear, never print the annotation, regardless of whether it is displayed on the screen. This can be useful, for example, for annotations representing an interactive link, which would serve no meaningful purpose on the printed page.
AF_NoZoom	(*PDF 1.3*) If set, do not scale the annotation’s appearance to match the magnification of the page. The location of the annotation on the page (defined by the upper-left corner of its annotation rectangle) remains fixed, regardless of the page magnification.
AF_NoRotate	(*PDF 1.3*) If set, do not rotate the annotation’s appearance to match the rotation of the page. The upper-left corner of the annotation rectangle remains in a fixed location on the page, regardless of the page rotation.
AF_NoView	(*PDF 1.3*) If set, do not display the annotation on the screen or allow it to interact with the user. The annotation may be printed (depending on the setting of the Print flag), but should be considered hidden for purposes of on-screen display and user interaction.
AF_ReadOnly	(*PDF 1.3*) If set, do not allow the annotation to interact with the user. The annotation may be displayed or printed (depending on the settings of the NoView and Print flags), but should not respond to mouse clicks or change its appearance in response to mouse motions.
AF_Locked	(*PDF 1.4*) If set, do not allow the annotation to be deleted or its properties (including position and size) to be modified by the user. However, this does not restrict changes to the annotation’s contents.
AF_ToggleNoView	(*PDF 1.5*) If set, invert the interpretation of the NoView flag for certain events. A typical use is to have an annotation that appears only when a mouse cursor is held over it.

PXC_AnnotBorder structure

The structure specifies a matrix for the coordinate system transformation.

```
typedef struct _PXC_AnnotBorder {
    double m_Width;
    PXC_AnnotBorderStyle m_Type;
    DWORD m_DashCount;
    double* m_DashArray;
} PXC_AnnotBorder;
```

Members

m_Width

The border width in points. If this value is 0, no border is drawn.

m_Type

The border style. May be any one of the following values:

Value	Meaning
ABS_Solid	A solid rectangle surrounding the annotation.
ABS_Dashed	A dashed rectangle surrounding the annotation. The dash pattern is specified by the *m_DashArray* field.
ABS_Bevel	A simulated embossed rectangle that appears to be raised above the surface of the page.
ABS_Inset	A simulated engraved rectangle that appears to be recessed below the surface of the page.
ABS_Underline	A single line along the bottom of the annotation rectangle.

Note: For links (**PXC_AddLink**) only `ABS_Solid` or `ABS_Dashed` style can be used.

m_DashCount

Specifies the number of items in the *m_DashArray* array.

m_DashArray

Pointer to the `double` values array that define the dash pattern for border drawing (for more details please see **PXC_SetPolyDash**).

2.6.1.5 PXC_CommonFieldAppearance

PXC_CommonFieldAppearance

Top Previous Next

The **PXC_CommonFieldAppearance** structure specifies appearance properties for interactive form fields.

```
typedef struct _PXC_CommonFieldAppearance {
    COLORREF crBgColor;
    DWORD dwFontSize;
    DWORD dwFontID;
    COLORREF crTextColor;
} PXC_CommonFieldAppearance;
```

Members

crBgColor

Specifies the background color.

dwFontSize

Specifies the font size.

dwFontID
Specifies the font idenifier returned by function **PXC_AddFont**.

crTextColor
Specifies the text color.

2.6.1.6 PXC_DrawTextStruct

PXC_DrawTextStruct

Top Previous Next

The **PXC_DrawTextStruct** structure specifies a matrix of coordinate system transformation.

```
typedef struct _PXC_DrawTextStruct {
   DWORD cbSize;
   DWORD mask;
// Returned values
   double endY;
   DWORD usedChars;
// Text format settings
   double lineSpace;
   double paraSpace;
   double paraIndent;
   DWORD fontID;
   double fontSize;
   DWORD newlineMode;
   double charSpace;
   double wordSpace;
   double textScale;
} PXC_DrawTextStruct;
```

Members

cbSize
[in] Specifies structure size in bytes.

mask
[in] Specifies which fields in structure a set to proper value. If one of fields are not set current value of corresponding text parameter will be used.
Value of this field may be any combination of following flags:

Constant	Value	Element	Value if not set
DTSF_LineSpace	0x0001	**lineSpace**	Current font size
DTSF_ParagraphSpace	0x0002	**paraSpace**	Current font size
DTSF_ParagraphIndent	0x0004	**paraIndent**	0.0
DTSF_FontID	0x0008	**fontID**	Current font
DTSF_FontSize	0x0010	**fontSize**	Current font size
DTSF_NewLine	0x0020	**newlineMode**	DTNL_NewParagraph

`DTSF_CharSpace`	`0x0040`	**`charSpace`**	Current character spacing
`DTSF_WordSpace`	`0x0080`	**`wordSpace`**	Current word spacing
`DTSF_textScale`	`0x0100`	**`textScale`**	Current text scaling

endY

[out] Return Y position of last text line.

Note: When `DTF_CalcOnly` flag is set vertical alignement is ignored and return value always will be calulated using `DTF_Align_Top`.

usedChars

[out] Return count of used characters. This value does not include skipped characters from buffer begining.

lineSpace

[in] This value specify vertical interval between lines. See table below for details.

Input value	**Resulting interval**
0.0 or not set	Current font size
Any positive (greater than 0.0)	Interval in points
Any negative (less than 0.0)	Modulus is interval in percents of current font size. For example text will be shown using font size `12` points and **lineSpace** is -125. Then interval will be `12*|-125|/100=15` points.

paraSpace

[in] This value specify vertical interval between paragraphs, actually between last line of previous paragraph and first line of current. See table below for details.

Input value	**Resulting interval**
0.0 or not set	Current font size
Any positive (greater than 0.0)	Interval in points
Any negative (less than 0.0)	Modulus is interval in percents of current font size. For example text will be shown using font size `12` points and **paraSpace** is -150. Then interval will be `12*|-150|/100=18` points.

paraIndent

[in] This value specify paragraph indent - indent for first line of paragraph. See table below for details.

Input value	**Resulting interval**
0.0 or not set	No indent
Any positive (greater than 0.0)	Indent in points
Any negative (less than 0.0)	Modulus multiplied by width of space character. Spcae character width is calculating using current font id, font size, character spasing and text scaling, but not using word spacing. For example text will be shown with space character width `8` points and (and **paraSpace** is -3.5. Then indent will be `8*|-3.5|=28` points.

fontID

[in] This value specify font id. If it is not set or zero current font will be used.

fontSize

[in] This value specify font size. If it is not set or zero current font size will be used.

newlineMode

[in] This value specify how newline characters will be threated. It may be one of following values.

Constant	Value	Description
`DTNL_NewParagraph`	0	Each newline character begin new paragraph
`DTNL_None`	1	Double newline character begin new paragraph, single ignored
`DTNL_Space`	2	Double newline character begin new paragraph, single converted to space
`DTNL_SingleSpace`	3	Double newline character begin new paragraph, single converted to space if there was no space character before or after it, otherwise ignored

If it is not set or invalid `DTNL_NewParagraph` value assumed.

Note: There are two newline characters 'CR' (Carriage return) and 'LF' (Line Feed). Function **PXC_DrawTextExW** treats any of following combinations as single new line characters: single 'CR', single 'LF', 'CR' + 'LF', 'LF' + 'CR', however combinations of 'CR' + 'CR' and 'LF' + 'LF' will be treated as two newline characters.

charSpace

[in] This value specify character spacing. If it is not set current character spacing used. Note that after returning from **PXC_DrawTextExW** previous character spacing will be restored.

wordSpace

[in] This value specify word spacing. If it is not set current word spacing used. Note that after returning from **PXC_DrawTextExW** previous word spacing will be restored.

textScale

[in] This value specify text scaling. If it is not set current text scaling used. Note that after returning from **PXC_DrawTextExW** previous text scaling will be restored.

2.6.1.7 PXC_FF_IconAppearance

PXC_FF_IconAppearance

Top Previous Next

The **PXC_FF_IconAppearance** structure specifies an icon's appearance.

```
typedef struct _PXC_FF_IconAppearance {
   _PXCImage* xcIcon;
   PXC_PB_IconFit FitOpts;
} PXC_FF_IconAppearance;
```

Members

xcIcon

specifies the image handler returned by a function such as (but not limited to) PXC_AddImageA.

FitOpts

specifies type PXC_PB_IconFit.

2.6.1.8 PXC_FF_SubmitAction

PXC_FF_SubmitAction

Top Previous Next

The **PXC_FF_SubmitAction** structure specifies the submit action properties

```
typedef struct _PXC_FF_SubmitAction {
   LPCSTR lpcszUrl;
   _PXCFormControl** lppFields;
   DWORD dwFieldsCount;
   PXC_FF_SendMethod smethod;
   BOOL bSendClickCoordinates;
   BOOL bSubmitAsPDF;
} PXC_FF_SubmitAction;
```

Members

lpcszUrl

specifies a URL to which the information will be Sent/posted

lppFields

specifies the form fields to send

dwFieldsCount

specifies the form fields count

smethod

specifies the send/post method. may be any one of the following values:

Constant	Value	Meaning
FF_SendMethod_GET	0	If set, field names and values are submitted using an HTTP GET request
FF_SendMethod_POST	1	If set, field names and values are submitted using a HTTP POST request

bSendClickCoordinates

If set, the coordinates of the mouse click that caused the submit-form action are transmitted as part of the form data. The coordinate values are relative to the upper-left corner of the field’s widget annotation rectangle.

bSubmitAsPDF

If set, the document is submitted as PDF, using the MIME content type `application/pdf`.

2.6.1.9 PXC_FontInfo

PXC_FontInfo

Top Previous Next

The **PXC_FontInfo** structure contains metrics describing font. All sizes are specified in points, and depends from used font size.

```
typedef struct _PXC_FontInfo {
   DWORD cbSize;
   double ftmHeight;
   double ftmAscent;
   double ftmDescent;
   double ftmILead;
   double ftmELead;
   double fotmAscent;
   double fotmDescent;
   double fotmLineGap;
   double fotmMacAscent;
   double fotmMacDescent;
   double fotmMacLineGap;
   PXC_RectF fontBox;
} PXC_FontInfo;
```

Members

cbSize
: Specifies the size of the structure and is provided for compatibility with future versions of PDF-XChange where this structure may be modified.

ftmHeight
: Specifies the height (ascent + descent) of characters.

ftmAscent
: Specifies the ascent (units above the base line) of characters.

ftmDescent
: Specifies the descent (units below the base line) of characters.

ftmILead
: Specifies the amount of leading (space) inside the bounds set by the **ftmHeight** member. Accent marks and other diacritical characters may occur in this area. The designer may set this member to zero.

ftmELead
: Specifies the amount of extra leading (space) that the application adds between rows. Since this area is outside the font, it contains no marks and is not altered by text output calls. The designer may set this member to zero.

fotmAscent

Specifies the maximum distance characters in this font extend above the base line. This is the typographic ascent for the font.

fotmDescent
Specifies the maximum distance characters in this font extend below the base line. This is the typographic descent for the font.

fotmLineGap
Specifies typographic line spacing.

fotmMacAscent
Specifies the maximum distance characters in this font extend above the base line for the Macintosh computer.

fotmMacDescent
Specifies the maximum distance characters in this font extend below the base line for the Macintosh computer.

fotmMacLineGap
Specifies line-spacing information for the Macintosh computer.

fontBox
Specifies the bounding box for the font.

2.6.1.10 PXC_Matrix

PXC_Matrix

Top Previous Next

The **PXC_Matrix** structure specifies a matrix of the coordinate system transformation.

```
typedef struct _PXC_Matrix {
   double a;
   double b;
   double c;
   double d;
   double e;
   double f;
} PXC_Matrix;
```

Members

a
Specifies the following.

Operation	Description
Scaling	Horizontal scaling component
Rotation	Cosine of a rotation angle
Reflection	Horizontal component

b
Specifies the following.

Operation	Description
Shear	Horizontal proportionality constant
Rotation	Sine of rotation angle

c

Specifies the following.

Operation	Description
Shear	Vertical proportionality constant
Rotation	Negative sine of rotation angle

d

Specifies the following.

Operation	Description
Scaling	Vertical scaling component
Rotation	Cosine of rotation angle
Reflection	Vertical reflection component

e

Specifies the horizontal translation component, in points.

f

Specifies the vertical translation component, in points.

Remarks

The following list describes how the members are used for each operation.

Operation	a	b	c	d
Rotation	Cosine	Sine	Negative sine	Cosine
Scaling	Horizontal scaling component	Not used	Not used	Vertical scaling component
Shear	Not used	Horizontal Proportionality Constant	Vertical Proportionality Constant	Not used
Reflection	Horizontal Reflection Component	Not used	Not used	Vertical Reflection Component

2.6.1.11 PXC_PB_IconFit

PXC_PB_IconFit

Top Previous Next

The **PXC_PB_IconFit** structure specifies push button icon fit options.

```
typedef struct _PXC_PB_IconFit {
   PXC_PB_IconScalingMode IconScalingMode;
   PXC_PB_IconScalingType IconScalingType;
   double Spacing[2];
   BOOL bFit;
} PXC_PB_IconFit;
```

Members

IconScalingMode

Specifies the icon scaling mode. This field may be any one of the following values:

Constant	Value	Description
PB_ISM_AlwaysScale	0	Always scale
PB_ISM_WhenIconBigger	1	Scale only when the icon is bigger than the annotation rectangle
PB_ISM_WhenIconSmaller	2	Scale only when the icon is smaller than the annotation rectangle
PB_ISM_NeverScale	3	Never scale

IconScalingType

Specifies the icon scaling type. This field can be one of the following values:

Constant	Value	Description
PB_IST_Anamorphic	0	Scale the icon to fill the rectangle exactly
PB_IST_Proportional	1	Default scaling type. Scale the icon to fit the width or height of the rectangle while maintaining the icon's original aspect ratio

Spacing[2]

An array of two numbers between 0.0 and 1.0 indicating the fraction of leftover space to allocate at the left and bottom of the icon. A value of [0.0][0.0] positions the icon at the bottom-left corner of the annotation rectangle. A value of [0.5][0.5] centers it within the rectangle. This entry is used only if the icon is scaled proportionally. Default value: [0.5][0.5].

bFit

If true, indicates that the button appearance should be scaled to fit fully within the bounds of the annotation without taking into consideration the line width of the border.

2.6.1.12 PXC_PushButton_Options

PXC_PushButton_Options

Top Previous Next

The **PXC_PushButton_Options** structure specifies push button options.

```
typedef struct _PXC_PushButton_Options {
   PXC_AnnotsFlags Flags;
   BOOL bReadOnly;
   BOOL bRequired;
   PXC_PB_Layout Layout;
   PXC_FF_HighlightMode HMode;
} PXC_PushButton_Options;
```

Members

Flags

Specifies a flag which can hold one of the following values:

Constant	Value	Description
AF_Invisible	0	If set, do not display the annotation if it does not belong to one of the standard annotation types and no annotation handler is available. If clear, display such an unknown annotation using an

		appearance stream specified by its appearance dictionary, if any
`AF_Hidden`	1	(PDF 1.2) If set, do not display or print the annotation or allow it to interact with the user, regardless of its annotation type or whether an annotation handler is available. In cases where screen space is limited, the ability to hide and show annotations selectively can be used in combination with appearance streams to display auxiliary pop-up information similar in function to online help systems
`AF_Print`	2	(PDF 1.2) If set, print the annotation when the page is printed. If clear, never print the annotation, regardless of whether it is displayed on the screen. This can be useful, for example, for annotations representing interactive pushbuttons, which would serve no meaningful purpose on the printed page
`AF_NoZoom`	3	(PDF 1.3) If set, do not scale the annotation's appearance to match the magnification of the page. The location of the annotation on the page (defined by the upper-left corner of its annotation rectangle) remains fixed, regardless of the page magnification. See below for further discussion
`AF_NoRotate`	4	(PDF 1.3) If set, do not rotate the annotation's appearance to match the rotation of the page. The upper-left corner of the annotation rectangle remains in a fixed location on the page, regardless of the page rotation
`AF_NoView`	5	(PDF 1.3) If set, do not display the annotation on the screen or allow it to interact with the user. The annotation may be printed (depending on the setting of the Print flag) but should be considered hidden for purposes of on-screen display and user interaction
`AF_Locked`	7	(PDF 1.4) If set, do not allow the annotation to be deleted or its properties (including position and size) to be modified by the user. However, this flag does not restrict changes to the annotation's contents, such as the value of a form field
`AF_ToggleNoView`	8	(PDF 1.5) If set, invert the interpretation of the NoView flag for certain events. A typical use is to have an annotation that appears only when a mouse cursor is held over it

bReadOnly

Specifies whether button will be ReadOnly

bRequired

Specifies whether button will be Required

Layout

Specifies the positioning of icon and text, can be the following values:

Constant	Value	Description
`PB_Layout_CaptionOnly`	0	No icon. Caption only
`PB_Layout_IconOnly`	1	No caption. Icon only
`PB_Layout_CaptionBelowIcon`	2	Caption below the icon
`PB_Layout_CaptionAboveIcon`	3	Caption above the icon

PB_Layout_CaptionRightIcon	4	Caption to the right of the icon
PB_Layout_CaptionLeftIcon	5	Caption to the left of the icon
PB_Layout_CaptionOverIcon	6	Caption overlaid directly on the icon

HMode

Specifies Highlighting mode. Possible values are:

Constant	Value	Meaning
PB_HM_None	0	No highlighting
PB_HM_Invert	1	Invert the contents of the annotation rectangle
PB_HM_Outline	2	Invert the annotation's border
PB_HM_Push	3	Display the annotation's down appearance, if any

2.6.1.13 Rectangles

Rectangles

Top Previous Next

PDF-XChange Library uses *rectangles* to specify rectangular areas on pages.

The structure PXC_RectF is used to define a rectangle. This structure specifies the coordinates of two points: the upper left and lower right corners of the rectangle. The sides of the rectangle extend from these two points and are parallel to the *x*- and *y*-axes.

The coordinate values for a rectangle are expressed as signed doubles. The coordinate value of a rectangle's right side must be greater than that of its left side. And the coordinate value of the top must be greater than that of the bottom, This is due to the fact that the *y* axis extends upwards.

The following structures are used with rectangles:
- PXC_PointF
- PXC_RectF

2.6.1.13.1 PXC_PointF

PXC_PointF

Top Previous Next

The **PXC_PointF** structure defines the x- and y- coordinates of a point.

```
typedef struct _PXC_PointF {
   double x;
   double y;
} PXC_PointF;
```

Members

x

Specifies the x-coordinate of the point.

y

Specifies the y-coordinate of the point.

2.6.1.13.2 PXC_RectF

PXC_RectF

Top Previous Next

The **PXC_RectF** structure defines the coordinates of the upper-left and lower-right corners of a rectangle.

```
typedef struct _PXC_RectF {
   double left;
   double top;
   double right;
   double bottom;
} PXC_RectF;
```

Members

left
Specifies the x-coordinate of the upper-left corner of the rectangle.

top
Specifies the y-coordinate of the upper-left corner of the rectangle.

right
Specifies the x-coordinate of the lower-right corner of the rectangle.

bottom
Specifies the y-coordinate of the lower-right corner of the rectangle.

Remarks

As the *y* axis of the *coordinate system* extends vertically upward, then a rectangle will be *normalized* if its *top* is greater than *bottom* and *right* is greater than *left*.

E.g. Defining a rectangle with the help of the point (x, y) and the height h and width w:

```
PXC_RectF rect;
rect.left = x;
rect.right = rect.left + w;
rect.bottom = y;
rect.top = rect.bottom + h;
```

2.6.1.14 PXC_TB_Options

PXC_TB_Options

Top Previous Next

The **PXC_TB_Options** structure specifies checkbox options.

```
typedef struct _PXC_TB_Options {
   DWORD tbStyle;
   BOOL bDoNotSpellCheck;
   BOOL bDoNotScroll;
   DWORD dwMaxLen;
} PXC_TB_Options;
```

Members

tbStyle

specifies the style of the textbox. May be bitwise OR a combination of the following styles:

Constant	Value	Description
TB_Multiline	0x2	If set, the field can contain multiple lines of text, if clear, the field's text is restricted to a single line
TB_Password	0x4	If set, the field is intended for entering a secure password that should not be echoed visibly to the screen. Characters typed from the keyboard should instead be echoed in some unreadable form, such as asterisks or bullet characters
TB_Comb	0x8	(PDF 1.5) Meaningful only if the MaxLen entry is present in the text field dictionary and if the Multiline, Password, and FileSelect flags are clear. If set, the field is automatically divided into as many equally spaced positions, or combs, as the value of MaxLen, and the text is laid out into those combs
TB_FileSelect	0x10	(PDF 1.4) If set, the text entered in the field represents the pathname of a file whose contents are to be submitted as the value of the field

bDoNotSpellCheck

(PDF 1.4) If TRUE, text entered in the field is not spell-checked.

bDoNotScroll

(PDF 1.4) If TRUE, the field does not scroll (horizontally for single-line fields, vertically for multiple-line fields) to accommodate more text than fits within its annotation rectangle. Once the field is full, no further text is accepted.

dwMaxLen

Specifies the maximum length of text content.

2.6.1.15 PXC_TextOptions

PXC_TextOptions

Top Previous Next

The **PXC_TextOptions** structure describes the current parameters to be applied to text drawing.

```
typedef struct _PXC_TextOptions {
   DWORD cbSize;
   DWORD fontID;
   double fontSize;
   PXC_TextPosition nTextPosition;
   PXC_TextAlign nTextAlign;
   double LineSpacing;
   double PapaSpacing;
   double SimItalicAngle;
   double SimBoldThickness;
} PXC_TextOptions;
```

Members

cbSize

Specifies the size of this structure.

fontID

Specifies the font identifier used for text output.

fontSize

Specifies the font size in points.

nTextPosition

Specifies vertical text position. May have one of the following values:

Constant	Value	Meaning
`TextPosition_Top`	0	The reference point will be on the top edge of the bounding rectangle.
`TextPosition_Baseline`	1	The reference point will be on the base line of the text.
`TextPosition_Bottom`	2	The reference point will be on the bottom edge of the bounding rectangle.

nTextAlign

Specifies text alignment. May be a combination of the following flags:

Constant	Value	Meaning
`TextAlign_Left`	0x0000	Align text to the left.
`TextAlign_Center`	0x0001	Centers text horizontally in the rectangle.
`TextAlign_Right`	0x0002	Align text to the left.
`TextAlign_Justify`	0x0003	Justify the text within the bounding rectangle, so the right 'edges' of the text lines (All excluding the final one) are on the right edge of the boundary rectangle
`TextAlign_FullJustify`	0x0007	The same as `TextAlign_Justify` except the final line that is fully justified too.
`TextAlign_Top`	0x0000	Align text to the top.
`TextAlign_VCenter`	0x0010	Centers text vertically within the rectangle.
`TextAlign_Bottom`	0x0020	Align text to the bottom.

LineSpacing

Specifies line spacing values in points (*reserved. currently not used.*).

PapaSpacing

Specifies spacing values in points between paragraphs (*reserved. currently not used.*).

SimItalicAngle

Specifies the angle to simulate italic for fonts which have no natural italic variant (in degrees).

SimBoldThickness

Specifies Bold text 'thickness' (in points). Used for *simulating* bold for fonts which have no natural bold variant.

2.6.1.16 PXC_TRIVERTEX

PXC_TRIVERTEX

Top Previous Next

The **PXC_TRIVERTEX** structure is used to pass location and color information for a point as part of a **PXC_GradientFill** operation.

```
typedef struct _PXC_TRIVERTEX {

   double x;

   double y;

   COLORREF color;

} PXC_TRIVERTEX;
```

Members

x

Specifies the x-coordinate, in points, of the upper-left corner of the rectangle.

y

Specifies the y-coordinate, in points, of the upper-left corner of the rectangle.

color

Indicates color information at the point (x, y).

2.6.1.17 PXC_Watermark (COPY)

PXC_Watermark

Top Previous Next

The **PXC_Watermark** structure describes the available watermark attributes and capabilities.

```
typedef struct _PXC_Watermark {
   DWORD m_Size;
   PXC_WaterType m_Type;
// Text part
   WCHAR m_FontName[64];
   DWORD m_FontWeight;
   BOOL m_bItalic;
   double m_FontSize;
   PXC_TextRenderingMode m_Mode;
   double m_LineWidth;
   COLORREF m_FColor;
   COLORREF m_SColor;
   WCHAR m_Text[256];
// Image Part
   WCHAR m_FileName[MAX_PATH];
   COLORREF m_TransColor;
   double m_Width;
   double m_Height;
   BOOL m_bKeepAspect;
// Commmon Part
   DWORD m_Align;
   double m_XOffset;
   double m_YOffset;
   double m_Angle;
   DWORD m_Opacity;
// Place Info
   PXC_WaterPlaceOrder m_PlaceOrder;
   PXC_WaterPlaceType m_PlaceType;
// Ranges
   DWORD m_NumRanges;
   LPDWORD m_Range;
// Available when library version is >= 3.30.0065
   DWORD m_ImagePageNumber;
} PXC_Watermark;
```

Members

m_Size

Specifies the size of the structure and is provided for compatibility with future versions of PDF-XChange where this structure may be modified.

m_Type

Specifies the type of watermark. May have any one of the following values:

Constant	**Meaning**
WaterType_Text	Text watermark. All structural Image content will be ignored.
WaterType_Image	Image watermark. All structural text content will be ignored.

m_FontName

Specifies the font name for the text watermark. The font name must be a null terminated string and may not exceed 64 chars (including any terminating null-symbol). Please note that font name used must be defined as UNICODE characters.

m_FontWeight

Specifies the font weigh for the text watermark. For more info see the **PXC_AddFontA** function.

m_bItalic

If this parameter is `TRUE` the italic variant of the font will be used.

m_FontSize

Specifies the font size for a text watermark in points. If this value is 0, then the text watermark will be 'fitted' to the page.

m_Mode

Specifies the text drawing mode. For more information about text drawing modes, see the **PXC_SetTextRMode** function.

Note: Only modes from `TextRenderingMode_Fill` to `TextRenderingMode_None` are supported.

m_LineWidth

Specifies the border width for text drawing modes `TextRenderingMode_Stroke` and

`TextRenderingMode_FillStroke`. For other modes this value is ignored. Line width is specified in points.

m_FColor

Specifies the color of the text (only for `TextRenderingMode_Fill` and `TextRenderingMode_FillStroke` drawing modes).

m_SColor

Specifies the color of the text border (only for `TextRenderingMode_Stroke` and `TextRenderingMode_FillStroke` drawing modes).

m_Text

A null-terminated string that specifies the text of the watermark.

m_FileName

A null-terminated string that specifies the image file name for use as a watermark.

m_TransColor

Specifies the transparent color for the image. If the high-order byte value of this member is not 0, then this member is ignored and a transparent color will not be used.

m_Width

Specifies the width of the image when placed on the page. This value is specified in points. If this member is 0, then the image will be resized to 'fit' to the page.

m_Height

Specifies height of the image when placed on the page. This value is specified in points. If this member is 0, then the image will be resized to 'fit' to the page.

m_bKeepAspect

Specifies how the **PDF-XChange Library** will scale the image during placement on the page. If this member is not 0 (zero), then the image will be resized to 'fit' in the rectangle with `m_Width` width and `m_Height` height and keeping its aspect ratio. Otherwise the aspect ratio will not be preserved.

m_Align

Specifies the horizontal and vertical alignment of the watermark text. The value is a combination of horizontal:

Value	**Meaning**
`TextAlign_Left`	Aligns text or image to the left.
`TextAlign_Center`	Centers text or image horizontally on the page.
`TextAlign_Right`	Aligns text or image to the right.

and vertical alignments:

Value	**Meaning**
`TextAlign_Top`	Aligns text or image to the top of the page.
`TextAlign_VCenter`	Centers text or image vertically.
`TextAlign_Bott`	Aligns text or image to the bottom of the page.

om

m_XOffset

Specifies the offset of the watermark by `X` from the normal position.

m_YOffset

Specifies the offset of the watermark by `Y` from the normal position.

m_Angle

Specifies the rotation angle for the watermark text or image. *m_Angle* is the rotation angle in degrees, and its value must be in the range -90.0 to 90.0.

m_Opacity

Specifies the opacity level of the watermark (PDF specification must be 1.4 or higher to use transparency). Must be in range from 0 to 255.

m_PlaceOrder

Specifies how the watermark will be placed on the page(s) - In the foreground or background. May have any one of the following values:

Constant	Meaning
PlaceOrder_Background	Watermark will be placed in the background.
PlaceOrder_Foreground	Watermark will be placed in the foreground.

m_PlaceType

Specifies on which page(s) the watermark will be placed. May be a valid page number or one of the following constants:

Value	Meaning
PlaceType_AllPages	Watermark will be placed on all pages.
PlaceType_FirstPage	Watermark will be placed only on the first page of the document.
PlaceType_LastPage	Watermark will be placed only on the last page of the document.
PlaceType_EvenPages	Vertical reflection component
PlaceType_OddPages	Watermark will be placed only on odd pages of the document.
PlaceType_Range	Watermark will be placed only on pages specified by *m_Range* field.

m_NumRanges

Specifies the number of pairs in the array pointed to by *m_Range*.
This field must be 0 when *m_PlaceType* is not equal `PlaceType_Range`.

m_Range

Pointer to an array of paired `DWORD`'s values. The first element of a such pair, specifies the starting page number; The second specifies an ending page number where the watermark may be placed. This field must be `NULL` when *m_PlaceType* is not equal to `PlaceType_Range`.

m_ImagePageNumber

Specifies the page number (zero based) from image, which will be used as watermark. If this value will point to the page which is absent into the image, last available page from the image will be used.

2.6.2 PXC_AddGotoAction

PXC_AddGotoAction

Top Previous Next

PXC_AddGotoAction specifies the area (rectangle) on the page, which should activate the showing of the specified page in the specified mode upon selection.

```
HRESULT  PXC_AddGotoAction(
    _PXCContent* content,
    LPCPXC_RectF rect,
    DWORD page,
    PXC_OutlineDestination mode,
    double v1,
    double v2,
    double v3,
    double v4,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

content

[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

page

[in] Specifies the page number (Zero-based), to be displayed when the user activates this annotation.

mode

[in] Specifies the mode in which the destination page, as specified by *page*, will be displayed, when the user views this item. For more information about possible values and their meanings see function **PXC_AddOutlineEntryW**.

v1

[in] Has the meaning, similar to a corresponding parameter of the function **PXC_AddOutlineEntryW**
.

v2

[in] Has the meaning, similar to a corresponding parameter of the function **PXC_AddOutlineEntryW**
.

v3

[in] Has the meaning, similar to a corresponding parameter of the function **PXC_AddOutlineEntryW**

.

v4

[in] Has the meaning, similar to a corresponding parameter of the function **PXC_AddOutlineEntryW**
.

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXC_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
    // Example shows how to add a GoTo annotation to the document.

    void AddGoToAnnotation(_PXCContent* pContent,  DWORD GoToPageNumber)
    {
        HRESULT hr = DS_OK;

        // Annotation rectangle

        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Common annotation information

        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Add GoTo annotation onto the first page

        hr = PXC_AddGotoAction(pContent, &rect, GoToPageNumber, Dest_Y, 100,
100, 100, 100, &AnnotInfo);
```

```
        if (IS_DS_FAILED(hr))
        {
            // report the error
            ...
        }

        // done.
    }
```

2.6.3 PXC_AddLaunchActionA

PXC_AddLaunchActionA

Top Previous Next

Function **PXC_AddLaunchActionA** specifies the rectangular area on the page that allows execution of a specified application, or the opening or printing of a specified document upon activation.

```
HRESULT  PXC_AddLaunchActionA(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCSTR lpszFileName,
    LPCSTR lpszParams,
    PXC_LaunchOperation oper,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect
: [in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

lpszFileName
: [in] Pointer to a null-terminated string that specifies the full path and file name of the application to be launched or the document to opened or printed.

lpszParams
: [in] Pointer to a null-terminated string that specifies a parameter string to be passed to the application as designated by the *lpszFileName* parameter. This parameter may be `NULL`.

oper
: [in] *oper* specifies the operation to perform. May be any one of the following values:

Value	Description
LO_Open	Open a document.
LO_Print	Print a document.

pInfo
: [in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, The document's global settings will be used (for more information see **PXC_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function has a UNICODE equivalent **PXC_AddLaunchActionW**.

Example (C++).

```
// Example shows, how to add Launch annotation to the document

    void AddLaunchAnnotation(_PXCContent* pContent,  LPCSTR PDFFileToOpen)
    {
        HRESULT hr = DS_OK;

        // Annotation rectangle

        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Common annotation information

        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Add lunch annotation to the specified content

        hr = PXC_AddLaunchActionA(pContent, &rect, PDFFileToOpen, NULL,
LO_Open, &AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report the error
            ...
        }

        // done.
    }
```

2.6.4 PXC_AddLaunchActionW

PXC_AddLaunchActionW

Top Previous Next

PXC_AddLaunchActionW specifies the rectangular area on the page that allows execution of a specified application, or the opening or printing of a specified document upon activation

This function is the UNICODE equivalent to the **PXC_AddLaunchActionA** function.

```
HRESULT  PXC_AddLaunchActionW(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCWSTR lpwszFileName,
    LPCWSTR lpwszParams,
    PXC_LaunchOperation oper,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

content
[in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect
[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

lpwszFileName
[in] Pointer to a null-terminated string that specifies the full path and file name of the application to be launched or the document to opened or printed.

lpwszParams
[in] Pointer to a null-terminated string that specifies a parameter string to be passed to the application as designated by the *lpwszFileName* parameter. This parameter may be `NULL`.

oper
[in] *oper* specifies the operation to perform. May be any one of the following values:

Value	Description
LO_Open	Open a document.
LO_Print	Print a document.

pInfo
[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXC_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows, how to add Launch annotation to the document

    void AddLaunchAnnotation(_PXCContent* pContent,  LPCWSTR PDFFileToOpen)
    {
        HRESULT hr = DS_OK;

        // Annotation rectangle

        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Common annotation information

        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Add lunch annotation to the specified content

        hr = PXC_AddLaunchActionW(pContent, &rect, PDFFileToOpen, NULL,
LO_Open, &AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report the error
            ...
        }

        // done.
    }
```

2.6.5 PXC_AddLineAnnotation

PXC_AddLineAnnotation

Top Previous Next

PXC_AddLineAnnotation adds *a line annotation* to the specified content. This displays a single straight line on the page.

```
HRESULT  PXC_AddLineAnnotation(
     _PXCContent* content,
     LPCPXC_PointF pntStart,
     LPCPXC_PointF pntEnd,
     PXC_LineAnnotsType sEndStyle,
     PXC_LineAnnotsType eEndStyle,
     COLORREF cInterior,
     const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

pntStart
: [in] Pointer to a `PXC_PointF` structure that specifies the starting coordinates of the line.

pntEnd
: [in] Pointer to a `PXC_PointF` structure that specifies the ending coordinates of the line.

sEndStyle
: [in] *sEndStyle* specifies the line ending style for the starting point of the line.

eEndStyle
: [in] *eEndStyle* specifies the line ending style for the ending point of the line.

cInterior
: [in] Specifies interior color for line endings. See **Comments**.

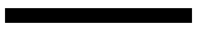
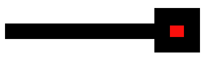
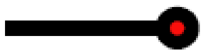
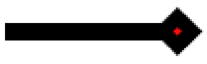
pInfo
: [in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXC_SetAnnotsInfo**).

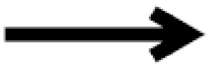
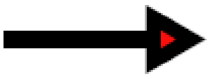
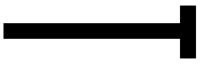
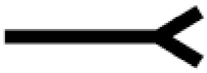
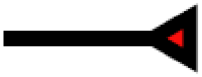
Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

Line ending styles table (`PXC_LineAnnotsType`):

Constant	Appearance	Description
LAType_None		No line ending.
LAType_Square		A square filled with the annotation's interior color.
LAType_Circle		A circle filled with the annotation's interior color.
LAType_Diamond		A diamond shape filled with the annotation's interior color.

`LAType_OpenArrow`		Two short lines meeting in an acute angle, forming an open arrowhead.
`LAType_ClosedArrow`		Two short lines meeting in an acute angle as in the `LAType_OpenArrow` style, connected by a third line to form a triangular closed arrowhead filled with the annotation's interior color.
`LAType_Butt`		(*PDF 1.5*) A short line at the endpoint perpendicular to the line itself.
`LAType_ROpenArrow`		(*PDF 1.5*) Two short lines in the reverse direction from `LAType_OpenArrow`.
`LAType_RClosedArrow`		(*PDF 1.5*) A triangular closed arrowhead in the reverse direction from `LAType_ClosedArrow`.

Example (C++).

```
// Example shows, how to add line annotation to the document

    void AddLineAnnotation(_PXCContent* pContent)
    {
        HRESULT hr = DS_OK;

        // Common annotation information

        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Start and end points

        PXC_PointF        startPnt;
        PXC_PointF        endPnt;

        // Points data

        startPnt.x = 300.0;
        startPnt.y = 300.0;
        endPnt.x = 10.0;
        endPnt.y = 10.0;

        // Annotation rectangle

        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Add annotation

        hr = PXC_AddLineAnnotation(pContent, &rect, &startPnt, &endPnt,
LAType_Square, LAType_Circle, RGB(0, 200, 150), &AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // done.
    }
```

2.6.6 PXC_AddLink

PXC_AddLink

Top Previous Next

PXC_AddLink adds a URL link to the current content.

```
HRESULT  PXC_AddLink(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCSTR lpszURL,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

content
    [in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect
    [in] *rect* specifies the bounding rectangle of the link.

lpszURL
    [in] *lpszURL* specifies the URL of the link. This parameter must be a null-terminated string.

pInfo
    [in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXC_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.

To determine if the function was successful use the defined macro's as described here: Error Handling.

Example (C++).

```
// Example shows how to add link annotation to the document

    void AddLinkAnnotation(_PXCContent* pContent)
    {
        HRESULT hr = DS_OK;

        // Specify the rect to the annotation

        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Common annotation information

        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Add annotation to the specified content

        hr = PXC_AddLink(pContent, &rect, "http://www.google.com",
&AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // done.
    }
```

2.6.7 PXC_AddOutlineEntryA

PXC_AddOutlineEntryA

Top Previous Next

PXC_AddOutlineEntryA adds a new outline *(also known as BookMarks)* entry to the document outline tree, this also allows a page to be viewed by the user in a pre-determined manner (i.e. fitted to the viewer window size) - if an outline entry is selected.

This function is the ASCII equivalent to the **PXC_AddOutlineEntryW** function.

```
void*  PXC_AddOutlineEntryA(
    _PXCDocument* pdf,
    void* parent,
    void* after,
    BOOL open,
    DWORD page,
    LPCSTR title,
    PXC_OutlineDestination mode,
    double v1,
    double v2,
    double v3,
    double v4,
    COLORREF color,
    DWORD flags
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

parent

[in] Specifies the identifier of the outline entry, that is the parent for the newly added item. If an item is inserted at the top level of the outline hierarchy, the *parent* must have an `OutlineItem_Root` value, otherwise it's value must be a valid outline entry identifier as returned by the previous call of **PXC_AddOutlineEntryA**.

after

[in] Specifies the identifier of the outline entry, after which the new entry will be added. Must be a valid outline entry identifier as returned by the previous call of the **PXC_AddOutlineEntryA** function, or one of the following values:

Value	Definition
OutlineItem_First	Inserts the item at the beginning of the list.
OutlineItem_Last	Inserts the item at the end of the list.

open

[in] If TRUE, the element will be initially opened when the file is viewed , otherwise it will be closed.

page

[in] Specifies the page number (zero-based), that should be displayed when the user views this item.

title

[in] Pointer to a null-terminated string that specifies title of the entry that will be displayed, in the tree.

mode

[in] Specifies the mode in which the destination page, as specified by *page*, will be displayed, when the user views this item.
Acceptable values:

Value	**Definition**
`Dest_Page`	Retain current display location and zoom. Parameters *v1*, *v2*, *v3*, and *v4* are not used.
`Dest_XYZ`	Display the page designated by *page*, with the coordinates `(v1, v2)` positioned at the top-left corner of the window and the contents of the page magnified by the factor *v3*. Parameters *v1* and *v2* are specified in points, and *v3* is specified in percentage points. Parameter *v4* is not used.
`Dest_Fit`	Display the page designated by *page*, with its contents magnified to fit the entire page within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are incompatible, use the smaller of the two, centering the page within the window. Parameters *v1*, *v2*, *v3*, and *v4* are not used.
`Dest_FitH`	Display the page designated by *page*, with the vertical coordinate *v1* (top), specified in points, positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of the page within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_FitV`	Display the page designated by *page*, with the horizontal coordinate *v1* (left), specified in points, positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire height of the page within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_FitR`	Display the page designated by *page*, with its contents magnified just enough to fit the rectangle specified by the coordinates *v1* (left), *v2* (top), *v3* (right), and *v4* (bottom) entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the rectangle within the window. Parameters *v1*, *v2*, *v3*, and *v4* are specified in points.
`Dest_FitB`	Display the page designated by *page*, with its contents magnified just enough to fit its bounding box entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are incompatible, use the smaller of the two, centering the bounding box within the window.

	Parameters *v1*, *v2*, *v3*, and *v4* are not used.
`Dest_FitBH`	Display the page designated by *page*, with the vertical coordinate's top positioned at the *v1* (top), specified in points, to the edge of the window and the contents of the page magnified just enough to fit the entire width of its bounding box within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_FitBV`	Display the page designated by *page*, with the horizontal coordinate left positioned at the *v1* (left), specified in points, to the edge of the window and the contents of the page magnified just enough to fit the entire height of its bounding box within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_Y`	Same as DST_XYZ, but specifies only `Y` coordinate (*v1*, in points), leave others parameters unchanged. Parameters *v2*, *v3*, and *v4* are not used.

v1

[in] Meaning of this parameter dependant on the parameter *mode*.

v2

[in] Meaning of this parameter dependant on the parameter *mode*.

v3

[in] Meaning of this parameter dependant on the parameter *mode*.

v4

[in] Meaning of this parameter dependant on the parameter *mode*.

color

[in] Specifies the color of the entry as it will be displayed in the outline tree. See **Comments** section.

flags

[in] Specifies additional flags for the entry. With these flags you can set the font type for which this item will be displayed. Can be 0 (`OutlineStyle_Normal`), or a combination of the following flags:

Flag	Value	Meaning
`OutlineStyle_Italic`	0x0001	If set, item will be displayed in *italic*.
`OutlineStyle_Bold`	0x0002	If set, item will be displayed in **bold**.

Return Values

If the function succeeds, the return value represents the valid identifier of the outline entry.
If the function fails, the return value is `NULL`.

Comments

Parameters *color* and *flags* are valid only if you have called and made use of the function **PXC_SetSpecVersion** and set your PDF format specification to at least `SpecVersion14` or a higher value.

Example (C++).

```
// Example shows how to do add different kind of outlines
    // to the document with the different destinations
    // Define for page width and height that corresponds to 'A4' page format

    #define PH                        I2L(11)
    #define PW                        I2L(8.5)

    // Define color constants

    #define black                RGB(0, 0, 0)
    #define white                RGB(255, 255, 255)
    #define gold                RGB(255, 215, 0)
    #define khaki                RGB(189, 183, 107)
    #define lkhaki                RGB(240, 230, 140)
    #define gray                RGB(128, 128, 128)

    HRESULT Outlines(_PXCDocument* pdf)
    {
        _PXCPage* page;
        _PXCContent* cpage;

        // Add new page

        HRESULT res = PXC_AddPage(pdf, PW, PH, &page);
        if (IS_DS_FAILED(res))
            return res;

        cpage = (_PXCContent*)page;

        // Calculate intermediate parameters for drawings

        double x = I2L(1);
        double y = PH - I2L(1);
        double w = PW - I2L(2);
        double h = PH / 2.0 - I2L(2);
        double w2 = (w - I2L(2)) / 3.0;
        double h2 = h - I2L(1);
        double x2 = x + I2L(0.5);
        double y2 = y - I2L(0.5);

        PXC_RectF rect, rect2;

        rect.left = x;
        rect.top = y;
        rect.right = x + w;
        rect.bottom = rect.top - h;

        rect2.left = x2;
        rect2.top = y2;
        rect2.right = rect2.left + w2;
        rect2.bottom = rect2.top - (h - I2L(1));
```

```
        // Add root item that will point to the entire page

        void* root = PXC_AddOutlineEntryA(pdf, OutlineItem_Root,
OutlineItem_First, TRUE, 0, "Root Element",
            Dest_Page, 0, 0, 0, 0, black, OutlineStyle_Normal);

        // Draw rectangle

        PXC_Rect(cpage, x, y, x + w, y - h);
        PXC_StrokePath(cpage, FALSE);

        // Add outline item that will point to that rectangle

        void* it1 = PXC_AddOutlineEntryA(pdf, root, OutlineItem_First, TRUE,
0, "Std. Element",
            Dest_FitR, x, y, x + w, y - h, black, OutlineStyle_Normal);

        // Draw one more rectangle

        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_StrokePath(cpage, FALSE);

        // Add outline item that will point to that rectangle

        PXC_AddOutlineEntryA(pdf, it1, OutlineItem_Last, TRUE, 0,
            "Bold", Dest_FitR, x2, y2, x2 + w2, y2 - h2, black,
OutlineStyle_Bold);


        // Do some other drawing and adding outlines
        // that will point to that drawings

        x2 += w2 + I2L(0.5);
        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_StrokePath(cpage, FALSE);

        PXC_AddOutlineEntryA(pdf, it1, OutlineItem_Last, TRUE, 0, "Italic",
            Dest_FitR, x2, y2, x2 + w2, y2 - h2, black, OutlineStyle_Italic);

        x2 += w2 + I2L(0.5);
        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_StrokePath(cpage, FALSE);

        PXC_AddOutlineEntryA(pdf, it1, OutlineItem_Last, TRUE, 0, "Bold-
Italic",
            Dest_FitR, x2, y2, x2 + w2, y2 - h2, black,
OutlineStyle_BoldItalic);

        y -= h + I2L(1);
        x2 = x + I2L(0.5);
        y2 = y - I2L(0.5);

        PXC_Rect(cpage, x, y, x + w, y - h);
        PXC_StrokePath(cpage, FALSE);

        it1 = PXC_AddOutlineEntryA(pdf, root, it1, TRUE, 0, "Colored Element",
            Dest_FitR, x, y, x + w, y - h, khaki, OutlineStyle_Normal);

        PXC_SetFillColor(cpage, khaki);
        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_FillPath(cpage, FALSE, TRUE, FillRule_Winding);

        PXC_AddOutlineEntryA(pdf, it1, OutlineItem_Last, TRUE, 0, "Bold",
            Dest_FitR, x2, y2, x2 + w2, y2 - h2, khaki, OutlineStyle_Bold);

        x2 += w2 + I2L(0.5);
        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_FillPath(cpage, FALSE, TRUE, FillRule_Winding);

        PXC_AddOutlineEntryA(pdf, it1, OutlineItem_Last, TRUE, 0, "Italic",
            Dest_FitR, x2, y2, x2 + w2, y2 - h2, khaki, OutlineStyle_Italic);

        x2 += w2 + I2L(0.5);
        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_FillPath(cpage, FALSE, TRUE, FillRule_Winding);

        PXC_AddOutlineEntryA(pdf, it1, OutlineItem_Last, TRUE, 0, "Bold-
Italic",
            Dest_FitR, x2, y2, x2 + w2, y2 - h2, khaki,
OutlineStyle_BoldItalic);

        return res;
    }
```

2.6.8 PXC_AddOutlineEntryW

PXC_AddOutlineEntryW

Top Previous Next

PXC_AddOutlineEntryW adds a new outline *(also known as BookMarks)* entry to the document outline tree, this also allows a page to be viewed by the user in a pre-determined manner (i.e. fitted to the viewer window size) - if an outline entry is selected.

This function is the UNICODE equivalent to the **PXC_AddOutlineEntryA** function.

```
void*  PXC_AddOutlineEntryW(
    _PXCDocument* pdf,
    void* parent,
    void* after,
    BOOL open,
    DWORD page,
    LPCWSTR title,
    PXC_OutlineDestination mode,
    double v1,
    double v2,
    double v3,
    double v4,
    COLORREF color,
    DWORD flags
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

parent

[in] Specifies the identifier of the outline entry, that is the parent for the newly added item. If an item is inserted at the top level of the outline hierarchy, the *parent* must have an `OutlineItem_Root` value, otherwise it's value must be a valid outline entry identifier as returned by the previous call of **PXC_AddOutlineEntryW**.

after

[in] Specifies the identifier of the outline entry, after which the new entry will be added. Must be a valid outline entry identifier as returned by the previous call of the **PXC_AddOutlineEntryW** function, or one of the following values:

Value	Meaning
OutlineItem_First	Inserts the item at the beginning of the list.
OutlineItem_Last	Inserts the item at the end of the list.

open

[in] If TRUE, the element will be initially opened when the file is viewed , otherwise it will be closed.

page

[in] Specifies the page number (zero-based), that should be displayed when the user views this item.

title

[in] Pointer to a null-terminated string that specifies title of the entry that will be displayed, in the tree.

mode

[in] Specifies the mode in which the destination page, as specified by *page*, will be displayed, when the user views this item.
Acceptable values:

Value	Meaning
Dest_Page	Retain current display location and zoom. Parameters *v1*, *v2*, *v3*, and *v4* are not used.
Dest_XYZ	Display the page designated by *page*, with the coordinates `(v1, v2)` positioned at the top-left corner of the window and the contents of the page magnified by the factor *v3*. Parameters *v1* and *v2* are specified in points, and *v3* is specified in percentage points. Parameter *v4* is not used.

`Dest_Fit`	Display the page designated by *page*, with its contents magnified to fit the entire page within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are incompatible, use the smaller of the two, centering the page within the window. Parameters *v1*, *v2*, *v3*, and *v4* are not used.
`Dest_FitH`	Display the page designated by *page*, with the vertical coordinate *v1* (top), specified in points, positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of the page within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_FitV`	Display the page designated by *page*, with the horizontal coordinate *v1* (left), specified in points, positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire height of the page within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_FitR`	Display the page designated by *page*, with its contents magnified just enough to fit the rectangle specified by the coordinates *v1* (left), *v2* (top), *v3* (right), and *v4* (bottom) entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the rectangle within the window. Parameters *v1*, *v2*, *v3*, and *v4* are specified in points.
`Dest_FitB`	Display the page designated by *page*, with its contents magnified just enough to fit its bounding box entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are incompatible, use the smaller of the two, centering the bounding box within the window. Parameters *v1*, *v2*, *v3*, and *v4* are not used.
`Dest_FitBH`	Display the page designated by *page*, with the vertical coordinate's top positioned at the *v1* (top), specified in points, to the edge of the window and the contents of the page magnified just enough to fit the entire width of its bounding box within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_FitBV`	Display the page designated by *page*, with the horizontal coordinate left positioned at the *v1* (left), specified in points, to the edge of the window and the contents of the page magnified just enough to fit the entire height of its bounding box within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_Y`	Same as DST_XYZ, but specifies only `Y` coordinate (*v1*, in points), leave others parameters unchanged. Parameters *v2*, *v3*, and *v4* are not used.

v1

[in] Meaning of this parameter dependant on the parameter *mode*.

v2

[in] Meaning of this parameter dependant on the parameter *mode*.

v3

[in] Meaning of this parameter dependant on the parameter *mode*.

v4

[in] Meaning of this parameter dependant on the parameter *mode*.

color

[in] Specifies the color of the entry as it will be displayed in the outline tree. See **Comments** section.

flags

[in] Specifies additional flags for the entry. With these flags you can set the font type for which this item will be displayed. Can be 0 (`OutlineStyle_Normal`), or a combination of the following flags:

Flag	Value	Meaning
`OutlineStyle_Italic`	0x0001	If set, item will be displayed in *italic*.
`OutlineStyle_Bold`	0x0002	If set, item will be displayed in **bold**.

Return Values

If the function succeeds, the return value represents the valid identifier of the outline entry.
If the function fails, the return value is `NULL`.

Comments

Parameters *color* and *flags* are valid only if you have called and made use of the function **PXC_SetSpecVersion** and set your PDF format specification to at least `SpecVersion14` or a higher value.

Example (C++).

```
// Example shows how to do add different kind of outlines
    // to the document with the different destinations

    // Defines for page width and height that corresponds to 'A4' page format

    #define PH                                  I2L(11)
    #define PW                                  I2L(8.5)

    // Define color constants

    #define black                    RGB(0, 0, 0)
    #define white                    RGB(255, 255, 255)
    #define gold                    RGB(255, 215, 0)
    #define khaki                    RGB(189, 183, 107)
    #define lkhaki                    RGB(240, 230, 140)
    #define gray                    RGB(128, 128, 128)

    HRESULT Outlines(_PXCDocument* pdf)
    {
        _PXCPage* page;
        _PXCContent* cpage;

        // Add new page

        HRESULT res = PXC_AddPage(pdf, PW, PH, &page);
        if (IS_DS_FAILED(res))
```

```
            return res;

        cpage = (_PXCContent*)page;

        // Calculate intermediate parameters for drawings

        double x = I2L(1);
        double y = PH - I2L(1);
        double w = PW - I2L(2);
        double h = PH / 2.0 - I2L(2);
        double w2 = (w - I2L(2)) / 3.0;
        double h2 = h - I2L(1);
        double x2 = x + I2L(0.5);
        double y2 = y - I2L(0.5);

        PXC_RectF rect, rect2;

        rect.left = x;
        rect.top = y;
        rect.right = x + w;
        rect.bottom = rect.top - h;

        rect2.left = x2;
        rect2.top = y2;
        rect2.right = rect2.left + w2;
        rect2.bottom = rect2.top - (h - I2L(1));

        // Add root item that will point to the entire page

        void* root = PXC_AddOutlineEntryW(pdf, OutlineItem_Root,
OutlineItem_First, TRUE, 0, L"Root Element",
            Dest_Page, 0, 0, 0, 0, black, OutlineStyle_Normal);

        // Draw rectangle

        PXC_Rect(cpage, x, y, x + w, y - h);
        PXC_StrokePath(cpage, FALSE);

        // Add outline item that will point to that rectangle

        void* it1 = PXC_AddOutlineEntryW(pdf, root, OutlineItem_First, TRUE,
0, L"Std. Element",
            Dest_FitR, x, y, x + w, y - h, black, OutlineStyle_Normal);

        // Draw one more rectangle

        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_StrokePath(cpage, FALSE);

        // Add outline item that will point to that rectangle
        PXC_AddOutlineEntryW(pdf, it1, OutlineItem_Last, TRUE, 0,
            L"Bold", Dest_FitR, x2, y2, x2 + w2, y2 - h2, black,
OutlineStyle_Bold);


        // Do some other drawing and adding outlines
        // that will point to that drawings

        x2 += w2 + I2L(0.5);
        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_StrokePath(cpage, FALSE);

        PXC_AddOutlineEntryW(pdf, it1, OutlineItem_Last, TRUE, 0, L"Italic",
            Dest_FitR, x2, y2, x2 + w2, y2 - h2, black, OutlineStyle_Italic);

        x2 += w2 + I2L(0.5);
        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_StrokePath(cpage, FALSE);

        PXC_AddOutlineEntryW(pdf, it1, OutlineItem_Last, TRUE, 0, L"Bold-
Italic",
            Dest_FitR, x2, y2, x2 + w2, y2 - h2, black,
OutlineStyle_BoldItalic);

        y -= h + I2L(1);
        x2 = x + I2L(0.5);
        y2 = y - I2L(0.5);

        PXC_Rect(cpage, x, y, x + w, y - h);
        PXC_StrokePath(cpage, FALSE);

        it1 = PXC_AddOutlineEntryW(pdf, root, it1, TRUE, 0, L"Colored
Element",
            Dest_FitR, x, y, x + w, y - h, khaki, OutlineStyle_Normal);

        PXC_SetFillColor(cpage, khaki);
        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_FillPath(cpage, FALSE, TRUE, FillRule_Winding);

        PXC_AddOutlineEntryW(pdf, it1, OutlineItem_Last, TRUE, 0, L"Bold",
            Dest_FitR, x2, y2, x2 + w2, y2 - h2, khaki, OutlineStyle_Bold);

        x2 += w2 + I2L(0.5);
        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_FillPath(cpage, FALSE, TRUE, FillRule_Winding);

        PXC_AddOutlineEntryW(pdf, it1, OutlineItem_Last, TRUE, 0, L"Italic",
            Dest_FitR, x2, y2, x2 + w2, y2 - h2, khaki, OutlineStyle_Italic);
        x2 += w2 + I2L(0.5);
        PXC_Rect(cpage, x2, y2, x2 + w2, y2 - h2);
        PXC_FillPath(cpage, FALSE, TRUE, FillRule_Winding);

        PXC_AddOutlineEntryW(pdf, it1, OutlineItem_Last, TRUE, 0, L"Bold-
Italic",
            Dest_FitR, x2, y2, x2 + w2, y2 - h2, khaki,
OutlineStyle_BoldItalic);

        return res;
    }
```

2.6.9 PXC_AddTextAnnotationA

PXC_AddTextAnnotationA

Top Previous Next

PXC_AddTextAnnotationA adds a *text annotation* object to the content of the PDF.

A text annotation represents a "sticky note" attached to a point in the PDF document. When closed, the annotation appears as an icon; when opened, it displays a pop-up window containing the text of the note, in a font and size chosen by the viewing application.

This function is the ASCII equivalent of the function **PXC_AddTextAnnotationW**.

```
HRESULT  PXC_AddTextAnnotationA(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCSTR pszTitle,
    LPCSTR pszAnnot,
    PXC_TextAnnotsType type,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier of the page content to which the function will be applied.

rect
: [in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

pszTitle
: [in] Pointer to a null-terminated string that specifies the title of the annotation's pop-up window.

pszAnnot
: [in] Pointer to a null-terminated string that specifies text to be displayed for the annotation.

type
: [in] Specifies the icon to be used in displaying the annotation. May be any one of the following values:
  - `TAType_Note`
  - `TAType_Comment`
  - `TAType_Key`

- `TAType_Help`
- `TAType_NewParagraph`
- `TAType_Insert`

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the documents global settings will be used (for more information see **PXC_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Comments

This function has a UNICODE equivalent - **PXC_AddTextAnnotationW**.

Example (C++).

```
// Example shows, how to add text annotation to the document

   void AddTextAnnotation(_PXCContent* pContent, LPCSTR pszTitle, LPCSTR
pszAnnot)
   {
       HRESULT hr = DS_OK;

       // Specify the rect to the annotation

       PXC_RectF rect;
       rect.left = 20.0;
       rect.top = 200.0;
       rect.right = 320.0;
       rect.bottom = 500.0;

       // The type of the annotation

       PXC_TextAnnotsType type = TAType_Note;

       // Common annotation information

       PXC_CommonAnnotInfo AnnotInfo;
       AnnotInfo.m_Color = RGB(200, 0, 100);
       AnnotInfo.m_Flags = 0;
       AnnotInfo.m_Opacity = 1.0;

       // Use dashed border

       AnnotInfo.m_Border.m_DashArray = new double[3];
       AnnotInfo.m_Border.m_DashArray[0] = 1.0;
       AnnotInfo.m_Border.m_DashArray[1] = 2.0;
       AnnotInfo.m_Border.m_DashArray[2] = 0.5;
       AnnotInfo.m_Border.m_DashCount = 3;
```

```
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 1.0;

        // Add annotation to the specified content

        hr = PXC_AddTextAnnotationA(pContent, &rect, pszTitle, pszAnnot, type, 
&AnnotInfo);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // done.
    }
```

2.6.10 PXC_AddTextAnnotationW

PXC_AddTextAnnotationW

Top Previous Next

PXC_AddTextAnnotationW adds a *text annotation* object to the content of the PDF.

A text annotation represents a "sticky note" attached to a point in the PDF document. When closed, the annotation appears as an icon; when opened, it displays a pop-up window containing the text of the note, in a font and size chosen by the viewing application.

This function is the UNICODE equivalent of the function **PXC_AddTextAnnotationA**.

```
HRESULT  PXC_AddTextAnnotationW(
    _PXCContent* content,
    LPCPXC_RectF rect,
    LPCWSTR pwszTitle,
    LPCWSTR pwszAnnot,
    PXC_TextAnnotsType type,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

content
: [in] Parameter *content* specifies the identifier of page content to which the function will be applied.

rect
: [in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

pwszTitle
: [in] Pointer to a null-terminated string that specifies the title of the annotation's pop-up window.

pwszAnnot
: [in] Pointer to a null-terminated string that specifies the text to be displayed for the annotation.

type
: [in] Specifies the icon to be used in displaying the annotation. May be any one of the following values:

- `TAType_Note`
- `TAType_Comment`
- `TAType_Key`
- `TAType_Help`
- `TAType_NewParagraph`
- `TAType_Insert`

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXC_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows, how to add text annotation to the document

    void AddTextAnnotation(_PXCContent* pContent, LPCWSTR pwszTitle, LPCWSTR
pwszAnnot)
    {
        HRESULT hr = DS_OK;

        // Specify the rect to the annotation

        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // The type of the annotation

        PXC_TextAnnotsType type = TAType_Note;

        // Common annotation information

        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;

        // Use dashed border

        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 1.0;
        AnnotInfo.m_Border.m_DashArray[1] = 2.0;
        AnnotInfo.m_Border.m_DashArray[2] = 0.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 1.0;

        // Add annotation to the specified content

        hr = PXC_AddTextAnnotationW(pContent, &rect, pwszTitle, pwszAnnot,
type, &AnnotInfo);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // done.
    }
```

2.6.11 PXC_AddWatermark

PXC_AddWatermark

Top Previous Next

PXC_AddWatermark adds a watermark to the PDF object.

```
HRESULT  PXC_AddWatermark(
    _PXCDocument* pdf,
    PXC_Watermark* watermark
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXC_NewDocument**.

watermark

[in] Pointer to the **PXC_Watermark** structure that describes the watermark to be added.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows how to add text watermark to the document

    _PXCDocument* pdf;

    ...

    PXC_Watermark wt;
```

```
// Fill watermark structure

memset(&wt, 0, sizeof(wt));

// size of the structure

wt.m_Size = sizeof(wt);

// This is text watermark

wt.m_Type = WaterType_Text;

// Set 'Tahoma' font and its parameters

lstrcpyW(wt.m_FontName, L"Tahoma");
wt.m_FontWeight = FW_NORMAL;
wt.m_bItalic = FALSE;
wt.m_FontSize = P2L(18);
wt.m_Mode = TextRenderingMode_Fill;
wt.m_LineWidth = P2L(0.5);

// Colors

wt.m_FColor = RGB(255, 0, 0);
wt.m_SColor = RGB(255, 0, 0);

// Commmon Part

wt.m_Align = TextAlign_Center;
wt.m_Angle = 0.0;

// Place watermak into odd pages only

wt.m_PlaceType = PlaceType_OddPages;
wt.m_PlaceOrder = PlaceOrder_Background;
wt.m_Opacity = 100;

// Watermark text

lstrcpyW(wt.m_Text, L"www.docu-track.com");

// Add watermak

HRESULT hr = PXC_AddWatermark(pdf, &wt);
if (IS_DS_FAILED(hr))
{
    // Handle error
    ...
}
```

2.6.12 PXC_SetAnnotsInfo

PXC_SetAnnotsInfo

Top Previous Next

PXC_SetAnnotsInfo sets the general annotation parameters (color, border type and width etc.) for the document.

These general parameters are used when the appropriate parameter for the functions **PXC_AddLink**, **PXC_AddLineAnnotation**, **PXC_AddLaunchActionA**, **PXC_AddTextAnnotationA** have a value of `NULL`.

```
HRESULT  PXC_SetAnnotsInfo(
    _PXCDocument* pdf,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by function **PXC_NewDocument**.

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure describing attributes of the annotation.

Return Values

If the function succeeds, the return value is non-negative integer.
If the function fails, the return value is an **error code**.
To determine if the function was successful use the defined macro's as described here: **Error Handling**.

Example (C++).

```
// Example shows, how to add several links
// using the same common annotation information to the document

   void AddSeveralLinks(_PXCDocument* pDoc,
          _PXCContent* pContent, LPCSTR LinkURL, DWORD LinkCount)
   {
       HRESULT hr = DS_OK;

       // Specify the 'start' rect to the annotation

       PXC_RectF rect;
       rect.left = 20.0;
       rect.top = 200.0;
       rect.right = 320.0;
       rect.bottom = 250.0;

       // Common annotation information

       PXC_CommonAnnotInfo AnnotInfo;
       AnnotInfo.m_Color = RGB(200, 0, 100);
```

```
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Set common information for all further annotation functions

        hr = PXC_SetAnnotsInfo(pDoc, &AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // Now place several links onto page

        for (DWORD lc = 0; lc < LinkCount; lc++)
        {
            // Place link
            // the last argument is NULL as the 'global' annotation
information is used

            hr = PXC_AddLink(pContent, &rect, LinkURL, NULL);
            if (IS_DS_FAILED(hr))
            {
                // report error
                ...
            }

            // Now 'move' the rect of the link

            rect.top += 51;
            rect.bottom = rect.top + 50;
        }

        // done.
    }
```

3 XCPRO40 LIB Functions

XCPRO40 LIB Functions

Top Previous Next

The functions found in the XCPRO40 Library are primarily aimed at manipulating existing PDF files - rather than creating new PDF files from NON PDF content.

One exception to the above - is if you are extracting content from an existing PDF file - to create a new PDF file etc.

3.1 High-Level API

High-Level API

Top Previous Next

PDF-XChange Pro Library library consists of two major components + some supporting and interacting elements.

The PDF-Tools library primarily (but not exclusively) utilises the XCPRO40 library dll which is divided into 2 distinct function sets - the `High-Level` and `Low-Level` API.

The `High-Level` API may be used entirely independently of the `Low-Level` API and is usually used for High level operations with existing PDF documents (such as copying pages, managing bookmarks etc).

Whereas the `Low-Level` API functions provide a powerful and flexible means to create both new PDF Pages/Documents and manipulate the content of existing PDF pages and files at the very lowest level. This set of functions offers a developer with advanced skills and an in-depth knowledge of the PDF format and structure, virtually unlimited and unrivalled flexibility in editing and creating PDF documents.

Extreme caution and care should be exercised when using the power and flexibility of the `Low-Level` API and this set of functions requires a developer to be very familiar with both the PDF format and the Adobe PDF format reference documentation. `Low-Level` API functions rely on functions from the `High-Level` API to read as well as write your

3.1.1 Document Operations

3.1.1.1 PXCp_AddWatermark

PXCp_AddWatermark

Top Previous Next

PXCp_AddWatermark adds a watermark to the PDF object.

```
HRESULT  PXCp_AddWatermark(
    PDFDocument pDocument,
    PXC_Watermark* watermark
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

watermark

[in] Pointer to the **PXC_Watermark** structure that describes the watermark to be added.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to add a watermark for each page in the document

    void AddWatermarks(PDFDocument hDoc)
    {
        HRESULT hr = DS_OK;

        // Watermark structure
        PXC_Watermark Water;
        ::ZeroMemory(&Water, sizeof(PXC_Watermark));
        Water.m_Size                = sizeof(PXC_Watermark);

        // To add image watermark uncomment next 2 lines:
        //const WCHAR* pwFileName = L"F:\\_test_image_.bmp";
        //::lstrcpynW(Water.m_FileName, pwFileName, sizeof(Water.m_FileName));

        Water.m_Opacity                = 128;
        Water.m_PlaceOrder                = PlaceOrder_Background;

        // Place to all pages
        Water.m_PlaceType                = PlaceType_AllPages;

        // To add image watermark uncomment next line:
        //Water.m_Type                = WaterType_Image;

        Water.m_Type                = WaterType_Text;
        Water.m_bKeepAspect        = FALSE;
        Water.m_Width                = 0.0;
        Water.m_Height                = 0.0;
        Water.m_Angle                = 30.0;
        Water.m_bItalic                = TRUE;
        Water.m_SColor                = RGB(255, 0, 0);
        Water.m_FColor                = RGB(0, 255, 0);
        Water.m_Mode                = TextRenderingMode_FillStroke;
        Water.m_Align                = TextAlign_Center | TextAlign_VCenter;

        // This is the text of the watermark
        ::lstrcpynW(Water.m_Text, L"Sample", sizeof(Water.m_Text));

        // This is the font to use for the text drawing
        ::lstrcpynW(Water.m_FontName, L"Helvetica", sizeof(Water.m_FontName));


        hr = PXCp_AddWatermark(hDoc, &Water);

        // Check for error
    }
```

3.1.1.2 PXCp_CheckPassword

PXCp_CheckPassword

Top Previous Next

PXCp_CheckPassword validates the supplied password against the current document. This function should only be called after **PXCp_ReadDocumentW** returns **`PS_ERR_DocEncrypted`**.

```
HRESULT  PXCp_CheckPassword(
    PDFDocument pObject,
    BYTE* pPassword,
    DWORD PassLen
);
```

Parameters

pObject
[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

pPassword
[in] *pPassword* specifies a pointer to a buffer which contains password data (buffer may contain zero `'\0'` symbol(s)).

PassLen
[in] *PassLen* specifies the length of the buffer.

Return Values

If the function succeeds, the return value is one of the following:

Value	Meaning
1	User password
2	Owner password

If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Generic example on how to read the specified document
   HRESULT hr = PXCp_ReadDocumentW(hDocument, FileName, 0);
    if (IS_DS_FAILED(hr))
    {
        if (hr == PS_ERR_DocEncrypted)
        {
            while (IS_DS_FAILED(hr))
            {
                BYTE*         Password;
                DWORD         PassLen;
                // Obtain password (i.e. showing some dialog)

                // ...

                // Check password
                hr = PXCp_CheckPassword(hDocument, Password, PassLen);
            }
            // Finish read document
            hr = PXCp_FinishReadDocument(hDocument, 0);
            if (IS_DS_FAILED(hr))
            {
                // In this case document appears to be corrupted
                // ...
            }
        }
        else
        {
            PXCp_Delete(hDocument);
            // In this case document appears to be corrupted
            // ...
        }
    }
    // In this place the document is completely read.
```

3.1.1.3 PXCp_Delete

PXCp_Delete

Top Previous Next

PXCp_Delete releases the PDF object, created previously using the **PXCp_Init** function. You must call this function once the PDF object is no longer required and/or updates are complete.

```
HRESULT  PXCp_Delete(
    PDFDocument pObject
);
```

Parameters

pObject
[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes

page.

Example (C++).

```
  PDFDocument        hDocument = NULL;
   // Please note - RegCode and DevCode are case sensitive
   LPCSTR regcode = "<Your personal serial/keycode code here>";
   LPCSTR devcode = "<Your personal developers' code here>";
   HRESULT res = PXCp_Init(&hDocument, regcode, devcode);
   if (IS_DS_FAILED(res))
       return res;
   ...
   // After all operations with the document are done -
   // then it should be deleted, to release all of the memory
   // used by it
   PXCp_Delete(hDocument);
```

3.1.1.4 PXCp_EnableSecurity

PXCp_EnableSecurity

Top Previous Next

PXCp_EnableSecurity enables/disables Security for the document.

This function is deprecated by function **PXCp_EnableSecurityEx**.

```
HRESULT  PXCp_EnableSecurity(
    PDFDocument pObject,
    BOOL bEnable,
    LPCSTR UserPwd,
    LPCSTR OwnerPwd
);
```

Parameters

pObject

[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

bEnable

[in] *bEnable* specifies whether security should be applied to the document when modifying file.

UserPwd

[in] *UserPwd* specifies a `NULL` terminated ASCII string that represents a `user` password for the document.

OwnerPwd

[in] *OwnerPwd* specifies a `NULL` terminated ASCII string that represents an `owner` password for the document.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
  PDFDocument        hDocument = NULL;
   // document passwords:
   LPCSTR    UserPassword = "user pass";
   LPCSTR    OwnerPassword = "owner pass";
   // switch on security for the document, and set user and owner password:
   HRESULT res = PXCp_EnableSecurity(hDocument, TRUE, UserPassword,
OwnerPassword);
   if (IS_DS_FAILED(res))
   {
       // Report an error
   }
   ...
   // write the document
   // ...
   // Clean up
   PXCp_Delete(hDocument);
   // Now after writing the document while opening it in the PDF viewr
   // one will be asked for the password
```

3.1.1.5 PXCp_EnableSecurityEx

PXCp_EnableSecurityEx

Top Previous Next

PXCp_EnableSecurityEx enables or disables PDF security for the document.

```
HRESULT  PXCp_EnableSecurityEx(
    PDFDocument* pObject,
    PXC_SecurityMethod nMethod,
    LPCSTR UserPwd,
    LPCSTR OwnerPwd
);
```

Parameters

pObject

[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

nMethod

[in] Specifies the enctyption method used to encrypt the document content. Can be one of the following value:

Constant	Value	Meaning
psm_None	0	Security disabled. Parameters *UserPwd* and *OwnerPwd* of this function will be ignored.
psm_RC4	1	RC4 encryption method will be used. RC4 can has 40-bit or 128-bit key length.
psm_AES	2	AES 128 bit encryption method will be used.

If the value of this parameter is `psm_AES`, or `psm_RC4`, security will be enabled with the specified user and owner passwords. Functions **PXCp_SetPermissions** should be called after this function to specify user's permission for the document.

Note: With AES encryption only key length 128 can be used.

UserPwd

[in] *UserPwd* specifies a `NULL` terminated ASCII string that represents a `user` password for the document.

OwnerPwd

[in] *OwnerPwd* specifies a `NULL` terminated ASCII string that represents an `owner` password for the document.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
  PDFDocument        hDocument = NULL;
   // document passwords:
   LPCSTR    UserPassword = "user pass";
   LPCSTR    OwnerPassword = "owner pass";
   // switch on security for the document, and set user and owner password:
   HRESULT res = PXCp_EnableSecurityEx(hDocument, psm_RC4, UserPassword,
OwnerPassword);
   if (IS_DS_FAILED(res))
   {
       // Report an error
   }
   ...
   // write the document
   // ...
   // Clean up
   PXCp_Delete(hDocument);
   // Now after writing the document while opening it in the PDF viewr
   // one will be asked for the password
```

3.1.1.6 PXCp_FinishReadDocument

PXCp_FinishReadDocument

Top Previous Next

PXCp_FinishReadDocument completes the reading of an encrypted document after **PXCp_ReadDocumentW** returns **`PS_ERR_DocEncrypted`**.

```
HRESULT  PXCp_FinishReadDocument(
     PDFDocument pObject,
     DWORD Flags
);
```

Parameters

pObject

[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

Flags

[in] *Flags* this flag is reserved for future use and should be set to `0`.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

This function should be called only once **PXCp_ReadDocumentW** has returned **`PS_ERR_DocEncrypted`**, however, in the case of a successful call to the function **PXCp_ReadDocumentW** there is no need to call the function.

Example (C++).

```
// Generic example on how to read a document
   HRESULT hr = PXCp_ReadDocumentW(hDocument, FileName, 0);
    if (IS_DS_FAILED(hr))
    {
        if (hr == PS_ERR_DocEncrypted)
        {
            while (IS_DS_FAILED(hr))
            {
                BYTE*        Password;
                DWORD        PassLen;
                // Obtain password (i.e. showing some dialog)

                // ...

                // Check password
                hr = PXCp_CheckPassword(hDocument, Password, PassLen);
            }
            // Finish reading the document
            hr = PXCp_FinishReadDocument(hDocument, 0);
            if (IS_DS_FAILED(hr))
            {
                // In this case document appears to be corrupt
                // ...
            }
        }
        else
        {
            PXCp_Delete(hDocument);
            // In this case document appears to be corrupt
            // ...
        }
    }
    // At this point the document is completely read.
```

3.1.1.7 PXCp_GetPermissions

PXCp_GetPermissions

Top Previous Next

PXCp_GetPermissions extracts the encryption level and user's permissions set for the document.

```
HRESULT  PXCp_GetPermissions(
    PDFDocument pDocument,
    DWORD* enclevel,
    DWORD* permFlags
);
```

Parameters

pDocument
: [in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

enclevel
: [in, out] specifies the pointer to a variable of the `DWORD` type, which receives encryption level information for the document. Possible values are `40` and `128`.

permFlags
: [in, out] specifies the pointer to the variable of the `DWORD` type which receives permission flags information for the document.
Possible values are combinations of those described in the **PXCp_SetPermissions** function.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
    PDFDocument  hDocument;
    BOOL         DocIsEncrypted = FALSE;
    // Check if the document is encrypted?:
    HRESULT res = PXCp_IsEncrypted(hDocument, &DocIsEncrypted);
    if (IS_DS_FAILED(res))
    {
        // Report an error
    }
    if (DocIsEncrypted)
    {
        // obtaine encryption parameters
        DWORD        EncryptParameters = 0;
        DWORD        PermissionFlags = 0;
        res = PXCp_GetPermissions(hDocument, &EncryptParameters,
&PermissionFlags);
        // ...
    }
    ...
    // Clean up
    PXCp_Delete(hDocument);
```

3.1.1.8 PXCp_Init

PXCp_Init

Top Previous Next

PXCp_Init creates a PDF object, usually required by (and called before) the majority of functions in the **PDF-XChange Pro Library** - either explicitly or implicitly.

```
HRESULT  PXCp_Init(
    PDFDocument* pObject,
    LPCSTR Key,
    LPCSTR DevCode
);
```

Parameters

pObject

[in, out] Pointer to the variable of a type `PDFDocument` that will receive the created PDF object.

Key

[in] Pointer to a null-terminated string which contains your licence key for use with **PDF-XChange Pro Library**. This parameter may be `NULL`, if so, the library will operate in 'evaluation' mode and a demo stamp/watermark will be printed on all output - such stamps cannot be removed subsquently.

DevCode

[in] Pointer to a null-terminated string which contains your individual developer code for use with **PDF-XChange Pro Library**. This parameter can be `NULL`, if so, the library will operate in 'evaluation' mode and a demo stamp/watermark will be printed on all output - such stamps cannot be removed subsquently.

Return Values

If the function succeeds, the return value is `DS_OK`, and a variable pointer to *pObject* will contain the valid PDF object.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
    PDFDocument        hDocument = NULL;
    // Please note, RegCode and DevCode are case sensitive
    LPCSTR regcode = "<Your personal serial/keycode code here>";
    LPCSTR devcode = "<Your personal developers' code here>";
    HRESULT res = PXCp_Init(&hDocument, regcode, devcode);
    if (IS_DS_FAILED(res))
```

```
        return res;
    ...
    PXCp_Delete(hDocument);
```

3.1.1.9 PXCp_IsEncrypted

PXCp_IsEncrypted

Top Previous Next

PXCp_IsEncrypted checks if the document is encrypted.

```
HRESULT  PXCp_IsEncrypted(
     PDFDocument pObject,
     BOOL* bEncrypted
);
```

Parameters

pObject
[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

bEncrypted
[in, out] *bEncrypted* specifies a pointer to a variable of the `BOOL` type which receives the results.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
    PDFDocument   hDocument;
    BOOL          DocIsEncrypted = FALSE;
    // Check if the document is encrypted?:
    HRESULT res = PXCp_IsEncrypted(hDocument, &DocIsEncrypted);
    if (IS_DS_FAILED(res))
    {
        // Report any error
    }
    if (DocIsEncrypted)
    {
        // i.e. obtaine encryption parameters, etc
        // ...
    }
    ...
    // Clean up
    PXCp_Delete(hDocument);
```

3.1.1.10 PXCp_PlaceContents

PXCp_PlaceContents

Top Previous Next

PXCp_PlaceContents overlays content from the specified page(s) of an Adobe PDF document source document as background or foreground content on the specified page(s) of a destination Adobe PDF document, this function is similar to the image based watermark functionality also available within the library.

```
HRESULT  PXCp_PlaceContents(
    PDFDocument pDest,
    PDFDocument pSource,
    size_t cnt,
    PXCp_ContentPlaceInfo* pContentsInfo,
    DWORD flags
);
```

Parameters

pDest

[in] *pDest* specifies the PDF object previously created by the function **PXCp_Init**, which will be used as the destination for placing the page contents.

pSource

[in] *pSource* specifies the PDF object previously created by the function **PXCp_Init**, which will be used as the source for retrieving the pages contents to place on the target document pages.

cnt

[in] *cnt* specifies the count of the **PXCp_ContentPlaceInfo** structures pointed by *pContentsInfo*.

pContentsInfo

[in] *pContentsInfo* specifies a pointer to the first element of the **PXCp_ContentPlaceInfo** structures array.

flags

[in] *flags* specifies operation flags, reserved for future use. Please set to zero for compatibitity.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

It is strongly recommended against calling this function several times with the same source and destination documents. If multiple overlays are required, these can be achieved in a single pass (by simply merging the arrays of the **PXCp_ContentPlaceInfo** structures into one) - additionally, this will have the added benefit of reducing the size of the resulting pdf over the alternate multi pass method.

Example (C++).

```
    // Example shows how to place the page from a source document
// over all pages in the target document

void PlacePageOver(PDFDocument hSource, PDFDocument hTarget)
{
        HRESULT hr = DS_OK;

        // Number of pages in the target document

        DWORD PageCnt = 0;

        // Retrieve the number of pages in the target document
        PXCp_GetPagesCount(hTarget, &PageCnt);

        // Prepare information for placing the contents

        PXCp_ContentPlaceInfo*        pci = new PXCp_ContentPlaceInfo
[PageCnt];

        for (DWORD i = 0; i < PageCnt; i++)
        {
            // Allignmet of the contents
            pci[i].Alignment = CPA_HorFit | CPA_VerFit | CPA_Foreground;
            // Destination page in the target document
            pci[i].DestPage = i;
            // Source page (first one) from the source document
            pci[i].SrcPage = 0;
        }

        // Place contents

        hr = PXCp_PlaceContents(hTarget, hSource, PageCnt, pci, 0);

        // Clean up

        delete[] pci;

        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // done.
    }
```

3.1.1.11 PXCp_ReadDocumentA

PXCp_ReadDocumentA

Top Previous Next

PXCp_ReadDocumentA reads the document from file.

```
HRESULT  PXCp_ReadDocumentA(
    PDFDocument pObject,
    LPCSTR pwFileName,
    DWORD Reserved
);
```

Parameters

pObject

[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

pwFileName

[in] *pwFileName* specifies a pointer to a `NULL` terminated UNICODE string that contains the full path to the file.

Reserved

[in] *Reserved* this argument is reserved for further usage and should be set to `0`.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function return value is equal to **`PS_ERR_DocEncrypted`**, then a password must be provided using **PXCp_CheckPassword** and **PXCp_FinishReadDocument** must be called to complete reading and parsing the document.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page

Comments

This function is the ASCII equivalent of the **PXCp_ReadDocumentW** function.

Example (C++).

```
// Generic example to read a document using the library
   HRESULT hr = PXCp_ReadDocumentA(hDocument, FileName, 0);
   if (IS_DS_FAILED(hr))
   {
       if (hr == PS_ERR_DocEncrypted)
       {
           while (IS_DS_FAILED(hr))
           {
               BYTE*        Password;
               DWORD        PassLen;
               // Obtain password (i.e. showing some dialog)

               // ...

               // Check password
               hr = PXCp_CheckPassword(hDocument, Password, PassLen);
           }
           // Finish read document
           hr = PXCp_FinishReadDocument(hDocument, 0);
           if (IS_DS_FAILED(hr))
           {
               // In this case document appears to be corrupted
               // ...
           }
       }
       else
       {
           PXCp_Delete(hDocument);
           // In this case document appears to be corrupted
           // ...
       }
   }
   // In this place the document is successfully read.
```

3.1.1.12 PXCp_ReadDocumentW

PXCp_ReadDocumentW

Top Previous Next

PXCp_ReadDocumentW reads the document from file.

```
HRESULT  PXCp_ReadDocumentW(
    PDFDocument pObject,
    LPCWSTR pwFileName,
    DWORD Reserved
);
```

Parameters

pObject
[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

pwFileName
[in] *pwFileName* specifies a pointer to a `NULL` terminated UNICODE string that contains the full path to the file.

Reserved
[in] *Reserved* this argument is reserved for further usage and should be set to `0`.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function return value is equal to **`PS_ERR_DocEncrypted`**, then a password must be provided using **PXCp_CheckPassword** and **PXCp_FinishReadDocument** must be called to complete reading and parsing the document.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function is the UNICODE equivalent of the **PXCp_ReadDocumentA** function.

Example (C++).

```
// Generic example to read a document using the library
    HRESULT hr = PXCp_ReadDocumentW(hDocument, FileName, 0);
    if (IS_DS_FAILED(hr))
    {
        if (hr == PS_ERR_DocEncrypted)
        {
            while (IS_DS_FAILED(hr))
            {
                BYTE*        Password;
                DWORD        PassLen;
                // Obtain password (i.e. showing some dialog)

                // ...

                // Check password
                hr = PXCp_CheckPassword(hDocument, Password, PassLen);
            }
            // Finish read document
            hr = PXCp_FinishReadDocument(hDocument, 0);
            if (IS_DS_FAILED(hr))
            {
                // In this case document appears to be corrupted
                // ...
            }
        }
        else
        {
            PXCp_Delete(hDocument);
            // In this case document appears to be corrupted
            // ...
        }
    }
    // In this place the document is successfully read.
```

3.1.1.13 PXCp_SetCallBack

PXCp_SetCallBack

Top Previous Next

PXCp_SetCallBack sets the callback function for use during extended operations with a document, i.e. writing to a file, optimization, etc.

```
HRESULT  PXCp_SetCallBack(
    PDFDocument pObject,
    CALLBACK_FUNC pProc,
    LPARAM UserData
);
```

Parameters

pObject

[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

pProc

[in] *pProc* specifies the callback function, must be defined as **CALLBACK_FUNC**:

```
typedef BOOL (__stdcall *CALLBACK_FUNC)(DWORD state, DWORD level, LPARAM
param);
```

The first parameter of this function is the callback state, the second indicates the progress level (see table below), and the third will always have the same value as passed in *UserData*.

Callback function's state constants table

Constant	Value	Meaning of level
`PXClb_Start`	1	`MaxVal` - maximum value of the level which will be passed
`PXClb_Processing`	2	Current progress level - any value from `0` to `MaxVal`
`PXClb_Finish`	3	May be any value from `0` to `MaxVal` (`MaxVal` if all passed), may be ignored

Note: The Callback function should return `TRUE` (any non-zero value) to continue processing or `FALSE` (zero) to abort the operation.

UserData

[in] *UserData* specifies a user-defined callback parameter to be passed as a third parameter to the function specified by *pProc*.

Return Values

If the current operation is terminated by returning `FALSE` from a callback then the current operation will return **`DPro_ERR_USER_BREAK`**.
If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Here is example of a simple callback function and its usage.
// This example is similar to one available the from pxclib30 help, as the
definition of callback
// function as well as the constants are the same

// This variable will store maximum values of the progress indicator
// We will pass its address as clbParam, to enable us to
// access it in our callback function.
DWORD dwMaxLevel;

BOOL __stdcall SampleCallback(DWORD state, DWORD level, LPARAM param)
{
    // Get pointer to dwMaxLevel;
    DWORD* pMaxLevel = (DWORD*)param;
    // Analise state
    switch (state)
    {
```

```
    case PXClb_Start: // start
        // store maximum value into dwMaxLevel
        *pMaxLevel = level;
        break;
    case PXClb_Processing: // processing
        // display current progress in percents
        {
            double p = 100.0 * (double)level / (double)(*pMaxLevel);
            printf("\r%.2f%%", p);
        }
        break;
    case PXClb_Finish: // finished
        // display final progress
        {
            double p = 100.0 * (double)level / (double)(*pMaxLevel);
            printf("\r%.2f%%, done.\n", p);
        }
        break;
    }
    return TRUE; // Always return TRUE to continue work
}

HRESULT SampleOfUse()
{
    // Create document
    PDFDocument* hDocumetn = NULL;
    LPCSTR Key = "<Enter here valid key>";
    LPCSTR DevCode = "<Enter here valid developer's code>";
    HRESULT res = PXCp_Init(&hDocument, Key, DevCode);
    if (IS_DS_FAILED(res))
        return res;
    // Set callback function and address of dwMaxLevel as parameter
    res = PXCp_SetCallBack(hDocumetn, SampleCallback, (LPARAM)&dwMaxLevel);
    if (IS_DS_FAILED(res))
    {
        // Do not forget to free the pdf document!
        PXCp_Delete(hDocumetn);
        return res;
    }
    // Some pdf generation code ommited
    ...
    // here we will write the document
    printf("Saving document:\n");
    res = PXCp_WriteDocumentW(hDocumetn, L"c:\\dummy.pdf",
PXCp_CreationDisposition_Overwrite, PXCp_Write_NoRelease);
    // and free it
    PXCp_Delete(hDocumetn);
    return res;
}
```

3.1.1.14 PXCp_SetPermissions

PXCp_SetPermissions

Top Previous Next

PXCp_SetPermissions applies the specified encryption level and user permissions to the document.

```
HRESULT  PXCp_SetPermissions(
    PDFDocument pObject,
    DWORD enclevel,
    DWORD permFlags
);
```

Parameters

pObject

[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

enclevel

[in] Specifies the encryption level for the document. Supported values are `40` and `128`.

permFlags

[in] This parameter specifies the permission flags for the document. This may be a logical combination of the following values:

Value	Meaning
Permit_Printing	When *enclevel* equals to `40` - print the document. When *enclevel* greater `40` - print the document (possibly not at the highest quality level, depending on whether `Permit_HighQualityPrinting` is set).
Permit_Modification	Modify the content of the document by operations other then those controlled by `Permit_Add_And_Modify_Annotations`, `Permit_FormFilling`, `Permit_Assemble`.
Permit_Copying_And_TextGraphicsExtractions	When *enclevel* is equal to `40` - Copy or otherwise extract text and graphics from the document, including extracting text and graphics (in support of accessibility to disabled users or for other purposes). When *enclevel* is greater than `40` - Copy or otherwise extract text and graphics from the document by operations other than that controlled by Permit_TextGraphicsExtractions.
Permit_Add_And_Modify_Annotations	Add or modify text annotations, fill in interactive form fields, and, if `Permit_Modification` is also set, create or modify interactive form fields (including signature fields).
Permit_FormFilling	Only when *enclevel* greater then `40`. Fill in the existing form interactive form fields (including signature fields), even if the

	Permit_Add_And_Modify_Annotations property is not set.
Permit_TextGraphicsExtractions	Only when *enclevel* greater then 40. Extract text and graphics (in support of accessibility for disabled users or for other purposes).
Permit_Assemble	Only when *enclevel* equals to 40. Assemble the document (insert, rotate,delete pages, create bookmarks and/or thumbnail images), even if the Permit_Modification property is not set.
Permit_HighQualityPrinting	Only when *enclevel* is greater than 40. Print the document as a faithful digital copy of the PDF content generated. If this permission is not set (and Permit_Printing is set), printing is limited to a low-level representation of the document - possibly of degraded quality.
Permit_Nothing	No operations are permitted.
Permit_All	All operations for the document are permitted.

For more information about the field values of this flag, developers should see Adobe's comprehensive documentation for the PDF format freely available from the Adobe web site.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
PDFDocument        hDocument = NULL;
DWORD Permissions = 0;
// Permits document printing and extracting text and graphics:
Permissions = Permit_Printing | Permit_TextGraphicsExtractions;
// Set this options (key length will be set to 40):
HRESULT res = PXCp_SetPermissions(&hDocument, 40, Permissions);
if (IS_DS_FAILED(res))
{
    // Report an error
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.1.15 PXCp_SignDocumentBufW

PXCp_SignDocumentBufW

Top Previous Next

PXCp_SignDocumentBufW adds a digital signature to the document, signs and places as required on the specified page. Uses a certificate stored in PKCS#7 format within the specified memory buffer.

```
HRESULT  PXCp_SignDocumentBufW(
    PDFDocument pdf,
    LPBYTE pPXCBuf,
    DWORD nPFXLen,
    LPCWSTR lpwszPFXPassword,
    DWORD pageIndex,
    LPCPXC_RectF rect,
    LPCWSTR lpwszReason,
    LPCWSTR lpwszLocation,
    LPCWSTR lpwszContactInfo,
    LPCWSTR lpwszImageFile,
    DWORD dwFlags
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXCp_Init**.

pPXCBuf

[in] Pointer to a memory buffer where the signatories certificate (in PKCS#7 format) is stored. If there is more than one certificate stored within the specified buffer, the first located will be used.

nPFXLen

[in] Specifies the length in bytes of the buffer addressed by the *pPXCBuf* parameter.

lpwszPFXPassword

[in] A string password used to decrypt and verify the PFX packet from the *pPXCBuf* buffer.

pageIndex

[in] Parameter *pageIndex* specifies the index of the page on which the signature field should be placed.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the signature field.

lpwszReason

[in] Pointer to a null-terminated string that specifies the reason for the signing, such as (`I agree...`). This parameter may be `NULL`.

lpwszLocation

[in] Pointer to a null-terminated string that specifies the CPU host name or physical location of the signing. This parameter may be `NULL`.

lpwszContactInfo

[in] Pointer to a null-terminated string that specifies the information provided by the signatory to enable a recipient to contact the signatory to verify the signature; for example, a phone number. This parameter may be `NULL`.

lpwszImageFile

[in] Specifies the full path and file name of the image (if any) to be displayed within the signature field. This parameter may be `NULL`.

dwFlags

[in] Combination of flags which determines how the signature field should appear on the page. For

more information about possible values, see the **PXCp_SignDocumentW** function.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
       // Example shows how to sign the document before writing
    // using certificate stored in the buffer

    void SignDocument(PDFDocument hDoc, LPBYTE buffer_PFX, DWORD bufLen,
LPCWSTR Password_PFX, LPCWSTR FileName_Image)
    {
        HRESULT hr = DS_OK;

        // Setup rectangle

        PXC_RectF sr;
        sr.left = I2L(1);
        sr.right = I2L(4);
        sr.top = I2L(9);
        sr.bottom = I2L(8);

        // Sign the document

        hr = PXCp_SignDocumentBufW(hDoc, buffer_PFX, DWORD bufLen,
Password_PFX, 0, &sr,
                    L"Test Reason", L"Test Location", L"Test Contact Info",
FileName_Image,
                    Sign_GR_Name | Sign_TX_Name | Sign_TX_Date |
Sign_TX_Location | Sign_TX_Reason | Sign_TX_DName);

        if (IS_DS_FAILED(hr))
        {
            // Handle error
            ...
        }

        // done.
    }
```

3.1.1.16 PXCp_SignDocumentUsingPFXW

PXCp_SignDocumentUsingPFXW

Top Previous Next

PXCp_SignDocumentUsingPFXW adds a digital signature to the document , signs and places as required on the specified page. Uses a certificate stored in a PKCS#7 file.

```
HRESULT  PXCp_SignDocumentUsingPFXW(
    PDFDocument pdf,
    LPCWSTR lpwszPFXFile,
    LPCWSTR lpwszPFXPassword,
    DWORD pageIndex,
    LPCPXC_RectF rect,
    LPCWSTR lpwszReason,
    LPCWSTR lpwszLocation,
    LPCWSTR lpwszContactInfo,
    LPCWSTR lpwszImageFile,
    DWORD dwFlags
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXCp_Init**.

lpwszPFXFile

[in] Pointer to a null-terminated string that specifies the full path and name of the PKCS#7 file where the signatories certificate is stored. If there is more than one certificate stored within the specified file, the first located will be used.

lpwszPFXPassword

[in] String password used to decrypt and verify the PFX packet from the *lpwszPFXFile* file.

pageIndex

[in] Parameter *pageIndex* specifies the index of the page on which the signature field should be placed.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the signature field.

lpwszReason

[in] Pointer to a null-terminated string that specifies the reason for the signature, such as (`I agree...`). This parameter may be `NULL`.

lpwszLocation

[in] Pointer to a null-terminated string that specifies the CPU host name or physical location of the signature. This parameter may be `NULL`.

lpwszContactInfo

[in] Pointer to a null-terminated string that specifies information provided by the signatory to enable a recipient to contact the signer to verify the signature; for example, a phone number. This parameter may be `NULL`.

lpwszImageFile

[in] Specifies the full path and file name of the image (if any) to be displayed within the signature field. This parameter may be `NULL`.

dwFlags

[in] Combination of flags which determine how the signature field should appear on the page. For more information about possible values, see the **PXCp_SignDocumentW** function.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows how to sign the document before writing
// using certificate stored in the file

   void SignDocument(PDFDocument hDoc, LPCWSTR FileName_PFX, LPCWSTR
Password_PFX, LPCWSTR FileName_Image)
   {
       HRESULT hr = DS_OK;

       // Setup rectangle

       PXC_RectF sr;
       sr.left = I2L(1);
       sr.right = I2L(4);
       sr.top = I2L(9);
       sr.bottom = I2L(8);

       // Sign the document

       hr = PXCp_SignDocumentUsingPFXW(hDoc, FileName_PFX, Password_PFX, 0,
&sr,
                   L"Test Reason", L"Test Location", L"Test Contact Info",
FileName_Image,
                   Sign_GR_Name | Sign_TX_Name | Sign_TX_Date |
Sign_TX_Location | Sign_TX_Reason | Sign_TX_DName);

       if (IS_DS_FAILED(hr))
       {
           // Handle error
           ...
       }

       // done.
   }
```

3.1.1.17 PXCp_SignDocumentW

PXCp_SignDocumentW

Top Previous Next

PXCp_SignDocumentW adds a digital signature to the document, signs and places as required on the specified page.

```
HRESULT  PXCp_SignDocumentW(
    PDFDocument pdf,
    PCCERT_CONTEXT pCert,
    DWORD pageIndex,
    LPCPXC_RectF rect,
    LPCWSTR lpwszReason,
    LPCWSTR lpwszLocation,
    LPCWSTR lpwszContactInfo,
    LPCWSTR lpwszImageFile,
    DWORD dwFlags
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXCp_Init**.

pCert

[in] *pCert* specifies the certificate context (for more information see the Microsoft MSDN internet resource regarding CryptoAPI documentation) to be used when signing the document.

pageIndex

[in] Parameter *pageIndex* specifies the index of the page on which the signature field should be placed.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the signature field.

lpwszReason

[in] Pointer to a null-terminated string that specifies the reason for the signature, such as (`I agree...`). This parameter may be `NULL`.

lpwszLocation

[in] Pointer to a null-terminated string that specifies the CPU host name or physical location of the signatory. This parameter may be `NULL`.

lpwszContactInfo

[in] Pointer to a null-terminated string that specifies the information provided by the signatory to enable a recipient to contact the signatory to verify the signature; for example, a phone number. This parameter may be `NULL`.

lpwszImageFile

[in] Specifies the full path and file name of the image (if any) to be displayed within the signature field. This parameter may be `NULL`.

dwFlags

[in] Combination of flags which determines how the signature field should appear on the page. May be combination of the following values:

Constant	Value	Meaning

Sign_GR_NoGraphics	0x00000	No graphics element of the signature field will be displayed.
Sign_GR_Image	0x00010	In the graphics element of the signature field the image specified by the *lpwszImageFile* parameter will be displayed.
Sign_GR_Name	0x00020	In the graphics element of the signature field the signatory's name will be displayed.
Sign_TX_Name	0x00100	In the text part of the signature field the signatory's name will be displayed.
Sign_TX_Date	0x00200	In the text part of the signature field the date and time of signing will be displayed.
Sign_TX_Location	0x00400	In the text part of the signature field the location specified by *lpwszLocation* parameter will be displayed.
Sign_TX_Reason	0x00800	In the text part of the signature field the reason of signing as specified by *lpwszReason* parameter will be displayed.
Sign_TX_DName	0x01000	In the text part of the signature field the detailed information about signatory will be displayed.
Sign_TX_Labels	0x08000	If this flag is specified, all text information (eg. Name, Date, etc.) will be labeled on the signature field.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows how to sign the document before writing
// Certificate is being obtained from the system

    void SignDocument(PDFDocument hDoc, LPCSTR SysStorageName, LPCWSTR
FileName_Image)
    {
        HRESULT hr = DS_OK;
        HCERTSTORE       hCertStore = NULL;
        PCCERT_CONTEXT   pCertContext = NULL;

        // Open Storage

        hCertStore = CertOpenSystemStore(NULL, SysStorageName));

        if (!hCertStore)
        {
            // Handle error
            ...
        }

        // Obtain certificate from the system
        pCertContext = CryptUIDlgSelectCertificateFromStore(
                hCertStore,      // Open store containing the certificates to
display
                NULL,
                NULL,
                NULL,
                CRYPTUI_SELECT_LOCATION_COLUMN,
                0,
                NULL);

        if (!pCertContext)
        {
            // Handle error
            ...
        }


        // Setup rectangle

        PXC_RectF sr;
        sr.left = I2L(1);
        sr.right = I2L(4);
        sr.top = I2L(9);
        sr.bottom = I2L(8);

        // Sign the document

        hr = PXCp_SignDocumentW(hDoc, pCertContext, 0, &sr,
                  L"Test Reason", L"Test Location", L"Test Contact Info",
FileName_Image,
                  Sign_GR_Name | Sign_TX_Name | Sign_TX_Date |
Sign_TX_Location | Sign_TX_Reason | Sign_TX_DName);

        if (IS_DS_FAILED(hr))
        {
            // Handle error
            ...
        }

        // When all processing is completed, clean up

        if(pCertContext)
        {
            CertFreeCertificateContext(pCertContext);
        }

        if(hCertStore)
        {
             if (!CertCloseStore(hCertStore,0))
```

```
                {
                        // Handle error
                        ...
                }
        }

        // done.
    }
```

3.1.1.18 PXCp_WriteDocumentA

PXCp_WriteDocumentA

Top Previous Next

PXCp_WriteDocumentA writes the generated PDF document to a (disk) file. This is a `UNICODE` function.

```
HRESULT  PXCp_WriteDocumentA(
    PDFDocument pdf,
    LPCSTR fName,
    PXCp_CreationDisposition CreationDesposition,
    DWORD WriteFlags
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXCp_Init**.

fName

[in] Pointer to a null-terminated string which specifies the fully qualified path and file name, to which the PDF object will be stored.

CreationDesposition

[in] *CreationDesposition* specifies the behaviour of the function should a file with the *fName* name exist already on disk. May be one of following values:

Value	Meaning
PXCp_CreationDisposition_Skip	If the file with the same name exists - then halt the process and return an error.
PXCp_CreationDisposition_Overwrite	Write the file, and should a file of the same name exist, replace with the newly created file.

WriteFlags

[in] *WriteFlags* specifies whether to close the document automatically after writing to disk. May any one of following values:

Value	Meaning
PXCp_Write_Release	After the function call the document is released and **PXCp_Delete** need not to be called. The Document handle is no longer valid after the call.
PXCp_Write_NoRelease	After the function call the document handle is valid and may be used for further operations. Writing a document to a file does not destroy it -

a document may be saved more than once. The handle should be released using the function **PXCp_Delete** when no longer required.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

Note: If the **PXCp_Write_Release** flag is used then the **PXCp_Delete** function should NOT be called for the document handle again as this may lead errors.

This flag may be used for convenience as usually after writing a document it would no longer be retained for further operations.

This function is the ASCII equivalent of UNICODE function **PXCp_WriteDocumentW**.

Remarks

The document may be saved into the the same file name as original in case it is not opened by other program (possibly the viewer or other).

Example (C++).

```
  PDFDocument        hDocument = NULL;
   HRESULT hr;
   // Let FileName is the variable which contains the full file name
   WCHAR FileName[MAX_PATH];
   // ... determine the file name & where to store the document
   // Next variable determines if we allow the existing document to be over
written:
   BOOL bAllowOverwriteExisting = TRUE;
   // Now we can write the document:
   if (bAllowOverwriteExisting)
   {
       // In a case a file with the FileName already exists on disk the
function will replace it with the new one
       hr = PXCp_WriteDocumentA(hDocument, FileName,
PXCp_CreationDisposition_Overwrite, PXCp_Write_NoRelease);
   }
   else
   {
       // In a case where a file with the FileName already exists on disk the
function will return an error
       hr = PXCp_WriteDocumentA(hDocument, FileName,
PXCp_CreationDisposition_Skip, PXCp_Write_NoRelease);
   }
   if (IS_DS_FAILED(hr))
   {
       // Report the error while writing
       // ...
   }
   ...
   // Now the document handle - hDocument - should be released as we used
```

```
PXCp_Write_NoRelease
   // constant to avoid automatic document release:
   PXCp_Delete(hDocument);
```

3.1.1.19 PXCp_WriteDocumentW

PXCp_WriteDocumentW

Top Previous Next

PXCp_WriteDocumentW writes the generated PDF document to a (disk) file. This is a `UNICODE` function.

```
HRESULT  PXCp_WriteDocumentW(
    PDFDocument pdf,
    LPCWSTR fName,
    PXCp_CreationDisposition CreationDesposition,
    DWORD WriteFlags
);
```

Parameters

pdf

[in] *pdf* specifies the PDF object previously created by the function **PXCp_Init**.

fName

[in] Pointer to a null-terminated string which specifies the fully qualified path and file name, to which the PDF object will be stored.

CreationDesposition

[in] *CreationDesposition* specifies the behaviour of the function should a file with the *fName* name exist already on disk. May be one of following values:

Value	**Meaning**
PXCp_CreationDisposition_Skip	If the file with the same name exists - then halt the process and return an error.
PXCp_CreationDisposition_Overwrite	Write the file, and should a file of the same name exist, replace with the newly created file.

WriteFlags

[in] *WriteFlags* specifies whether to close the document automatically after writing to disk. May any one of following values:

Value	**Meaning**
PXCp_Write_Release	After the function call the document is released and **PXCp_Delete** need not to be called. The Document handle is no longer valid after the call.
PXCp_Write_NoRelease	After the function call the document handle is valid and may be used for further operations. Writing a document to a file does not destroy it - a document may be saved more than once. The handle should be released using the function **PXCp_Delete** when no longer required.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.

To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

Note: If the **`PXCp_Write_Release`** flag is used then the **PXCp_Delete** function should NOT be called for the document handle again as this may lead errors.

This flag may be used for convenience as usually after writing a document it would no longer be retained for further operations.

This function is the UNICODE equivalent of ASCII function **PXCp_WriteDocumentA**.

Remarks

The document may be saved into the the same file name as original in case it is not opened by other program (possibly the viewer or other).

Example (C++).

```
  PDFDocument        hDocument = NULL;
   HRESULT hr;
   // Let FileName is the variable which contains the full file name
   WCHAR FileName[MAX_PATH];
   // ... determine the file name & where to store the document
   // Next variable determines if we allow the existing document to be over
written:
   BOOL bAllowOverwriteExisting = TRUE;
   // Now we can write the document:
   if (bAllowOverwriteExisting)
   {
       // In a case a file with the FileName already exists on disk the
function will replace it with the new one
       hr = PXCp_WriteDocumentW(hDocument, FileName,
PXCp_CreationDisposition_Overwrite, PXCp_Write_NoRelease);
   }
   else
   {
       // In a case where a file with the FileName already exists on disk the
function will return an error
       hr = PXCp_WriteDocumentW(hDocument, FileName,
PXCp_CreationDisposition_Skip, PXCp_Write_NoRelease);
   }
   if (IS_DS_FAILED(hr))
   {
       // Report the error while writing
       // ...
   }
   ...
   // Now the document handle - hDocument - should be released as we used
PXCp_Write_NoRelease
   // constant to avoid automatic document release:
   PXCp_Delete(hDocument);
```

3.1.2 Document Information

3.1.2.1 PXCp_SetSpecVersion

PXCp_SetSpecVersion

Top Previous Next

PXCp_SetSpecVersion sets the version of the Adobe PDF Format to be used for the document when written to a file.

```
HRESULT  PXCp_SetSpecVersion(
    PDFDocument pDocument,
    PXC_SpecVersion ver
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

ver
[in] *ver* specifies the variable for the `PXC_SpecVersion` type (for details refer to **PDF-XChange Library** help).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
PDFDocument        hDocument;
PXC_SpecVersion SpecVersion;
// Let's set the version to PDF 1.4:
SpecVersion = SpecVersion14;
// Set this options:
HRESULT res = PXCp_SetSpecVersion(hDocument, SpecVersion);
if (IS_DS_FAILED(res))
{
    // Report any error
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.2.2 PXCp_GetDocumentInfoA

PXCp_GetDocumentInfoA

Top Previous Next

PXCp_GetDocumentInfoA retrieves standard information field data regarding the structure of a pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

```
HRESULT  PXCp_GetDocumentInfoA(
    PDFDocument pDoc,
    PXC_StdInfoField field,
    LPSTR value,
    DWORD* bufLen
);
```

Parameters

pDoc

[in] *pDoc* specifies the PDF object previously created by the function **PXCp_Init**.

field

[in] *field* specifies an information tag to retrieve. Possible values are:

Value	**Meaning**
InfoField_Title	Sets the **Title** field in the info structure of the pdf file.
InfoField_Subject	Sets the **Subject** field in the info structure of the pdf file.
InfoField_Author	Sets the **Author** field in the info structure of the pdf file.
InfoField_Keywords	Sets the **Keywords** field in the info structure of the pdf file.
InfoField_Creator	Sets the **Creator** field in the info structure of the pdf file.
InfoField_Producer	Sets the **Producer** field in the info structure of the pdf file.

value

[in, out] *value* specifies a pointer to a buffer where the information should be inserted.
Note: To determine the required buffer size you should pass NULL as *value*.
In this case the function will return **DPro_Wrn_NeedGreaterBuffer**.

bufLen

[in, out] *bufLen* specifies an available buffer size in characters (including a null-terminating character).
Note: When *value* is set to NULL then *bufLen* will contain the required buffer size

Return Values

If the function succeeds, the return value is DS_OK.
If the required buffer size is greater then that available *bufLen* the function returns **DPro_Wrn_NeedGreaterBuffer**.
If there is no information set for the document - the function returns **DPro_Wrn_InfoNotSet**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function is the ASCII equivalent of UNICODE function **PXCp_GetDocumentInfoW**.

Example (C++).

```
// Example on how to retrieve information tag from the document
    LPWSTR          buf = NULL;
    DWORD          len = 0;
    hr = PXCp_GetDocumentInfoA(hDoc, InfoField_Title, buf, &len);
    if (IS_DS_FAILED(hr) || !len)
    {
        // Handle errors
        // ...
    }
    buf = new WCHAR[len];
    hr = PXCp_GetDocumentInfoA(hDoc, InfoField_Title, buf, &len);
    if (IS_DS_FAILED(hr) || !len)
    {
        delete buf;
        // handle error
    }
    // Here we've got in the 'buf' the required information
    // ...
    delete buf;
```

3.1.2.3 PXCp_GetDocumentInfoW

PXCp_GetDocumentInfoW

Top Previous Next

PXCp_GetDocumentInfoW retrieves standard information field data regarding the structure of a pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

```
HRESULT  PXCp_GetDocumentInfoW(
    PDFDocument pDoc,
    PXC_StdInfoField field,
    LPWSTR value,
    DWORD* bufLen
);
```

Parameters

pDoc

[in] *pDoc* specifies the PDF object previously created by the function **PXCp_Init**.

field

[in] *field* specifies an information tag to retrieve. Possible values are:

Value	Meaning
InfoField_Title	Sets the **Title** field in the info structure of the pdf file.
InfoField_Subject	Sets the **Subject** field in the info structure of the pdf file.
InfoField_Author	Sets the **Author** field in the info structure of the pdf file.
InfoField_Keywords	Sets the **Keywords** field in the info structure of the pdf file.
InfoField_Creator	Sets the **Creator** field in the info structure of the pdf file.
InfoField_Producer	Sets the **Producer** field in the info structure of the pdf file.

value

[in, out] *value* specifies a pointer to a buffer where the information should be inserted.
Note: To determine the required buffer size you should pass `NULL` as *value*.
In this case the function will return **`DPro_Wrn_NeedGreaterBuffer`**.

bufLen

[in, out] *bufLen* specifies an available buffer size in characters (including a null-terminating character).
Note: When *value* is set to `NULL` then *bufLen* will contain the required buffer size

Return Values

If the function succeeds, the return value is `DS_OK`.
If the required buffer size is greater then that available *bufLen* the function returns **`DPro_Wrn_NeedGreaterBuffer`**.
If there is no information set for the document - the function returns **`DPro_Wrn_InfoNotSet`**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function is the UNICODE equivalent of ASCII function **PXCp_GetDocumentInfoA**.

Example (C++).

```
// Example on how to retrieve information tag from the document
    LPWSTR         buf = NULL;
    DWORD          len = 0;
    hr = PXCp_GetDocumentInfoW(hDoc, InfoField_Title, buf, &len);
    if (IS_DS_FAILED(hr) || !len)
    {
        // Handle errors
        // ...
    }
    buf = new WCHAR[len];
    hr = PXCp_GetDocumentInfoW(hDoc, InfoField_Title, buf, &len);
    if (IS_DS_FAILED(hr) || !len)
    {
        delete buf;
        // handle error
    }
    // Here we've got in the 'buf' the required information
    // ...
    delete buf;
```

3.1.2.4 PXCp_GetDocumentInfoExA

PXCp_GetDocumentInfoExA

Top Previous Next

PXCp_GetDocumentInfoExA retrieves all information (standard and additional field data) regarding the structure of a pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

```
HRESULT  PXCp_GetDocumentInfoExA(
        PDFDocument pDoc,
        DWORD index,
        LPSTR key,
        DWORD* keybufLen,
        LPSTR value,
        DWORD* valuebufLen
);
```

Parameters

pDoc

[in] *pDoc* specifies the PDF object previously created by the function **PXCp_Init**.

index

[in] *index* specifies an information tag index.

key

[in, out] *key* specifies a pointer to a buffer where the key name should be filled.
Note: To determine the required buffer size you should pass `NULL` as *key*.
In this case the function will return **`DPro_Wrn_NeedGreaterBuffer`**.

keybufLen

[in, out] *keybufLen* specifies an available buffer size in characters (including a null-terminating character).
Note: When *key* is set to `NULL` then *keybufLen* will contain the required buffer size.

value

[in, out] *value* specifies a pointer to a buffer where the information will be inserted.
Note: To determine the required buffer size you should pass `NULL` as *value*.
In this case the function will return **`DPro_Wrn_NeedGreaterBuffer`**.

valuebufLen

[in, out] *valuebufLen* specifies an available buffer size in characters (including a null-terminating character).
Note: When *value* is set to `NULL` then *valuebufLen* will contain the required buffer size

Return Values

If the function succeeds, the return value is `DS_OK`.
If a required buffer size is less than that passed *keybufLen* or *valuebufLen* the function returns **`DPro_Wrn_NeedGreaterBuffer`**.
If there is no information in the passed index (or greater than available) the function will return **`DPro_Wrn_InfoTagNotSet`**.
If there is no information set for the document - the function will return **`DPro_Wrn_InfoNotSet`**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function is the ASCII equivalent of UNICODE function **PXCp_GetDocumentInfoExW**.

Example (C++).

```
// Example on how to retrieve all document information
// including standard and additional tags
    hr = DS_OK;
    DWORD index = 0;
    do
    {
        LPSTR         KeyName = NULL;
        LPSTR         KeyVal = NULL;
        DWORD    lenKeyName = 0;
        DWORD    lenKeyVal = 0;
        hr = PXCp_GetDocumentInfoExA(hDoc, index, KeyName, &lenKeyName,
KeyVal, &lenKeyVal);
        if (IS_DS_FAILED(hr))
            break;
        if (lenKeyName)
        {
            KeyName = new WCHAR[lenKeyName];
        }
        if (lenKeyVal)
        {
            KeyVal = new WCHAR[lenKeyVal];
        }
        hr = PXCp_GetDocumentInfoExA(hDoc, index, KeyName, &lenKeyName,
KeyVal, &lenKeyVal);
        // Handle obtained information
        // ...
        if (KeyName) delete KeyName;
        if (KeyVal) delete KeyVal;
        index++;
    } while(IS_DS_SUCCESSFUL(hr) && (hr != DPro_Wrn_InfoTagNotSet));
```

3.1.2.5 PXCp_GetDocumentInfoExW

PXCp_GetDocumentInfoExW

Top Previous Next

PXCp_GetDocumentInfoExW retrieves all information (standard and additional field data) regarding the structure of a pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

```
HRESULT  PXCp_GetDocumentInfoExW(
    PDFDocument pDoc,
    DWORD index,
    LPWSTR key,
    DWORD* keybufLen,
    LPWSTR value,
    DWORD* valuebufLen
);
```

Parameters

pDoc

[in] *pDoc* specifies the PDF object previously created by the function **PXCp_Init**.

index

[in] *index* specifies an information tag index.

key

[in, out] *key* specifies a pointer to a buffer where the key name should be filled.
Note: To determine the required buffer size you should pass `NULL` as *key*.
In this case the function will return **`DPro_Wrn_NeedGreaterBuffer`**.

keybufLen

[in, out] *keybufLen* specifies an available buffer size in characters (including a null-terminating character).
Note: When *key* is set to `NULL` then *keybufLen* will contain the required buffer size.

value

[in, out] *value* specifies a pointer to a buffer where the information will be inserted.
Note: To determine the required buffer size you should pass `NULL` as *value*.
In this case the function will return **`DPro_Wrn_NeedGreaterBuffer`**.

valuebufLen

[in, out] *valuebufLen* specifies an available buffer size in characters (including a null-terminating character).
Note: When *value* is set to `NULL` then *valuebufLen* will contain the required buffer size

Return Values

If the function succeeds, the return value is `DS_OK`.
If a required buffer size is less than that passed *keybufLen* or *valuebufLen* the function returns **`DPro_Wrn_NeedGreaterBuffer`**.
If there is no information in the passed index (or greater than available) the function will return **`DPro_Wrn_InfoTagNotSet`**.
If there is no information set for the document - the function will return **`DPro_Wrn_InfoNotSet`**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function is the UNICODE equivalent of ASCII function **PXCp_GetDocumentInfoExA**.

Example (C++).

```
// Example on how to retrieve all document information
// including standard and additional tags
    hr = DS_OK;
    DWORD index = 0;
    do
    {
        LPWSTR         KeyName = NULL;
        LPWSTR         KeyVal = NULL;
        DWORD   lenKeyName = 0;
        DWORD   lenKeyVal = 0;
        hr = PXCp_GetDocumentInfoExW(hDoc, index, KeyName, &lenKeyName,
KeyVal, &lenKeyVal);
        if (IS_DS_FAILED(hr))
            break;
        if (lenKeyName)
        {
            KeyName = new WCHAR[lenKeyName];
        }
        if (lenKeyVal)
        {
            KeyVal = new WCHAR[lenKeyVal];
        }
        hr = PXCp_GetDocumentInfoExW(hDoc, index, KeyName, &lenKeyName,
KeyVal, &lenKeyVal);
        // Handle obtained information
        // ...
        if (KeyName) delete KeyName;
        if (KeyVal) delete KeyVal;
        index++;
    } while(IS_DS_SUCCESSFUL(hr) && (hr != DPro_Wrn_InfoTagNotSet));
```

3.1.2.6 PXCp_GetPageLayout

PXCp_GetPageLayout

Top Previous Next

PXCp_GetPageLayout function retrieves the applied document page layout mode settings.

```
HRESULT PXCp_GetPageLayout(
    PDFDocument pDocument,
    PXC_PageLayout* playout
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

playout

[out] *playout* specifies a pointer to a variable for the `PXC_PageLayout` type which receives the page layout mode. Possible values are described in **PXCp_SetPageLayout** function.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
  PDFDocument        hDocument;
   // Get information about current page layout:
   PXC_PageLayout         PageLayout;
   HRESULT res = PXCp_GetPageLayout(hDocument, &PageLayout);
   if (IS_DS_FAILED(res))
```

```
    {
        // Report an error
    }
    switch (PageLayout)
    {
        // display information about page layout
        // ...
    }
    ...
    // Clean up
    PXCp_Delete(hDocument);
```

3.1.2.7 PXCp_GetPageMode

PXCp_GetPageMode

Top Previous Next

PXCp_GetPageMode retrieves the applied page mode settings for the document.

```
HRESULT  PXCp_GetPageMode(
    PDFDocument pDocument,
    PXC_PageMode* pmode
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

pmode
[in] *pmode* specifies a pointer to a variable containing the `PXC_PageMode` type value.
For possible values refer to **PXCp_SetPageMode**.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
  PDFDocument        hDocument;
   // Get information about the current page mode:
   PXC_PageMode        PageMode;
   HRESULT res = PXCp_GetPageMode(&hDocument, &PageMode);
   if (IS_DS_FAILED(res))
   {
       // Report an error
   }
   switch (PageMode)
   {
       // display information about page mode
       // ...
```

```
    }
    ...
    // Clean up
    PXCp_Delete(hDocument);
```

3.1.2.8 PXCp_GetSpecVersion

PXCp_GetSpecVersion

Top Previous Next

PXCp_GetSpecVersion retrieves the version of the Adobe PDF Format currently applicable to the document.

```
HRESULT  PXCp_GetSpecVersion(
    PDFDocument pDocument,
    PXC_SpecVersion* pver
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

pver

[out] *pver* is a pointer to a variable of the `PXC_SpecVersion` type (for details refer to **PDF-XChange Library** help).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
  PDFDocument        hDocument;
   PXC_SpecVersion SpecVersion;
   // Retrieve documents version:
   HRESULT res = PXCp_GetSpecVersion(hDocument, &SpecVersion);
   if (IS_DS_FAILED(res))
   {
       // Report an error
   }
   ...
   // Clean up
   PXCp_Delete(hDocument);
```

3.1.2.9 PXCp_GetViewerPreferences

PXCp_GetViewerPreferences

Top Previous Next

PXCp_GetViewerPreferences retrieves the applied viewer preferences of a document.

```
HRESULT  PXCp_GetViewerPreferences(
    PDFDocument pDocument,
    DWORD* pvprefs
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

pvprefs

[out] specifies a pointer to a variable of the DWORD type which recieves the viewer preferences. This bit field is a combination of the flags described in **PXCp_SetViewerPreferences**

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
PDFDocument        hDocument;
// Get information relating to current viewer preference:
DWORD   ViewerPreferences = 0;
HRESULT res = PXCp_GetViewerPreferences(&hDocument, &ViewerPreferences);
if (IS_DS_FAILED(res))
{
    // Report an error
}
// display information about current viewer preference
// i.e. check if FitWindow flag is set:
if (ViewerPreferences & VP_FitWindow)
{
    // ...
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.2.10 PXCp_SetDocumentInfoA

PXCp_SetDocumentInfoA

Top Previous Next

PXCp_SetDocumentInfoA stores standard information field data regarding the structure of a pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

```
HRESULT  PXCp_SetDocumentInfoA(
    PDFDocument pDocument,
    PXC_StdInfoField field,
    LPCSTR KeyVal
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

field

[in] *field* specifies information tag to be set. Possible values are:

Value	Meaning
InfoField_Title	Sets the **Title** field in the info structure of the pdf file.
InfoField_Subject	Sets the **Subject** field in the info structure of the pdf file.
InfoField_Author	Sets the **Author** field in the info structure of the pdf file.
InfoField_Keywords	Sets the **Keywords** field in the info structure of the pdf file.
InfoField_Creator	Sets the **Creator** field in the info structure of the pdf file.
InfoField_Producer	Sets the **Producer** field in the info structure of the pdf file.

KeyVal

[in] Pointer to a null-terminated ASCII string that specifies the value for the field *field*.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function is the ASCII equivalent of UNICODE function **PXCp_SetDocumentInfoW**.

Example (C++).

```
PDFDocument        hDocument;
LPCSTR        DocTitle = "This is the new document title";
// Set new title:
HRESULT res = PXCp_SetDocumentInfoA(hDocument, InfoField_Title, DocTitle);
if (IS_DS_FAILED(res))
{
    // Report an error
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.2.11 PXCp_SetDocumentInfoW

PXCp_SetDocumentInfoW

Top Previous Next

PXCp_SetDocumentInfoW stores standard information field data regarding the structure of a pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

```
HRESULT  PXCp_SetDocumentInfoW(
    PDFDocument pDocument,
    PXC_StdInfoField field,
    LPCWSTR KeyVal
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

field

[in] *field* specifies information tag to be set. Possible values are:

Value	**Meaning**
InfoField_Title	Sets the **Title** field in the info structure of the pdf file.
InfoField_Subject	Sets the **Subject** field in the info structure of the pdf file.
InfoField_Author	Sets the **Author** field in the info structure of the pdf file.
InfoField_Keywords	Sets the **Keywords** field in the info structure of the pdf file.
InfoField_Creator	Sets the **Creator** field in the info structure of the pdf file.
InfoField_Producer	Sets the **Producer** field in the info structure of the pdf file.

KeyVal

[in] Pointer to a null-terminated UNICODE string that specifies the value for the field *field*.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function is the UNICODE equivalent of ASCII function **PXCp_SetDocumentInfoA**.

Example (C++).

```
PDFDocument        hDocument;
LPCWSTR     DocTitle = L"This is the new document title";
// Set new title:
HRESULT res = PXCp_SetDocumentInfoW(hDocument, InfoField_Title, DocTitle);
if (IS_DS_FAILED(res))
{
     // Report an error
}
...
// Clean up
```

```
PXCp_Delete(hDocument);
```

3.1.2.12 PXCp_SetDocumentInfoExA

PXCp_SetDocumentInfoExA

Top Previous Next

PXCp_SetDocumentInfoExA stores additional information to the structure of a pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

```
HRESULT  PXCp_SetDocumentInfoExA(
    PDFDocument pDocument,
    LPCSTR KeyName,
    LPCSTR KeyVal
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

KeyName

[in] *KeyName* is a pointer to a null-terminated UNICODE string that specifies the value for the `Key` of the field.

KeyVal

[in] *KeyVal* is a pointer to a null-terminated UNICODE string that specifies the value for the field.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

PXCp_SetDocumentInfoExA may be used to set standard informational tags as well.

This function is the ASCII equivalent of UNICODE function **PXCp_SetDocumentInfoExW**.

Example (C++).

```
PDFDocument        hDocument;
LPCSTR     MyTagName  = "CoolTagName";
LPCSTR     MyTagValue = "Cool tag value";
// Set my tag:
HRESULT res = PXCp_SetDocumentInfoExA(hDocument, MyTagName, MyTagValue);
if (IS_DS_FAILED(res))
{
    // Report an error
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.2.13 PXCp_SetDocumentInfoExW

PXCp_SetDocumentInfoExW

Top Previous Next

PXCp_SetDocumentInfoExW stores additional information to the structure of a pdf object (for example when using a mouse in Windows Explorer and selecting a file - this information becomes viewable when you 'right click' and select the 'Properties' option).

```
HRESULT  PXCp_SetDocumentInfoExW(
    PDFDocument pDocument,
    LPCWSTR KeyName,
    LPCWSTR KeyVal
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

KeyName

[in] *KeyName* is a pointer to a null-terminated UNICODE string that specifies the value for the `Key` of the field.

KeyVal

[in] *KeyVal* is a pointer to a null-terminated UNICODE string that specifies the value for the field.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

PXCp_SetDocumentInfoExW may be used to set standard informational tags as well.

This function is the UNICODE equivalent of ASCII function **PXCp_SetDocumentInfoExA**.

Example (C++).

```
PDFDocument        hDocument;
LPCWSTR     MyTagName  = L"CoolTagName";
LPCWSTR     MyTagValue = L"Cool tag value";
// Set my tag:
HRESULT res = PXCp_SetDocumentInfoExW(hDocument, MyTagName, MyTagValue);
if (IS_DS_FAILED(res))
{
    // Report an error
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.2.14 PXCp_SetPageLayout

PXCp_SetPageLayout

Top Previous Next

PXCp_SetPageLayout sets the desired page layout mode.

```
HRESULT  PXCp_SetPageLayout(
    PDFDocument pDocument,
    PXC_PageLayout layout
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

layout

[in] *layout* specifies the page layout mode. Possible values are:

Constant	Value	Meaning
PageLayout_SinglePage	0	Display one page at a time.
PageLayout_OneColumn	1	Display the pages in one column.
PageLayout_TwoColumns_Left	2	Display the pages in two columns, with odd numbered pages on the left.
PageLayout_TwoColumns_Right	3	Display the pages in two columns, with odd numbered pages on the right.

For details refer to **PDF-XChange Library** help.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
PDFDocument        hDocument = NULL;
// display one page at a time:
HRESULT res = PXCp_SetPageLayout(&hDocument, PageLayout_SinglePage);
if (IS_DS_FAILED(res))
{
    // Report any error
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.2.15 PXCp_SetPageMode

PXCp_SetPageMode

Top Previous Next

The **PXCp_SetPageMode** sets the desired page mode for the current document.

```
HRESULT  PXCp_SetPageMode(
    PDFDocument pDocument,
    PXC_PageMode mode
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

mode

[in] Specifies page mode. Possible values are:

Constant	Value	Meaning
PageMode_None	0	Neither document outline nor thumbnail images visible.
PageMode_Outlines	1	Document outline visible.
PageMode_Thumbnails	2	Thumbnail images visible.
PageMode_FullScreen	3	Full-screen mode, with no menu bar, window controls, or any other window visible.

For details refer to **PDF-XChange Library** help.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
PDFDocument        hDocument = NULL;
// Set Thumbnail mode:
HRESULT res = PXCp_SetPageMode(&hDocument, PageMode_Thumbnails);
if (IS_DS_FAILED(res))
{
    // Report an error
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.2.16 PXCp_SetViewerPreferences

PXCp_SetViewerPreferences

Top Previous Next

PXCp_SetViewerPreferences applies the required viewer preferences.

```
HRESULT  PXCp_SetViewerPreferences(
    PDFDocument pDocument,
    DWORD vprefs
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

vprefs

[in] Specifies the viewer preferences. A bit field with a combination of the following flags:

Constant	Value	Meaning
VP_HideToolbar	0x0001	A flag specifying whether to hide the viewer application tool bars when the document is active.
VP_HideMenubar	0x0002	A flag specifying whether to hide the viewer application menu bar when the document is active.
VP_HideWindowUI	0x0004	A flag specifying whether to hide user interface elements in the document window (such as scroll bars and navigation controls), leaving only the document contents displayed.
VP_FitWindow	0x0008	A flag specifying whether to resize the document window to fit the size of the first displayed page.
VP_CenterWindow	0x0010	A flag specifying whether to position the document window to the center of the screen.
VP_DisplayDocTitle	0x0020	A flag specifying whether the window's title bar should display the document title taken from the Title field of the document information (see **PXCp_SetDocumentInfoW**). If this flag is not set, the title bar will display the name of the PDF file containing the document. **Note:** Valid only if **PDF specification** is 1.4 or greater.
VP_Direction_R2L	0x0040	A flag specifying the predominant reading order for text: Left-to-Right, when flag not set, or Right-to-Left (including vertical writing systems such as Chinese, Japanese, and Korean), when set. This flag has no direct effect on the document contents or page numbering, but can be used to determine the relative positioning of pages when displayed side by side or printed *n*-up.

Additionally one of the following flags may be used to determine the document *page mode*, specifying how to display the document on exiting *full-screen mode*:

Constant	*Value*	*Meaning*
VP_FSPM_None	0x0000	Neither document outline nor thumbnail images visible.
VP_FSPM_Outlines	0x0100	Document outline visible.
VP_FSPM_Tumbnails	0x0200	Thumbnail images visible.
VP_FSPM_OC	0x0400	Optional content group panel visible.

Note: Flags are meaningful only if the **PageMode** of the document is `PageMode_FullScreen`; and is ignored otherwise.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
 PDFDocument        hDocument = NULL;
  DWORD   flags = 0;
  // Let's construct the desired options for the document:
  // 1. Hide toolbar and menubar:
  flags = VP_HideToolbar | VP_HideMenubar
  // 2. Show bookmarks in the left panel:
  flags |= VP_FSPM_Outlines;
  // Set this options:
  HRESULT res = PXCp_SetViewerPreferences(&hDocument, flags);
  if (IS_DS_FAILED(res))
  {
      // Report any error
  }
  ...
  // Clean up
  PXCp_Delete(hDocument);
```

3.1.3 Document Optimization

3.1.3.1 PXCp_OptimizeFonts

PXCp_OptimizeFonts

Top Previous Next

PXCp_OptimizeFonts optimizes fonts within a PDF document and removes font duplication. Additionally attempts to merge similar fonts into a single font entry where possible.

N.B. We strongly recommend applying this function after merging two or more files or copying or adding pages to an existing file etc.

```
HRESULT  PXCp_OptimizeFonts(
    PDFDocument pDoc,
    DWORD Flags
);
```

Parameters

pDoc

[in] *pDoc* specifies the PDF object previously created by the function **PXCp_Init**.

Flags

[in] *Flags* reserved for future use. Must be set to `0`.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
 PDFDocument  hDocument;
...
// Optimize fonts in the document

HRESULT res = PXCp_OptimizeFonts(hDocument, 0);
if (IS_DS_FAILED(res))
{
    // Report an error
    ...
}

// Now, after saving the document it's size will be reduced where possible
// if the document contained unoptimised fonts
// i.e. after merging etc
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.3.2 PXCp_OptimizeRemoveNamedDests

PXCp_OptimizeRemoveNamedDests

Top Previous Next

PXCp_OptimizeRemoveNamedDests optimizes a PDF document by removing all named objects.

Named objects/destinations - are unnecessary, but commonly used for convenience by some PDF Libraries/ drivers etc. This often causes bloated and enlarged file sizes. This function safely removes such objects from a file and reduces size.

```
HRESULT  PXCp_OptimizeRemoveNamedDests(
    PDFDocument pDoc
);
```

Parameters

pDoc

[in] *pDoc* specifies the PDF object previously created by the function **PXCp_Init**.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes

page.

Remarks

In the event the file does not contain named objects/destinations, this function offers no benefits, but causes no harm.

Example (C++).

```
        PDFDocument  hDocument;
    ...
    // Remove named objects from the document

    HRESULT res = PXCp_OptimizeRemoveNamedDests(hDocument);
    if (IS_DS_FAILED(res))
    {
        // Report an error
        ...
    }

    // Now, after saving the document it's size will be reduced if possible.
    ...
    // Clean up
    PXCp_Delete(hDocument);
```

3.1.3.3 PXCp_OptimizeStreamCompression

PXCp_OptimizeStreamCompression

Top Previous Next

PXCp_OptimizeStreamCompression optimizes the compression applied in all streams (the binary data stored within a PDF document). This operation may reduce file size significantly on files where such optimization has not previously been applied.

The Function compresses streams which are not compressed and checks the existing compression for streams and applies a more efficient compression where possible.

```
HRESULT  PXCp_OptimizeStreamCompression(
    PDFDocument pDoc
);
```

Parameters

pDoc

[in] *pDoc* specifies the PDF object previously created by the function **PXCp_Init**.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

N.B. For best results, this function should be performed **after** **PXCp_OptimizeRemoveNamedDests** and **PXCp_OptimizeFonts**functions as a final action.

Example (C++).

```
     PDFDocument  hDocument;
...
// Optimize compression of the streams in the document

HRESULT res = PXCp_OptimizeStreamCompression(hDocument);
if (IS_DS_FAILED(res))
{
    // Report an error
    ...
}

// Now, after saving the document it's size will be reduced
// in case the streams were not compressed optimally.
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.4 Page Information

3.1.4.1 PXCp_GetPagesCount

PXCp_GetPagesCount

Top Previous Next

PXCp_GetPagesCount retrieves the page count from within a PDF document.

```
HRESULT  PXCp_GetPagesCount(
    PDFDocument pDocument,
    DWORD* count
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

count
[out] *count* specifies a pointer to a variable of the DWORD type to receive the page count.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
    PDFDocument  hDocument;
```

```
DWORD        PageNumber = 0;
// Get number of pages in the document:
HRESULT res = PXCp_GetPagesCount(hDocument, &PageNumber);
if (IS_DS_FAILED(res))
{
    // Report an error
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.4.2 PXCp_PageGetBox

PXCp_PageGetBox

Top Previous Next

PXCp_PageGetBox retrieves the specified page boundaries rectangle.

For more information regarding page boundaries see **PXCp_PageSetBox** function.

```
HRESULT  PXCp_PageGetBox(
    PDFDocument pDocument,
    DWORD PageNumber,
    PXC_PageBox pBoxID,
    LPPXC_RectF rect
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] *PageNumber* specifies a page number within the document.

pBoxID
[in] *pBoxID* specifies the ID of the page's rectangle, and may be any one of the following values.

Constant	Value	Meaning
PB_MediaBox	0	Specifies *media box* to be retrieved.
PB_CropBox	1	Specifies *crop box* to be retrieved.
PB_BleedBox	2	Specifies *bleed box* to be retrieved.
PB_TrimBox	3	Specifies *trim box* to be retrieved.
PB_ArtBox	4	Specifies *art box* to be retrieved.

rect
[out] Pointer to a `PXC_RectF` structure from **PDF-XChange Library** that should receive the coordinates of the specified page's box.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes

page.

Example (C++).

```
PDFDocument  hDocument;
PXC_RectF    MediaBox = {0};
// Retrieve the width and height of the first page in the document:
HRESULT res = PXCp_PageGetBox(hDocument, 0, PB_MediaBox, &PageRect);
if (IS_DS_FAILED(res))
{
    // Report an error
}
double width  = MediaBox.right - MediaBox.left;
double height = MediaBox.bottom - MediaBox.top;
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.4.3 PXCp_PageGetRotate

PXCp_PageGetRotate

Top Previous Next

PXCp_PageGetRotate retrieve's a page's rotation angle.

```
HRESULT  PXCp_PageGetRotate(
    PDFDocument pDocument,
    DWORD PageNumber,
    LONG* pangle
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] *PageNumber* specifies a page number within the document.

pangle
[out] Specifies a pointer to a variable which receives the rotation angle of the page.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
   PDFDocument  hDocument;
   LONG         Angle = 0;
   // Retrieve the the original rotation angle for the first page in the
document:
   HRESULT res = PXCp_PageGetRotate(hDocument, 0, &Angle);
```

```
    if (IS_DS_FAILED(res))
    {
        // Report an error
    }
    ...
    // Clean up
    PXCp_Delete(hDocument);
```

3.1.4.4 PXCp_PageSetBox

PXCp_PageSetBox

Top Previous Next

PXCp_PageSetBox allows the specification of different bounding rectangles for a PDF page.

For more detail see **Comments**.

```
HRESULT  PXCp_PageSetBox(
    PDFDocument pDocument,
    DWORD PageNumber,
    PXC_PageBox pBoxID,
    LPCPXC_RectF rect
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] *PageNumber* specifies a page number within the document.

pBoxID

[in] *pBoxID* specifies the ID of the page's rectangle, and may be any one of the following values.

Constant	Value	Meaning
PB_MediaBox	0	Specifies *media box* to be set.
PB_CropBox	1	Specifies *crop box* to be set.
PB_BleedBox	2	Specifies *bleed box* to be set.
PB_TrimBox	3	Specifies *trim box* to be set.
PB_ArtBox	4	Specifies *art box* to be set.

rect

[in] Pointer to a `PXC_RectF` structure from **PDF-XChange Library** that contains the coordinates of the specified page's box.
Refer to **PDF-XChange Library** help.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

A PDF page may be prepared either for a finished medium, such as a sheet of paper, or as part of a prepress process in which the content of the page is placed on an intermediate medium, such as film or an imposed reproduction plate. In the latter case, it is important to distinguish between the intermediate page and the finished page. The intermediate page may often include additional production-related content, such as bleeds or printer marks, that falls outside the boundaries of the finished page. To handle such cases, a PDF page can define as many as five separate boundaries to control various aspects of the imaging process:

- The *media box* defines the boundaries of the physical medium on which the page is to be printed. It may include any extended area surrounding the finished page for bleed, printing marks, or other such purposes. It may also include areas close to the edges of the medium that cannot be marked because of physical limitations of the output device. Content falling outside this boundary can safely be discarded without affecting the meaning of the PDF file.
- The crop box defines the region to which the contents of the page are to be clipped (cropped) when displayed or printed. Unlike the other boxes, the crop box has no defined meaning in terms of physical page geometry or intended use; it merely imposes clipping on the page contents. However, in the absence of additional information, the crop box will determine how the page's contents are to be positioned on the output medium. The default value is the page's media box.
- The bleed box (PDF Spec >= 1.3) defines the region to which the contents of the page should be clipped when output in a production environment. This may include any extra "bleed area" needed to accommodate the physical limitations of cutting, folding, and trimming equipment. The actual printed page may include printing marks that fall outside the bleed box. The default value is the page's crop box.
- The trim box (PDF Spec >= 1.3) defines the intended dimensions of the finished page after trimming. It may be smaller than the media box, to allow for productionrelated content such as printing instructions, cut marks, or color bars. The default value is the page's crop box.
- The art box (PDF Spec >= 1.3) defines the extent of the page's meaningful content (including potential white space) as intended by the page's creator. The default value is the page's crop box.

Example (C++).

```
PDFDocument  hDocument;
PXC_RectF    MediaBox = {0};
// Retrieve the media box of the first page in the document:
HRESULT res = PXCp_PageGetBox(hDocument, 0, PB_MediaBox, &PageRect);
if (IS_DS_FAILED(res))
{
    // Report any error
}
// Now set the crop box with half the width of the original media box
double width  = MediaBox.right - MediaBox.left;
MediaBox.right = MediaBox.left + width / 2;
res = PXCp_PageSetBox(hDocument, 0, PB_CropBox, &PageRect);
if (IS_DS_FAILED(res))
{
    // Report an error
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.4.5 PXCp_PageSetRotate

PXCp_PageSetRotate

Top Previous Next

PXCp_PageSetRotate rotates a page by the specified angle.

```
HRESULT  PXCp_PageSetRotate(
    PDFDocument pDocument,
    DWORD PageNumber,
    LONG angle
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] *PageNumber* specifies a page number within the document.

angle
[in] *angle* specifies a rotation angle to be applied. Possible values are: `0, 90, 180, 270, -90, -180, -270`.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
   PDFDocument  hDocument;
   LONG         Angle = 0;
   // Retrieve the the original rotation angle for the first page in the
document:
   HRESULT res = PXCp_PageGetRotate(hDocument, 0, &Angle);
   if (IS_DS_FAILED(res))
   {
       // Report any error
   }
   // Now rotate the page 90 degree's clockwise:
   Angle += 90;
   // Check if the angle is not greater then 360 degree:
   if (Angle >= 360) Angle -= 360;
   // Set new rotation angle:
   res = PXCp_PageSetRotate(hDocument, 0, Angle);
   ...
   // Clean up
   PXCp_Delete(hDocument);
```

3.1.5 Bookmarks

3.1.5.1 PXCp_BMDeleteAllItems

PXCp_BMDeleteAllItems

Top Previous Next

PXCp_BMDeleteAllItems deletes all items in a PDF document. If the document is saved - all bookmark items are permanently removed.

```
HRESULT  PXCp_BMDeleteAllItems(
    PDFDocument pDocument
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
  // Delete all bookmark items in the document:
   PDFDocument    hDocument;
   HRESULT res = PXCp_BMDeleteAllItems(hDocument);
   if (IS_DS_FAILED(res))
   {
       // Report an error
   }
   // Now the document contains no items
   ...
   // Clean up
   PXCp_Delete(hDocument);
```

3.1.5.2 PXCp_BMDeleteItem

PXCp_BMDeleteItem

Top Previous Next

PXCp_BMDeleteItem deletes a specified item and all child entries.

```
HRESULT  PXCp_BMDeleteItem(
    PDFDocument pDocument,
    PXCp_BMHandle bmItem
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

bmItem
[in] *bmItem* specifies an item to delete.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Delete the bookmark item from the document:
 PDFDocument    hDocument;
 PXCp_BMHandle  hBMItem;
 HRESULT res = PXCp_BMDeleteItem(hDocument, hBMItem);
 if (IS_DS_FAILED(res))
 {
     // Report an error
 }
 // Now the item and all it's child entries are removed from the document
 ...
 // Clean up
 PXCp_Delete(hDocument);
```

3.1.5.3 PXCp_BMGetItem

PXCp_BMGetItem

Top Previous Next

PXCp_BMGetItem retrieves a bookmark item handle by identifying it's relative position to another bookmark item.

```
HRESULT  PXCp_BMGetItem(
    PDFDocument pDocument,
    PXCp_BMHandle bmItem,
    PXCp_OutlinePos itemPos,
    PXCp_BMHandle* pbmItem
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

bmItem
[in] *bmItem* specifies a handle to the item, relative to which the item will be returned.

Note: *bmItem*may be set to `NULL` only when *itemPos* is set to `PBM_ROOT` - in this case the root outline item is retrieved and the function call is equivalent to the **PXCp_GetRootBMItem** function.

itemPos
[in] *itemPos* specifies the relative position of the retrieved item.
May be one of the following:

Value	Meaning
PBM_FIRST	Retrieve the first child item.
PBM_LAST	Retrieve the last child item.
PBM_CHILD	Has the same meaning as the previous
PBM_NEXT	Retrieve the next sibling item.
PBM_PARENT	Retrieve the parent of the specified item.
PBM_PREVIOUS	Retrieve the previous sibling item.

pbmItem

[out] *pbmItem* specifies a pointer to a `PXCp_BMHandle` for the retrieved item.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the specified item is absent then the function returns **`DPro_Err_BMItemNotPresent`**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
  // Example shows, how to retrieve an item relative to its parent, child, etc
   // This example walks through the whole tree, and changes the color of each
item

   // This function changes the item color
   // And after that calls itself for the next and first child item
   // In this recursive way all the bookmarks in the tree will change there
color

   void OutlinesChangecolor(PXCp_BMHandle root, COLORREF& color, DWORD growBy)
   {
     // Check if this is valid handle?
     if (!root) return;

     PXCp_BMInfo  bmInfo;

     // Change the current color - all items will have different colors
     color += growBy;

     // Set the color
     bmInfo.Color  = color;

     // Set the open status - all items will be opened in the tree
     bmInfo.bOpen  = TRUE;

     bmInfo.cbSize  = sizeof(PXCp_BMInfo);
     // This is what we chnage for the item
     bmInfo.Mask = BMIM_Color | BMIM_Open;

     // Set new properties to the item
     PXCp_BMSetItemInfo(hDoc, root, &bmInfo);
```

```
      PXCp_BMHandle  hChild = NULL;

      // Try to get first child
      HRESULT hr = PXCp_BMGetItem(hDoc, root, PBM_CHILD, &hChild);

      // If it is present then change its color too
      if (hChild)
        OutlinesChangecolor(hChild, color, growBy);

      PXCp_BMHandle hNext = NULL;

      // Look for the next item in the hierarhy
      hr = PXCp_BMGetItem(hDoc, root, PBM_NEXT, &hNext);

      // If it is present then change its color too
      if (hNext)
        OutlinesChangecolor(hNext, color, growBy);

      return;
    }

    // And this is the 'main' function which starts the process

    void DoIt(PDFDOcument hDoc)
    {
        PXCp_BMHandle  bmRoot;
        HRESULT hr = PXCp_GetRootBMItem(hDoc, &bmRoot);
        if (IS_DS_FAILED(hr))
        {
            // Fail to retrive the root item
            return;
        }

        // Change colors of all bookmarks
        COLORREF  startColor = RGB(0, 0, 0);
        OutlinesChangecolor(bmRoot, startColor, RGB(1, 2, -1));
    }
```

3.1.5.4 PXCp_BMGetItem

PXCp_BMGetItem

PXCp_BMGetItemInfo retrieves the specified information about the specified bookmark item.

```
HRESULT  PXCp_BMGetItemInfo(
    PDFDocument pDocument,
    PXCp_BMHandle bmItem,
    LPPXCp_BMInfo pbmItemInfo
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

bmItem

[in] *bmItem* specifies a bookmark item handle.

pbmItemInfo

[out] *pbmItemInfo* specifies a pointer to the **PXCp_BMInfo** structure which receives information about the item.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve the title and the color of the bookmark item:
 PDFDocument    hDocument;
 PXCp_BMHandle  hBMItem;
 // The next structure will containe the necessary information:
 PXCp_BMInfo bmItemInfo = {0};
 bmItemInfo.cbSize = sizeof(PXCp_BMInfo);
 // We like to get just title and color of the item:
 bmItemInfo.Mask = BMIM_TitleW | BMIM_Color;
 HRESULT res = PXCp_BMGetItemInfo(hDocument, hBMItem, &bmItemInfo);
 if (IS_DS_FAILED(res))
 {
     // Report an error
 }
 if (bmItemInfo.LengthOfTitle)
 {
     bmItemInfo.TitleW = new WCHAR[bmItemInfo.LengthOfTitle];
     hr = PXCp_BMGetItemInfo(hDoc, bmHandle, &bmItemInfo);
     if (IS_DS_FAILED(hr))
     {
       // problem getting title...
       return;
     }
    // Now the title is stored in 'bmItemInfo.TitleW'
    // ....
    delete[] bmItemInfo.TitleW;
  }

 ...
 // Clean up
 PXCp_Delete(hDocument);
```

3.1.5.5 PXCp_BMInsertItem

PXCp_BMInsertItem

Top Previous Next

PXCp_BMInsertItem creates and inserts a specified item into the outline tree at the required position.

```
HRESULT  PXCp_BMInsertItem(
    PDFDocument pDocument,
    PXCp_BMHandle bmParent,
    PXCp_BMHandle bmInsertAfter,
    PXCp_BMHandle* bmItem,
    PXCp_BMInfo* pItemInfo
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

bmParent

[in] *bmParent* specifies the handle of the parent item.
If this item is set to `NULL` then *bmInsertAfter* should be designated by the `PXCp_OutlinePos` constant:

Value	Meaning
PBM_FIRST	Item will be set as the first at the top level.
PBM_LAST	Item will be set as the last at the top level.
PBM_ROOT	Item will be set as the root/parent outline for all existing outline items. All other top level items will become children to this item

bmInsertAfter

[in] *bmInsertAfter* specifies the handle of the item after which to insert.
If *bmParent* is not set to `NULL` then *bmInsertAfter*could be set to one the next constants:

Value	Meaning
PBM_FIRST	Item will be set as the first child of the *bmParent*.
PBM_LAST	Item will be set as the last child of the *bmParent*.

bmItem

[out] *bmItem* specifies the handle for the inserted item.

pItemInfo

[in] *pItemInfo* specifies information about the item to be inserted.
For details see **PXCp_BMGetItemInfo** function description.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
  // Example removes all bookmarks and insert new item(s):
   PDFDocument    hDocument;
   // Remove all bookmarks:
```

```
   PXCp_BMDeleteAllItems(hDoc);

   // Prepare the data for the new bookmarks:
   PXCp_BMInfo        bmInfo;
   ::ZeroMemory(&bmInfo, sizeof(PXCp_BMInfo));
   bmInfo.cbSize = sizeof(PXCp_BMInfo);
   bmInfo.Destination.Mask = 0;
   bmInfo.Destination.DestType = Dest_FitH;
   bmInfo.Destination.Top = 0.10;
   bmInfo.Destination.Left = -100.0;
   bmInfo.Destination.Zoom = 0.0;
   bmInfo.Destination.PageNumber = 0;

   bmInfo.Mask = BMIM_Destination | BMIM_Color | BMIM_Open | BMIM_TitleW |
BMIM_Style;
   bmInfo.Color        = RGB(200, 0, 0);
   bmInfo.bOpen        = TRUE;
   bmInfo.TitleW        = L"PXCp created";
   bmInfo.Style        = OutlineStyle_BoldItalic;

   PXCp_BMHandle        hBM = NULL;
   // This one will be the root one:
   hr = PXCp_BMInsertItem(hDoc, NULL, (PXCp_BMHandle)PBM_ROOT, &hBM, &bmInfo);
   PXCp_BMHandle        hBMChild = NULL;
   // This one will be the the fist child of the root one:
   PXCp_BMInsertItem(hDoc, hBM, (PXCp_BMHandle)PBM_FIRST, &hBMChild, &bmInfo);
   PXCp_BMHandle        hBMLastRoot = NULL;
   bmInfo.TitleW = L"Must be last root";
   // This one will be the the last child of the root one:
   PXCp_BMInsertItem(hDoc, NULL, (PXCp_BMHandle)PBM_LAST, &hBMLastRoot,
&bmInfo);
   bmInfo.TitleW = L"Must be parent to all old roots";
   PXCp_BMHandle        hBMRootest = NULL;
   // This one will be the the new root and all other will be it's children:
   PXCp_BMInsertItem(hDoc, NULL, (PXCp_BMHandle)PBM_ROOT, &hBMRootest,
&bmInfo);
   ...
   // Clean up
   PXCp_Delete(hDocument);
```

3.1.5.6 PXCp_BMMoveItem

PXCp_BMMoveItem

Top Previous Next

PXCp_BMMoveItem moves the specified item to a new position in the outlines tree.

```
HRESULT  PXCp_BMMoveItem(
    PDFDocument pDocument,
    PXCp_BMHandle bmItem,
    PXCp_BMHandle bmParent,
    PXCp_BMHandle bmInsertAfter
);
```

Parameters

pDocument
: [in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

bmItem
: [in] *bmItem* specifies an item to be moved.

bmParent
: [in] *bmParent* specifies the new item parent.

bmInsertAfter
: [in] *bmInsertAfter* specifies the item before which to insert the required entry.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

Possible values of *bmItem* and *bmParent* are the same as in the **PXCp_BMInsertItem** function.

Example (C++).

```
  // Example removes all bookmarks and insert new items and after that move
the item:
   PDFDocument    hDocument;

   // Remove all bookmarks:
   PXCp_BMDeleteAllItems(hDoc);

   // Prepare the data for the new bookmarks:
   PXCp_BMInfo        bmInfo;
   ::ZeroMemory(&bmInfo, sizeof(PXCp_BMInfo));
   bmInfo.cbSize = sizeof(PXCp_BMInfo);
   bmInfo.Destination.Mask = 0;
   bmInfo.Destination.DestType = Dest_FitH;
   bmInfo.Destination.Top = 0.10;
   bmInfo.Destination.Left = -100.0;
   bmInfo.Destination.Zoom = 0.0;
   bmInfo.Destination.PageNumber = 0;

   bmInfo.Mask = BMIM_Destination | BMIM_Color | BMIM_Open | BMIM_TitleW |
BMIM_Style;
   bmInfo.Color         = RGB(200, 0, 0);
   bmInfo.bOpen         = TRUE;
   bmInfo.TitleW         = L"PXCp created";
   bmInfo.Style         = OutlineStyle_BoldItalic;

   PXCp_BMHandle         hBM = NULL;
   // This one will be the root one:
   hr = PXCp_BMInsertItem(hDoc, NULL, (PXCp_BMHandle)PBM_ROOT, &hBM, &bmInfo);
   PXCp_BMHandle        hBMChild = NULL;
   // This one will be the the fist child of the root one:
   PXCp_BMInsertItem(hDoc, hBM, (PXCp_BMHandle)PBM_FIRST, &hBMChild, &bmInfo);
   PXCp_BMHandle        hBMLastRoot = NULL;
   bmInfo.TitleW = L"Must be last root";
   // This one will be the the last child of the root one:
   PXCp_BMInsertItem(hDoc, NULL, (PXCp_BMHandle)PBM_LAST, &hBMLastRoot,
&bmInfo);
   bmInfo.TitleW = L"Must be parent to all old roots";
   PXCp_BMHandle        hBMRootest = NULL;
   // This one will be the the new root and all other will be it's children:
   PXCp_BMInsertItem(hDoc, NULL, (PXCp_BMHandle)PBM_ROOT, &hBMRootest,
&bmInfo);

   // And now move the item
   PXCp_BMMoveItem(hDoc, hBM, NULL, (PXCp_BMHandle)PBM_LAST);
   // Clean up
   PXCp_Delete(hDocument);
```

3.1.5.7 PXCp_BMSetItemInfo

PXCp_BMSetItemInfo

Top Previous Next

PXCp_BMSetItemInfo sets the designated information to the specified bookmark item.

```
HRESULT  PXCp_BMSetItemInfo(
    PDFDocument pDocument,
    PXCp_BMHandle bmItem,
    LPPXCp_BMInfo pbmItemInfo
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

bmItem

[in] *bmItem* specifies a bookmark item handle.

pbmItemInfo

[in] *pbmItemInfo* specifies a pointer to the **PXCp_BMInfo** structure containing information apply for the item.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
  // Set the new title to the bookmark item specified:
   PDFDocument    hDocument;
```

```
PXCp_BMHandle  hBMItem;
// The next structure will containe the necessary information:
PXCp_BMInfo bmItemInfo = {0};
bmItemInfo.cbSize = sizeof(PXCp_BMInfo);
// We like to set just the title of the item:
bmItemInfo.Mask = BMIM_TitleW;
bmItemInfo.TitleW = L"This is the new title";
HRESULT res = PXCp_BMSetItemInfo(hDocument, hBMItem, &bmItemInfo);
if (IS_DS_FAILED(res))
{
    // Report an error
}
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.5.8 PXCp_GetRootBMItem

PXCp_GetRootBMItem

Top Previous Next

PXCp_GetRootBMItem retrieves the handle for the root Bookmark (Outline) item for the document.

```
HRESULT  PXCp_GetRootBMItem(
    PDFDocument pDocument,
    PXCp_BMHandle* bmItem
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

bmItem

[out] *bmItem* specifies a pointer to a variable of the `PXCp_BMHandle` which should receive the root bookmark handle.
`PXCp_BMHandle` is defined as follow:
`typedef void* PXCp_BMHandle;`

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
  // Get root bookmark item of the document:
   PDFDocument    hDocument;
   PXCp_BMHandle  hBMrootItem = NULL;
   HRESULT res = PXCp_GetRootBMItem(hDocument, &hBMrootItem);
   if (IS_DS_FAILED(res))
   {
       // Report an error
   }
   // Now 'hBMrootItem' contains the valid handle of the root item
   ...
   // Clean up
   PXCp_Delete(hDocument);
```

3.1.6 Page Manipulation

3.1.6.1 PXCp_InsertPagesTo

PXCp_InsertPagesTo

Top Previous Next

PXCp_InsertPagesTo makes a copy of a page from one PDF document and inserts this page into another. Pages to copy may also be specified by a range of required pages to copy/insert in one action or a single page may be copied/inserted several times - within the target document. The position within the target document may be specified for each page.

The page inserted within the target document may also be empty (simply created by the **PXCp_Init** function).

```
HRESULT  PXCp_InsertPagesTo(
    PDFDocument pSrcObject,
    PDFDocument pDestObject,
    LPPXCp_CopyPageRange PageRanges,
    DWORD RangesCount,
    DWORD Flags
);
```

Parameters

pSrcObject

[in] pSrcObject specifies the PDF object previously created by the function PXCp_Init for the source document from which pages will be taken.

pDestObject

[in] pDestObject specifies the PDF object previously created by the function PXCp_Init for the target document.

PageRanges

[in] PageRanges specifies an array of a Range of pages. Each element being of the PXCp_CopyPageRange type.

RangesCount

[in] RangesCount specifies the number of Range's contained within the PageRanges array.

Flags

[in] *Flags* is a reserved argument. Should be set to `0`.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
       // Example on how to take the first page from a source document
       // and place it twice in a (new) target document
    PDFDocument hDestDoc;
    hr = PXCp_Init(&hDestDoc, NULL, NULL);
    if (IS_DS_FAILED(hr))
    {
     // Handle any error creating new document
     // ...
    }
    PXCp_CopyPageRange        MyRange[2];
    MyRange[0].StartPage = 0;                // take first page
    MyRange[0].EndPage = 0;                 // and only first page
    MyRange[0].InsertBefore = -1;         // insert in the end of the document
    MyRange[0].Reserved = 0;                // reserved value
    // The same with the second range
    MyRange[1].StartPage = 0;
    MyRange[1].EndPage = 0;
    MyRange[1].InsertBefore = -1;
    MyRange[1].Reserved = 0;
    // Now we can insert two copies of one page into new document
    hr = PXCp_InsertPagesTo(hSrcDoc, hDestDoc, MyRange, 2, 0);
    if (IS_DS_FAILED(hr))
    {
     // Handle error inserting pages
     // ...
    }

    // In this place the destination document could be saved to a file or
further processed
```

3.1.6.2 PXCp_RemovePage

PXCp_RemovePage

Top Previous Next

PXCp_RemovePage removes a specified page and all it's content from a document.

```
HRESULT  PXCp_RemovePage(
     PDFDocument pDocument,
     DWORD PageNumber
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] *PageNumber* specifies number of the page to be removed.
Note: `Important !` PDF file page numbering is Zero based - the initial page being numbered `0`, the second page value being numbered `1` etc.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
   PDFDocument        hDocument;
   // Remove first page from the document:
   // remember, that pages are numerated from 0 (ie: within a PDF file, page 1
is actually page Zero!)!
   HRESULT res = PXCp_RemovePage(hDocument, 0);
   if (IS_DS_FAILED(res))
   {
       // Report any error
   }
   // Now, after saving the document the first page has been removed and page
2 is now page 1 (or page zero in a pdf file)
   ...
   // Clean up
   PXCp_Delete(hDocument);
```

3.1.6.3 PXCp_TransformPage

PXCp_TransformPage

Top Previous Next

PXCp_TransformPage allows the "transformation" of a PDF page. The page may be scaled, rotated etc. All transformation is controlled by the PDF matrix.

```
HRESULT  PXCp_TransformPage(
    PDFDocument pDocument,
    DWORD PageNumber,
    LPCPXC_Matrix matrix,
    DWORD flags
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] *PageNumber* specifies a page number within the document.

matrix

[in] *matrix* The Matrix which defines transformation parameters and is a combination of all required transformations (position, scaling, rotation etc) that are to be used to produce the desired output. See PDF Reference 1.6, section 4.2.2 for Common Transformations for more detailed information.

flags

[in] Flag that controls transformation of the page that may contain a 'Crop Box' (see. **PXCp_PageGetBox** for details). If this parameter is set **TPF_KeepPageContentCropped** (equal to 0x0001) then the 'CropBox' of the page will be taken into account and only the visible portion of the page will be transformed.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
     PDFDocument  hDocument;
    PXC_Matrix   TransformMatrix = {0};

    // Set transformation matrix to scale on X and Y axis

    TransformMatrix.a = 2.0; // X scale factor
    TransformMatrix.d = 3.0; // Y scale factor

    // Transform first page

    HRESULT hr = PXCp_TransformPage(hDocument, 0, &TransformMatrix,
TPF_KeepPageContentCropped);
    if (IS_DS_FAILED(hr))
    {
        // Handle any error
        ...
    }
    ...
```

3.1.7 Image Manipulation

3.1.7.1 PXCp_GetDocImageAsXCPage

PXCp_GetDocImageAsXCPage

Top Previous Next

PXCp_GetDocImageAsXCPage retrieves an image specified by it's ID from a PDF document and stores as an **Image-XChange Library** image page object.

The Image ID may be obtained using the **PXCp_ImageGetFromPage** function.

```
HRESULT  PXCp_GetDocImageAsXCPage(
    PDFDocument pObject,
    DWORD ImageID,
    void** pImage
);
```

Parameters

pObject

[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

ImageID

[in] *ImageID* specifies the image ID.

pImage

[out] *pImage* specifies a pointer to an **Image-XChange Library** image page object (`_XCPage`).
Note: When an image object is no longer required for use it should be be deleted from memory using the **Image-XChange Library** functions. Refer to **Image-XChange Library** help.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// example shows how to store all images from the document
   // into specified a folder

   HRESULT StoreAllImagesIntoFolder(PDFDocument hDoc, LPCWSTR FolderName)
   {
       HRESULT hr = DS_OK;

       // Get images count in the entire document
       DWORD ImageNumber = 0;
       hr = PXCp_ImageGetCount(hDoc, &ImageNumber, FALSE);
       if (IS_DS_FAILED(hr) || (ImageNumber == 0))
       {
           // either an error occured or there ares no images within the
document
           return hr;
       }

       void* pImagePage = NULL;

       for (DWORD i = 0; i < ImageNumber; i++)
       {
           // Get 'i' image
           hr = PXCp_GetDocImageAsXCPage(hDoc, i, &pImagePage);
           if (IS_DS_FAILED(hr))
           {
               // error retrieving this image - just continue
```

```
                continue;
            }
            // Saving page
            _XCImage pImage = NULL;
            hr = IMG_ImageCreateEmpty(&pImage);
            if(pImage == 0)
                continue;
            hr = IMG_ImageInsertPage(pImage, -1, pImagePage);

            wchar_t PathName[MAX_PATH];
            UINT PathSize = MAX_PATH;

            // Prepare correct file name
            wsprintfW(PathName, L"%s\\xxx%d.bmp", FolderName, i);

            // Set image options
            IMG_PageSetFormatLongParameter(pImagePage, FP_ID_FORMAT,
FMT_BMP_ID);
            IMG_PageSetFormatLongParameter(pImagePage, FP_ID_ITYPE,
ImageFormat_RGB_8);

            // Write image into file
            IMG_ImageSaveToFileW(pImage, PathName,
CreationDisposition_Overwrite);

            // Destroy unnecesary image
            IMG_ImageDestroy(pImage);
        }
        // Finished
        return DS_OK;
    }
```

3.1.7.2 PXCp_ImageClearAllData

PXCp_ImageClearAllData

Top Previous Next

PXCp_ImageClearAllData clears all data used to retrieve images from a document and should be called after all image processing is complete to free memory.

```
HRESULT  PXCp_ImageClearAllData(
    PDFDocument pDocument
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to retrieve information relating to images within the
document
// and for each page

    void PrintImageInformation(PDFDocument hDoc)
    {
        DWORD ImageCount = 0;
        HRESULT hr = PXCp_ImageGetCount(hDoc, &ImageCount, TRUE);
        if (IS_DS_FAILED(hr))
        {
            // process error
            ...
        }
        printf("Document contains %d images\n", ImageCount);

        // Get number of pages in the document
        DWORD PageNumber = 0;
        PXCp_GetPagesCount(hDoc, &PageNumber);

        // retrive information for each page
        for (DWORD pn = 0; pn < PageNumber; pn++)
        {
            DWORD ImagesOnPage = 0;
            hr = PXCp_ImageGetCountOnPage(hDoc, pn, &ImagesOnPage);
            if (IS_DS_FAILED(hr))
                continue;
            printf("On page %d there are %d images\n", pn, ImagesOnPage);

            // Now clear data relating to images on this page
            PXCp_ImageClearPageData(hDoc, pn);
        }

        // Clear all temporary data relating to images
        PXCp_ImageClearAllData(hDoc);

        return ImageCount;
    }
```

3.1.7.3 PXCp_ImageClearPageData

PXCp_ImageClearPageData

PXCp_ImageClearPageData clears all data used to retrieve images from a PDF page and should be called after all image processing from the page is complete to free used memory.

```
HRESULT  PXCp_ImageClearPageData(
    PDFDocument pDocument,
    DWORD PageNumber
);
```

Parameters

pDocument
[in] *pDocument* specifies an PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] *PageNumber* specifies a page number.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to retrieve the information relating to images in the
document
// and for each page

    void PrintImageInformation(PDFDocument hDoc)
    {
        DWORD ImageCount = 0;
        HRESULT hr = PXCp_ImageGetCount(hDoc, &ImageCount, TRUE);
        if (IS_DS_FAILED(hr))
        {
            // process error
            ...
        }
        printf("Document contains %d images\n", ImageCount);

        // Get number of pages in the document
        DWORD PageNumber = 0;
        PXCp_GetPagesCount(hDoc, &PageNumber);

        // retrive information for each page
        for (DWORD pn = 0; pn < PageNumber; pn++)
        {
            DWORD ImagesOnPage = 0;
            hr = PXCp_ImageGetCountOnPage(hDoc, pn, &ImagesOnPage);
            if (IS_DS_FAILED(hr))
                continue;
            printf("On page %d there are %d images\n", pn, ImagesOnPage);

            // Now clear data relating to images on this page
            PXCp_ImageClearPageData(hDoc, pn);
        }

        // Clear all temporary data relating to images
        PXCp_ImageClearAllData(hDoc);

        return ImageCount;
    }
```

3.1.7.4 PXCp_ImageGetCount

PXCp_ImageGetCount

Top Previous Next

PXCp_ImageGetCount retrieves the total count of unique images from within a PDF document. If an image appears within a document more than once, subsequent occurences are ignored to allow the function to identify a count of unique images and assign a reference number for future use by the **PXCp_GetDocImageAsXCPage** function.

```
HRESULT  PXCp_ImageGetCount(
    PDFDocument pDocument,
    DWORD* pImageCnt,
    BOOL bForceRecalc
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

pImageCnt
[out] *pImageCnt* specifies a pointer to a variable of the `DWORD` type to receive the image count.

bForceRecalc
[in] *bForceRecalc* specifies whether recalculation of images should be performed.
Note: If a document is modified by adding or removing pages - this parameter should be set to `TRUE`, otherwise set to `FALSE` to decrease processing time.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

When all operations with images are complete, **PXCp_ImageClearAllData** should be called to free memory used for image processing.

Example (C++).

```
// Example shows, how to retrieve the number of images in the entire document

    DWORD GetNumberOfImagesInTheDocument(PDFDocument hDoc)
    {
        DWORD ImageCount = 0;
        HRESULT hr = PXCp_ImageGetCount(hDoc, &ImageCount, FALSE);
        if (IS_DS_FAILED(hr))
        {
            // process error
            ...
        }
        // Clear all temporary data relating to images
        PXCp_ImageClearAllData(hDoc);

        return ImageCount;
    }
```

3.1.7.5 PXCp_ImageGetCountOnPage

PXCp_ImageGetCountOnPage

Top Previous Next

PXCp_ImageGetCountOnPage retrieves information relating to images placed on a specified PDF page, by parsing all it's contents.

This information may then be utilised by the **PXCp_ImageGetFromPage** function.

```
HRESULT  PXCp_ImageGetCountOnPage(
    PDFDocument pDocument,
    DWORD PageNumber,
    DWORD* pImageCnt
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] *PageNumber* specifies a page number.

pImageCnt
[out] *pImageCnt* specifies a pointer to a variable to receive the image count.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the page contains no images then the function will return **DPro_Wrn_PageHasNoImages** warning.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

Once all processing is complete the **PXCp_ImageClearPageData** function should be called to ensure memory resources are released.

Example (C++).

```
// Example shows how to retrieve the number of images on the specified page

    DWORD GetNumberOfImagesOnThePage(PDFDocument hDoc, DWORD PageNumber)
    {
        DWORD ImageCount = 0;
        HRESULT hr = PXCp_ImageGetCountOnPage(hDoc, PageNumber, &ImageCount);
        if (IS_DS_FAILED(hr))
        {
            // process error
            ...
        }
        // Clear all temporary data relating to images
        PXCp_ImageClearAllData(hDoc);

        return ImageCount;
    }
```

3.1.7.6 PXCp_ImageGetFromPage

PXCp_ImageGetFromPage

Top Previous Next

PXCp_ImageGetFromPage retrieves information about a specified image from a specified PDF page.

Returns an image ID by which the image may be retrieved using the **PXCp_GetDocImageAsXCPage**. Also retrieves information about the image placement returning it's matrix containing comprehensive information regarding the image placement, rotation, skew and scaling on the page.

If the image intersects with another or is placed above or beneath another,then it's order is defined by it's index value. A lower index value indicates the image is placed beneath another of a higher value.

Note: This function is the equivalent of **`PXCp_ImageGetFromPageEx`**`(`*`pDocument`*`, `*`PageNumber`*`, `*`ImageOnPageNumber`*`, `*`pImageHandle`*`, `*`pMatrix`*`, 0).`

```
HRESULT  PXCp_ImageGetFromPage(
    PDFDocument pDocument,
    DWORD PageNumber,
    DWORD ImageOnPageNumber,
    LONG* pImageHandle,
    PXC_Matrix* pMatrix
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] *PageNumber* specifies a page number.

ImageOnPageNumber

[in] *ImageOnPageNumber* specifies an image number. The Total image count for a page may be retrived using the **PXCp_ImageGetCountOnPage** function.

pImageHandle

[out] *pImageHandle* specifies a pointer to a variable which receives the image ID. In the case of an error this variable is set to `-1`.

pMatrix

[out] Matrix which defines element transformation parameters. It is a combination of all transformations (position, scaling, rotation and so on) that have been used to produce the resulting output. The `e` and `f` parameters contain the starting location of the text element on the page, and are based on the mediabox coordinates.
See PDF Reference 1.6, section 4.2.2 Common Transformations for more information.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example of how to retrieve all images from the document page
    DWORD PageNumber = 0;
    DWORD cnt = 0;
    hr = PXCp_ImageGetCountOnPage(hDoc, PageNumber, &cnt);
    if (IS_DS_FAILED(hr) || (cnt == 0))
    {
        // Handle error or image absence
        // ...
    }
    // Go through all image and retrieve them
    for (DWORD j = 0; j < cnt; j++)
    {
        LONG                handle = 0;
        PXC_Matrix        matrix;
        hr = PXCp_ImageGetFromPage(hDoc, PageNumber, j, &handle, &matrix);
        if (IS_DS_FAILED(hr))
        {
            Log << nl;
            continue;
        }
        // Now we've got the matrix with the required image information
        _XCPage pImagePage = NULL;
        hr = PXCp_GetDocImageAsXCPage(hDoc, (DWORD)handle, &pImagePage);
        if (IS_DS_FAILED(hr))
        {
            // Handle error
            continue;
        }
        // At this point we have the image - handle it
        // ...
        // When no longer required - free it
        IMG_PageDestroy(pImagePage);
    }
    // Free data regarding images from this page
    PXCp_ImageClearPageData(hDoc, PageNumber);
```

3.1.7.7 PXCp_ImageGetFromPageEx

PXCp_ImageGetFromPageEx

Top Previous Next

PXCp_ImageGetFromPageEx retrieves information about a specified image from a specified PDF page.

Returns an image ID by which the image may be extracted using the **PXCp_GetDocImageAsXCPage**. Also retrieves information relating to image placement returning the image's matrix, containing comprehensive information regarding the image location, rotation, skew and scaling on the page.

If the image intersects with another or is placed above or beneath another, the image order is reflected in the image index value. A lower index value indicates the image is placed beneath another of a higher value.

```
HRESULT  PXCp_ImageGetFromPageEx(
    PDFDocument pDocument,
    DWORD PageNumber,
    DWORD ImageOnPageNumber,
    LONG* pImageHandle,
    PXC_Matrix* pMatrix,
    DWORD flags
);
```

Parameters

pDocument
: [in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber
: [in] *PageNumber* specifies a page number.

ImageOnPageNumber
: [in] *ImageOnPageNumber* specifies an image number. The Total image count for a page may be retrieved using the **PXCp_ImageGetCountOnPage** function.

pImageHandle
: [out] *pImageHandle* specifies a pointer to a variable which receives the image ID. In the case of an error this variable is set to `-1`.

pMatrix

[out] Matrix which defines element transformation parameters. It is a combination of all transformations (position, scaling, rotation and so on) that have been used to produce the resulting output. The `e` and `f` parameters contain the starting location of the text element on the page, and are based on the mediabox coordinates.
See PDF the Adobe PDF Reference 1.6, section 4.2.2 Common Transformations for more information.

flags

[in] *flags* specifies additional flags which determine how to treat information regarding the image. This may be `zero` or next value.

Constant	Value	Meaning
IGFPEF_IgnorePageRotation	0x0004	Do not include page rotation within the image matrix if the page is rotated.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example of how to retrieve all images from a document page
    DWORD PageNumber = 0;
    DWORD cnt = 0;
    hr = PXCp_ImageGetCountOnPage(hDoc, PageNumber, &cnt);
    if (IS_DS_FAILED(hr) || (cnt == 0))
    {
        // Handle error or an abscence of any images
        // ...
    }
    // Go through all images and retrieve them
    for (DWORD j = 0; j < cnt; j++)
    {
        LONG                handle = 0;
        PXC_Matrix        matrix;
        hr = PXCp_ImageGetFromPageEx(hDoc, PageNumber, j, &handle, &matrix,
0);
        if (IS_DS_FAILED(hr))
        {
            Log << nl;
            continue;
        }
        // Now we have the matrix with the required image information
        _XCPage pImagePage = NULL;
        hr = PXCp_GetDocImageAsXCPage(hDoc, (DWORD)handle, &pImagePage);
        if (IS_DS_FAILED(hr))
        {
            // Handle error
            continue;
        }
        // In this place we've got image - handle it
        // ...
        // After it is no longer required - free it
        IMG_PageDestroy(pImagePage);
    }
    // Free data regarding images from this page
    PXCp_ImageClearPageData(hDoc, PageNumber);
```

3.1.7.8 PXCp_SaveDocImageIntoFileA

PXCp_SaveDocImageIntoFileA

Top Previous Next

PXCp_SaveDocImageIntoFileA retrieves an image specified by it's ID from a PDF document and saves the specified image(s) from the PDF document, using the required (Raster) image format parameters as defined and supported by the library.

The Image ID may be obtained using the **PXCp_ImageGetFromPage** function.

```
HRESULT  PXCp_SaveDocImageIntoFileA(
    PDFDocument pObject,
    DWORD ImageID,
    LPCSTR FileName,
    PXCp_SaveImageOptions* pSaveOptions
);
```

Parameters

pObject
[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

ImageID
[in] *ImageID* specifies the image ID.

FileName
[out] *FileName* specifies a `NULL` terminated UNICODE string with the full file name for the file.

pSaveOptions
[out] *pSaveOptions* specifies a pointer to the **PXCp_SaveImageOptions** structure which applies the options for the required image file.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function is the ASCII equivalent of UNICODE function **PXCp_SaveDocImageIntoFileW**.

Example (C++).

```
        // This example shows, how to retrieve all images from a specified page
    // and to save them within a single TIFF file
    void SaveAllImagesFromPageIntoTIFF(PDFDocument hDoc, DWORD PageNumber,
LPCSTR FileName)
    {
        HRESULT hr = DS_OK;

        // Get images count on the page
        DWORD ImageNumber = 0;
        hr = PXCp_ImageGetCountOnPage(hDoc, PageNumber, &ImageNumber);
        if (IS_DS_FAILED(hr) || (ImageNumber == 0))
        {
            // Report error
            return;
        }

        // Fill save options for TIFF file
        PXCp_SaveImageOptions SaveOpts;
        ::ZeroMemory(&SaveOpts, sizeof(PXCp_SaveImageOptions));

        // Allow appending images to existing file
        SaveOpts.bAppendToExisting = TRUE;
        SaveOpts.bConvertToGray = FALSE;
        SaveOpts.bDither = 0;
        SaveOpts.bWriteAlpha = 0;

        // Write to TIFF
        SaveOpts.fmtID = PRO_FMT_TIFF_ID;
        SaveOpts.imgType = ImType_rgb_24bpp;
        SaveOpts.CompressionMethod = ImCompression_None;

        for (DWORD ic = 0; ic < ImageNumber; ic++)
        {
            LONG                handle = 0;
            PXC_Matrix        matrix;
            // Get information about the image
            hr = PXCp_ImageGetFromPage(hDoc, PageNumber, ic, &handle,
&matrix);
            if (IS_DS_FAILED(hr))
            {
                // Process this error
                continue;
            }

            // Save (append) this image into the file: ";
            hr = PXCp_SaveDocImageIntoFileA(hDoc, handle, FileName,
&SaveOpts);
            if (IS_DS_FAILED(hr))
            {
                // Process this error
                continue;
            }
        }

        // All done
        // Clear all temporary image data
        PXCp_ImageClearAllData(hDoc);
    }
```

3.1.7.9 PXCp_SaveDocImageIntoFileW

PXCp_SaveDocImageIntoFileW

Top Previous Next

PXCp_SaveDocImageIntoFileW retrieves an image specified by it's ID from a PDF document and saves the specified image(s) from the PDF document, using the required (Raster) image format parameters as defined and supported by the library.

The Image ID may be obtained using the **PXCp_ImageGetFromPage** function.

```
HRESULT  PXCp_SaveDocImageIntoFileW(
    PDFDocument pObject,
    DWORD ImageID,
    LPCWSTR FileName,
    PXCp_SaveImageOptions* pSaveOptions
);
```

Parameters

pObject
[in] *pObject* specifies the PDF object previously created by the function **PXCp_Init**.

ImageID
[in] *ImageID* specifies the image ID.

FileName
[out] *FileName* specifies a `NULL` terminated UNICODE string with the full file name for the file.

pSaveOptions
[out] *pSaveOptions* specifies a pointer to the **PXCp_SaveImageOptions** structure which applies the options for the required image file.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function is the UNICODE equivalent of ASCII function **PXCp_SaveDocImageIntoFileA**.

Example (C++).

```
    // This example shows, how to retrieve all images from a specified page
    // and to save them within a single TIFF file

    void SaveAllImagesFromPageIntoTIFF(PDFDocument hDoc, DWORD PageNumber,
LPCWSTR FileName)
    {
        HRESULT hr = DS_OK;

        // Get images count on the page
        DWORD ImageNumber = 0;
        hr = PXCp_ImageGetCountOnPage(hDoc, PageNumber, &ImageNumber);
        if (IS_DS_FAILED(hr) || (ImageNumber == 0))
        {
            // Report error
            return;
        }

        // Fill save options for TIFF file
        PXCp_SaveImageOptions SaveOpts;
        ::ZeroMemory(&SaveOpts, sizeof(PXCp_SaveImageOptions));

        // Allow appending images to existing file
        SaveOpts.bAppendToExisting = TRUE;
        SaveOpts.bConvertToGray = FALSE;
        SaveOpts.bDither = 0;
        SaveOpts.bWriteAlpha = 0;

        // Write to TIFF
        SaveOpts.fmtID = PRO_FMT_TIFF_ID;
        SaveOpts.imgType = ImType_rgb_24bpp;
        SaveOpts.CompressionMethod = ImCompression_None;

        for (DWORD ic = 0; ic < ImageNumber; ic++)
        {
            LONG                handle = 0;
            PXC_Matrix        matrix;
            // Get information about the image
            hr = PXCp_ImageGetFromPage(hDoc, PageNumber, ic, &handle,
&matrix);
            if (IS_DS_FAILED(hr))
            {
                // Process this error
                continue;
            }

            // Save (append) this image into the file: ";
            hr = PXCp_SaveDocImageIntoFileW(hDoc, handle, FileName,
&SaveOpts);
            if (IS_DS_FAILED(hr))
            {
                // Process this error
                continue;
            }
        }

        // All done
        // Clear all temporary image data
        PXCp_ImageClearAllData(hDoc);
    }
```

3.1.8 Thumbnails Manipulation

3.1.8.1 PXCp_PageGetThumbnail

PXCp_PageGetThumbnail

Top Previous Next

PXCp_PageGetThumbnail retrieves a thumbnail image associated with the specified PDF page.

```
HRESULT  PXCp_PageGetThumbnail(
    PDFDocument pDocument,
    DWORD PageNumber,
    void** pImage
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] *PageNumber* specifies a page number in the document.

pImage

[out] *pImage* specifies a pointer to an **Image-XChange Library** image page object (`_XCPage`).
Note: When the image object is no longer required it should be deleted by the **Image-XChange Library** functions. Refer to **Image-XChange Library** help.

Return Values

If the function succeeds, the return value is a non-negative integer.
If there is no thumbnail associated with the page then the function will return **`DPro_Wrn_PageHasNoThumbnail`** warning.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
   // Check if the first page has a thumbnail and retrieve it:
   PDFDocument  hDocument;
   BOOL         ThumbnailIsPresent = FALSE;
   // Retrieve the information about the thumbnail for the first page in the
document:
```

```
HRESULT res = PXCp_PageHasThumbnail(hDocument, 0, &ThumbnailIsPresent);
if (IS_DS_FAILED(res))
{
    // Report an error
}
if (!ThumbnailIsPresent)
  return; // no thumbnail is set to the page;
void* xcPage = NULL;
res = PXCp_PageGetThumbnail(hDocument, 0, &xcPage);
if (IS_DS_FAILED(res))
{
    // Report an error
}
// Now Image-XChange SDK Library object for the image page is obtained
// and can be used for different image operations
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.8.2 PXCp_PageHasThumbnail

PXCp_PageHasThumbnail

Top Previous Next

PXCp_PageHasThumbnail checks if there is thumbnail image associated with the specified PDF page.

```
HRESULT  PXCp_PageHasThumbnail(
    PDFDocument pDocument,
    DWORD PageNumber,
    BOOL* bThumbnailPresent
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] *PageNumber* specifies a page number in the document.

bThumbnailPresent
[out] *bThumbnailPresent* specifies a pointer to a variable of the BOOL type to receives the result.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
PDFDocument  hDocument;
BOOL         ThumbnailIsPresent = FALSE;
// Retrive the information about thumbnail for the first page in the
```

```
document:
   HRESULT res = PXCp_PageHasThumbnail(hDocument, 0, &ThumbnailIsPresent);
   if (IS_DS_FAILED(res))
   {
       // Report an error
   }
   ...
   // Clean up
   PXCp_Delete(hDocument);
```

3.1.8.3 PXCp_PageRemoveThumbnail

PXCp_PageRemoveThumbnail

Top Previous Next

PXCp_PageRemoveThumbnail removes a thumbnail image associated with the specified PDF page.

```
HRESULT  PXCp_PageRemoveThumbnail(
    PDFDocument pDocument,
    DWORD PageNumber
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] *PageNumber* specifies a page number in the document.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
   PDFDocument  hDocument;
   DWORD        PageNumber = 0;

   // Removes thumbnail from the page in the document:

   HRESULT res = PXCp_PageRemoveThumbnail(hDocument, PageNumber);
   if (IS_DS_FAILED(res))
   {
       // Report an error
   }
   ...
   // Clean up
   PXCp_Delete(hDocument);
```

3.1.8.4 PXCp_PageSaveThumbnailToFile

PXCp_PageSaveThumbnailToFile

Top Previous Next

PXCp_PageSaveThumbnailToFile retrieve's a thumbnail image associated with the indicated PDF page and saves it to the specified file.

```
HRESULT  PXCp_PageSaveThumbnailToFile(
    PDFDocument pDocument,
    DWORD PageNumber,
    LPCWSTR FileName,
    PXCp_SaveImageOptions* pSaveOptions
);
```

Parameters

pDocument
[in] Specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] Specifies a page number in the document.

FileName
[in] Specifies a pointer to a `NULL` terminated UNICODE string that contains full file name.

pSaveOptions
[in] Specifies a pointer to a **PXCp_SaveImageOptions** structure which applies the options for the required image file.

Return Values

If the function succeeds, the return value is a non-negative integer.
If there is no thumbnail associated with the page then the function will return **`DPro_Wrn_PageHasNoThumbnail`** warning.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
   // Check if the first page has a thumbnail and save it to the file:
   PDFDocument  hDocument;
   BOOL         ThumbnailIsPresent = FALSE;
   // Retrive the information about thumbnail for the first page in the
document:
   HRESULT res = PXCp_PageHasThumbnail(hDocument, 0, &ThumbnailIsPresent);
   if (IS_DS_FAILED(res))
   {
       // Report an error
   }
   if (!ThumbnailIsPresent)
     return; // no thumbnail is set to the page;
   // Prepare save options:
   PXCp_SaveImageOptions        SaveOpts;
   ::ZeroMemory(&SaveOpts, sizeof(PXCp_SaveImageOptions));
   SaveOpts.bAppendToExisting = FALSE;
   SaveOpts.bConvertToGray = FALSE;
   SaveOpts.bDither = 0;
   SaveOpts.bWriteAlpha = 0;
   SaveOpts.fmtID = PRO_FMT_BMP_ID;          // will save to 'bmp' format
   SaveOpts.imgType = ImType_rgb_24bpp;
   SaveOpts.CompressionMethod = ImCompression_None;

   hr = PXCp_PageSaveThumbnailToFile(hDocument, 0, L"C:\\thumbnail.bmp",
&SaveOpts);
   if (IS_DS_FAILED(res))
   {
       // Report an error
   }
   // Now image is saved to the file
   ...
   // Clean up
   PXCp_Delete(hDocument);
```

3.1.8.5 PXCp_PageSetThumbnail

PXCp_PageSetThumbnail

Top Previous Next

PXCp_PageSetThumbnail sets a thumbnail image to a specified page.

```
HRESULT  PXCp_PageSetThumbnail(
    PDFDocument pDocument,
    DWORD PageNumber,
    void* pImage,
    PXCp_ThumbFlag flag
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] *PageNumber* specifies the required page number within the document.

pImage

[in] *pImage* specifies **Image-XChange Library** image page object - `_XCPage`. Refer to **Image-XChange Library** help for details.

flag

[in] *flag* specifies the scaling method for the image within the PDF document.
Possible values are:

Value	Meaning
thf_SetAsIs	Place the specified image without any scaling. `Caution!` - a large image will occupy a large space within the document.

`thf_Scale`	Scale the image to be as small as possible. The Image will have the same proportions as the page.
`thf_KeepProportions`	The existing Image proportions will be retained.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

Usually thumbnails are shown as small images by PDF viewers, consequently there is little logic in using a large image as thumbnail. To ensure efficient file sizes - scale thumbnail images.

Remarks

Note: After the function call image *pImage* may be modified (according to *flag* value).

Example (C++).

```
// Set thumbnail to the first page of the document:
PDFDocument  hDocument;
// The variable 'xcPage' is the Image-XChange SDK (library) image page
// obtained by some relevant method:
void* xcPage;
// Set this image as a thumbnail to the first page in the document,
// using scaling, though the image will be scaled and will have proportions
// the same as the page. The size of the data for the image in PDF document
// will be as small as possible:
HRESULT res = PXCp_PageSetThumbnail(hDocument, 0, xcPage, thf_Scale);
if (IS_DS_FAILED(res))
{
    // Report an error
}
// Now save the document and open the PDF viewer if required.
...
// Clean up
PXCp_Delete(hDocument);
```

3.1.8.6 PXCp_PageSetThumbnailFromFile

PXCp_PageSetThumbnailFromFile

Top Previous Next

PXCp_PageSetThumbnailFromFile sets a thumbnail image specified by an image file name to a specified page.

```
HRESULT  PXCp_PageSetThumbnailFromFile(
    PDFDocument pDocument,
    DWORD PageNumber,
    LPCWSTR imageFileName,
    DWORD imagePageNumber,
    PXCp_ThumbFlag flag
);
```

Parameters

pDocument
: [in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber
: [in] *PageNumber* specifies the required page number within the document.

imageFileName
: [in] *imageFileName* specifies NULL terminated UNICODE string with the image file name.

imagePageNumber
: [in] *imagePageNumber* specifies the page number in the image file.

flag
: [in] *flag* specifies the scaling method for the image within the PDF document.
Possible values are:

Value	Meaning
thf_SetAsIs	Place the specified image without any scaling. `Caution!` - a large image will occupy a large space within the document.
thf_Scale	Scale the image to be as small as possible. The Image will have the same proportions as the page.
thf_KeepProportions	The existing Image proportions will be retained.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

Usually thumbnails are shown as small images by PDF viewers, consequently there is little logic in using a large image as thumbnail. To ensure efficient file sizes - scale thumbnail images.

Example (C++).

```
   // Set thumbnail to the first page of the document, using the image file
name

   PDFDocument  hDocument;

   // the variable 'ImageFileName' is the image file name
   // obtained as required:

   LPCWSTR ImageFileName;

   // Set this image as thumbnail to the first page in the document,
   // using scaling, though the image will be scaled and will have proportions
   // the same as the page. The size of the data for the image in PDF document
   // will be as small as possible:
```

```
   HRESULT res = PXCp_PageSetThumbnailFromFile(hDocument, 0, ImageFileName, 0,
thf_Scale);

   if (IS_DS_FAILED(res))
   {
       // Report an error
   }

   // Now save the document and open the PDF viewer if required
   ...

   // Clean up
   PXCp_Delete(hDocument);
```

3.1.8.7 PXCp_PageSetThumbnailFromHBITMAP

PXCp_PageSetThumbnailFromHBITMAP

Top Previous Next

PXCp_PageSetThumbnailFromHBITMAP sets a thumbnail image specified by `HBITMAP` to a specified page.

```
HRESULT  PXCp_PageSetThumbnailFromHBITMAP(
    PDFDocument pDocument,
    DWORD PageNumber,
    HBITMAP hBitmap,
    HPALETTE hPal,
    PXCp_ThumbFlag flag
);
```

Parameters

pDocument
[in] *pDocument* specifies the PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] *PageNumber* specifies the required page number within the document.

hBitmap
[in] *hBitmap* specifies the `HBITMAP` object of the image.

hPal
[in] *hPal* specifies the `HPALETTE` object of image palette.

flag
[in] *flag* specifies the scaling method for the image within the PDF document.
Possible values are:

Value	Meaning
thf_SetAsIs	Place the specified image without any scaling. `Caution!` - a large image will occupy a large space within the document.
thf_Scale	Scale the image to be as small as possible. The Image will have the

	same proportions as the page.
`thf_KeepProportions`	The existing Image proportions will be retained.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

Usually thumbnails are shown as small images by PDF viewers, consequently there is little logic in using a large image as thumbnail. To ensure efficient file sizes - scale thumbnail images.

Example (C++).

```
       // Set thumbnail to a specified page of the document
    void SetThumbnail(PDFDocument  hDocument, DWORD PageNumber, LPCWSTR
ImageFileName)
    {
        // We need variables for the Image-XChange objects (uses Library
functions from our related Image-XChange SDK)
        // to read the image and to retrieve the required HBITMAP:

        _XCImage         _image = NULL;
        _XCPage         _iPage = NULL;

        // Read image file

        hr = IMG_ImageCreateEmpty(&_image);
        if (IS_DS_FAILED(hr))
        {
            // Handle error
            //...
        }
        hr = IMG_ImageLoadW(_image, ImageFileName, 0);
        if (IS_DS_FAILED(hr))
        {
            IMG_ImageDestroy(_image);
            // Handle error
            //...
        }
        // Loading first image page:
        hr = IMG_ImageGetPage(_image, 0, &_iPage);
        if (IS_DS_FAILED(hr))
        {
            IMG_ImageDestroy(_image);
            // Handle error
            //...
        }

        HBITMAP hBitmap;
        // Retrive HBITMAP from the image
        hr = IMG_PageGetHBITMAP(_iPage, &hBitmap, NULL, NULL, 0);
        if (IS_DS_FAILED(hr))
        {
            IMG_ImageDestroy(_image);
            // Handle error
            //...
        }
        // Set thumbnal using obtained HBITMAP:
        hr = PXCp_PageSetThumbnailFromHBITMAP(hDoc, 0, hBitmap, NULL,
thf_Scale);
        if (IS_DS_FAILED(hr))
        {
            IMG_ImageDestroy(_image);
            // Handle error
            //...
        }

        // Done.
    }
```

3.1.9 Annotations Manipulation

3.1.9.1 PXCp_Add3DAnnotationA

PXCp_Add3DAnnotationA

Top Previous Next

PXCp_Add3DAnnotationA adds a *3D annotation (artwork/image)* to the content of the PDF object.

Note: For more information about 3D annotations please refer to the Adobe PDF Specification Version 1.6 or later.

```
HRESULT  PXCp_Add3DAnnotationA(
    PDFDocument hDocument,
    DWORD PageNumber,
    LPCPXC_RectF rect,
    LPCSTR pszTitle,
    DWORD dwAnnotOption,
    void* AltImage,
    DWORD imFlag,
    const _PXC_3DView* def_view,
    LONG def_view_id,
    LPBYTE lpBuf,
    UINT nBufSize,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument

[in] *hDocument* specifies a PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] Parameter *PageNumber* specifies the page number into which to add the annotation.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

pszTitle

[in] Pointer to a null-terminated string that specifies the title of the annotation's pop-up window.

dwAnnotOption

[in] Specifies the options specific for the 3D annotation. The possible values are the logical combinations of the following values, that may be grouped by meaning into the following groups:

Activation:

Constant	Value	Meaning
ActivateOn_Explicit	0x0000	The annotation should remain inactive until explicitly activated by a script or user action. This is the default action.
ActivateOn_OnPageOpen	0x0001	The annotation should be activated as soon as the page containing the annotation is opened.
ActivateOn_OnPageVisible	0x0002	The annotation should be activated as soon as any part of the page containing the annotation becomes visible.

Activation Effect:

Constant	Value	Meaning
ActivationEff_Live	0x0000	The artwork is instantiated, and animations, if present, are enabled. This is the default value.
ActivationEff_Loaded	0x0010	The artwork is instantiated, but animations are disabled.

Deactivation:

Constant	Value	Meaning
DeactivateOn_OnPageInvisible	0x0000	The annotation should be deactivated as soon as the page is closed. This is the default value.
DeactivateOn_OnPageClose	0x0100	The annotation should be activated as soon as the page containing the annotation is opened.
DeactivateOn_Explicit	0x0200	The annotation should remain active until explicitly deactivated by a script or user action.

Deactivation Effect:

Constant	**Value**	**Meaning**
`DeactivationEff_Unloaded`	`0x0000`	The artwork instance becomes uninstantiated. This is the default value.
`DeactivationEff_Loaded`	`0x1000`	The artwork instance should stay instantiated.
`DeactivationEff_Live`	`0x2000`	The artwork instance should stay live.

AltImage

[in] Specifies the image to be shown instead of a 3D annotation in PDF viewer's that do not support this type of annotations.

imFlag

[in] Specifies the way in which the *AltImage* image is specified. Possible values are:

Value	**Description**
Im3dAnnot_IXCObject	*AltImage* is the **Image-XChange Library** object - `_XCPage`.
Im3dAnnot_FileName	*AltImage* is the full image file name. The type of *AltImage* is `LPCSTR`.

def_view

[in] Specifies the pointer to the **PXC_3DView** structure and defines default view.

def_view_id

[in] Specifies the index of a U3D stream view which should be used as the default view. This parameter is ignored when *def_view* is `NULL`.

lpBuf

[in] Specifies the buffer that contains the 3D stream to be shown in the annotation.

nBufSize

[in] Specifies the size (in `BYTEs`) of the 3D stream buffer.

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXCp_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to add 3D annotation to the document
```

```
    void Add3DAnnotation(PDFDocument hDoc,  LPCSTR Stream3DFileName, LPCSTR
AltImageFileName)
    {
        HRESULT hr = DS_OK;

        // Annotation rectangle

        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Common annotation information

        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;


        DWORD _Options = ActivateOn_OnPageOpen | DeactivateOn_OnPageInvisible
|
            ActivationEff_Live | DeactivationEff_Loaded;

        // Fill 3DView structure

        PXC_3DView view = {sizeof(PXC_3DView)};
        ::lstrcpyW(view.m_ExtName, L"Default");
        view.m_CO = 1300.32;
        view.m_FOV = 30.0;
        view.m_C2W[0] = view.m_C2W[7] = 1.0;
        view.m_C2W[5] = -1.0;
        view.m_C2W[9] = -0.0893402;
        view.m_C2W[10] = -1300.32;
        view.m_C2W[11] = 20.335;

        // Prepare 3D stream data

        LPBYTE        _buf = NULL;
        UINT        _bufLen = 0;
        // Ope file for 3D stream

        HANDLE f = CreateFileA(Stream3DFileName, GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (f == INVALID_HANDLE_VALUE)
        {
            // report error and exit
            ...
        }

        // Get file size

        _bufLen = GetFileSize(f, NULL);

        // Create the buffer for stream data
        _buf = new BYTE[_bufLen];
        DWORD numreaded;

        // Read data to the buffer

        ReadFile(f, _buf, _bufLen, &numreaded, NULL);

        // Close file handle

        CloseHandle(f);

        // Place annotation onto first page
        hr = PXCp_Add3DAnnotationA(hDoc, 0, &rect, "This is title", _Options,
AltImageFileName,
            Im3dAnnot_FileName, &view, 0, _buf, _bufLen, NULL);

        // Clear the buffer

        delete[] _buf;

        if (IS_DS_FAILED(hr))
        {
            // report the error
            ...
        }

        // done
    }
```

3.1.9.2 PXCp_Add3DAnnotationW

PXCp_Add3DAnnotationW

Top Previous Next

PXCp_Add3DAnnotationW adds a *3D annotation (artwork/Image)* to the content of the PDF object.

Note: This function is the UNICODE equivalent of the function **PXCp_Add3DAnnotationA**.

Note: For more information about 3D annotations please refer to the Adobe PDF Specification Version 1.6 or later.

```
HRESULT  PXCp_Add3DAnnotationW(
    PDFDocument hDocument,
    DWORD PageNumber,
    LPCPXC_RectF rect,
    LPCWSTR pwszTitle,
    DWORD dwAnnotOption,
    void* AltImage,
    DWORD imFlag,
    const _PXC_3DView* def_view,
    LONG def_view_id,
    LPBYTE lpBuf,
    UINT nBufSize,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument

[in] *hDocument* specifies a PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] Parameter *PageNumber* specifies the page number to which to add the annotation.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

pwszTitle

[in] Pointer to a null-terminated string that specifies the title of the annotation's pop-up window.

dwAnnotOption

[in] Specifies the options specific for the 3D annotation. The possible values are the logical combinations of the following values, that may be grouped by meaning into the following groups:

Activation:

Constant	Value	Meaning
ActivateOn_Explicit	0x0000	The annotation should remain inactive until explicitly activated by a script or user action. This is the default action.
ActivateOn_OnPageOpen	0x000	The annotation should be activated as soon as the page

	1	containing the annotation is opened.
ActivateOn_OnPageVisible	0x0002	The annotation should be activated as soon as any part of the page containing the annotation becomes visible.

Activation Effect:

Constant	Value	Meaning
ActivationEff_Live	0x0000	The artwork is instantiated, and animations, if present, are enabled. This is the default value.
ActivationEff_Loaded	0x0010	The artwork is instantiated, but animations are disabled.

Deactivation:

Constant	Value	Meaning
DeactivateOn_OnPageInvisible	0x0000	The annotation should be deactivated as soon as the page is closed. This is the default value.
DeactivateOn_OnPageClose	0x0100	The annotation should be activated as soon as the page containing the annotation is opened.
DeactivateOn_Explicit	0x0200	The annotation should remain active until explicitly deactivated by a script or user action.

Deactivation Effect:

Constant	Value	Meaning
DeactivationEff_Unloaded	0x0000	The artwork instance becomes uninstantiated. This is the default value.
DeactivationEff_Loaded	0x1000	The artwork instance should stay instantiated.
DeactivationEff_Live	0x2000	The artwork instance should stay live.

AltImage

[in] Specifies the image to be shown instead of a 3D annotation in PDF viewer's that do not support this type of annotation.

imFlag

[in] Specifies the way in which the *AltImage* image is specified. Possible values are:

Value	Description
Im3dAnnot_IXCObject	*AltImage* is the **Image-XChange Library** object - `_XCPage`.
Im3dAnnot_FileName	*AltImage* is the full image file name. The type of *AltImage* is `LPCSTR`.

def_view

[in] Specifies a pointer to the **PXC_3DView** structure and defines the default view.

def_view_id

[in] Specifies the index of a U3D stream view which should be used as the default view. This parameter is ignored when *def_view* is `NULL`.

lpBuf

[in] Specifies a buffer that contains the 3D stream to be shown in the annotation.

nBufSize

[in] Specifies the size (in `BYTEs`) of the 3D stream buffer.

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXCp_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to add 3D annotations to the document

    void Add3DAnnotation(PDFDocument hDoc,  LPCWSTR Stream3DFileName, LPCWSTR
AltImageFileName)
    {
        HRESULT hr = DS_OK;

        // Annotation rectangle

        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Common annotation information

        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;


        DWORD _Options = ActivateOn_OnPageOpen | DeactivateOn_OnPageInvisible
|
            ActivationEff_Live | DeactivationEff_Loaded;

        // Fill 3DView structure

        PXC_3DView view = {sizeof(PXC_3DView)};
        ::lstrcpyW(view.m_ExtName, L"Default");
        view.m_CO = 1300.32;
        view.m_FOV = 30.0;
        view.m_C2W[0] = view.m_C2W[7] = 1.0;
        view.m_C2W[5] = -1.0;
        view.m_C2W[9] = -0.0893402;
        view.m_C2W[10] = -1300.32;
        view.m_C2W[11] = 20.335;

        // Prepare 3D stream data

        LPBYTE        _buf = NULL;
        UINT        _bufLen = 0;

        // Ope file for 3D stream

        HANDLE f = CreateFileW(Stream3DFileName, GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (f == INVALID_HANDLE_VALUE)
        {
            // report error and exit
            ...
        }

        // Get file size

        _bufLen = GetFileSize(f, NULL);

        // Create the buffer for stream data
        _buf = new BYTE[_bufLen];
        DWORD numreaded;

        // Read data to the buffer

        ReadFile(f, _buf, _bufLen, &numreaded, NULL);

        // Close file handle
        CloseHandle(f);

        // Place annotation onto first page
        hr = PXCp_Add3DAnnotationW(hDoc, 0, &rect, L"This is title", _Options,
AltImageFileName,
            Im3dAnnot_FileName, &view, 0, _buf, _bufLen, NULL);

        // Clear the buffer

        delete[] _buf;

        if (IS_DS_FAILED(hr))
        {
            // report the error
            ...
        }

        // done
    }
```

3.1.9.3 PXCp_AddGotoAction

PXCp_AddGotoAction

Top Previous Next

PXCp_AddGotoAction specifies a rectangular area on the specified page, that when selected, triggers the display of another specified page.

```
HRESULT  PXCp_AddGotoAction(
    PDFDocument hDocument,
    DWORD PageNumber,
    LPCPXC_RectF rect,
    DWORD page,
    PXC_OutlineDestination mode,
    double v1,
    double v2,
    double v3,
    double v4,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument
[in] *hDocument* specifies a PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] Parameter *PageNumber* specifies the page number into which to add the annotation.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

page

[in] Specifies the page number (Zero-based), to be displayed when the user activates this annotation.

mode

[in] Specifies the mode in which the destination page, as specified by *mode*, will be displayed, when the user views this item.

Acceptable values:

Value	Meaning
`Dest_Page`	Retain current display location and zoom. Parameters *v1*, *v2*, *v3*, and *v4* are not used.
`Dest_XYZ`	Display the page designated by *page*, with the coordinates `(v1, v2)` positioned at the top-left corner of the window and the contents of the page magnified by the factor *v3*. Parameters *v1* and *v2* are specified in points, and *v3* is specified in percentage points. Parameter *v4* is not used.
`Dest_Fit`	Display the page designated by *page*, with its contents magnified to fit the entire page within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are incompatible, use the smaller of the two, centering the page within the window. Parameters *v1*, *v2*, *v3*, and *v4* are not used.
`Dest_FitH`	Display the page designated by *page*, with the vertical coordinate *v1* (top), specified in points, positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of the page within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_FitV`	Display the page designated by *page*, with the horizontal coordinate *v1* (left), specified in points, positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire height of the page within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_FitR`	Display the page designated by *page*, with its contents magnified just enough to fit the rectangle specified by the coordinates *v1* (left), *v2* (top), *v3* (right), and *v4* (bottom) entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the rectangle within the window. Parameters *v1*, *v2*, *v3*, and *v4* are specified in points.
`Dest_FitB`	Display the page designated by *page*, with its contents magnified just enough to fit its bounding box entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are incompatible, use the smaller of the two, centering the bounding box within the window. Parameters *v1*, *v2*, *v3*, and *v4* are not used.

`Dest_FitBH`	Display the page designated by *page*, with the vertical coordinate's top positioned at the *v1* (top), specified in points, to the edge of the window and the contents of the page magnified just enough to fit the entire width of its bounding box within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_FitBV`	Display the page designated by *page*, with the horizontal coordinate left positioned at the *v1* (left), specified in points, to the edge of the window and the contents of the page magnified just enough to fit the entire height of its bounding box within the window. Parameters *v2*, *v3*, and *v4* are not used.
`Dest_Y`	Same as DST_XYZ, but specifies only `Y` coordinate (*v1*, in points), leave others parameters unchanged. Parameters *v2*, *v3*, and *v4* are not used.

v1

[in] The meaning of this parameter is dependant on the parameter *mode*.

v2

[in] The meaning of this parameter is dependant on the parameter *mode*.

v3

[in] The meaning of this parameter is dependant on the parameter *mode*.

v4

[in] The meaning of this parameter is dependant on the parameter *mode*.

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXCp_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to add a GoTo annotation to the document

    void AddGoToAnnotation(PDFDocument hDoc,  DWORD GoToPageNumber)
    {
        HRESULT hr = DS_OK;

        // Annotation rectangle

        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;
        // Common annotation information

        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Add GoTo annotation onto the first page

        hr = PXCp_AddGotoAction(hDoc, 0, &rect, GoToPageNumber, Dest_Y, 100,
100, 100, 100, &AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report the error
            ...
        }

        // done.
    }
```

3.1.9.4 PXCp_AddLaunchActionA

PXCp_AddLaunchActionA

Top Previous Next

PXCp_AddLaunchActionA specifies a rectangular area on a page that triggers execution of a specified application or document operation.

```
HRESULT  PXCp_AddLaunchActionA(
    PDFDocument hDocument,
    DWORD PageNumber,
    LPCPXC_RectF rect,
    LPCSTR lpszFileName,
    LPCSTR lpszParams,
    PXC_LaunchOperation oper,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument

[in] Specifies a PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] Specifies the page number into which to add the annotation.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

lpszFileName

[in] Pointer to a null-terminated string that specifies the full path and file name of the application to be launched or the document to opened or printed.

lpszParams

[in] Pointer to a null-terminated string that specifies a parameter string to be passed to the application as designated by the *rect* parameter. This parameter may be `NULL`.

oper

[in] Specifies the operation to perform. May be any one of the following values:

Value	Description
LO_Open	Open a document.
LO_Print	Print a document.

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXCp_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to add Launch annotation to the document

   void AddLaunchAnnotation(PDFDocument hDoc,  LPCSTR PDFFileToOpen)
   {
       HRESULT hr = DS_OK;

       // Annotation rectangle

       PXC_RectF rect;
       rect.left = 20.0;
       rect.top = 200.0;
       rect.right = 320.0;
       rect.bottom = 500.0;

       // Common annotation information

       PXC_CommonAnnotInfo AnnotInfo;
       AnnotInfo.m_Color = RGB(200, 0, 100);
```

```
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Add lunch annotation onto the first page

        hr = PXCp_AddLaunchActionA(hDoc, 0, &rect, PDFFileToOpen, NULL,
LO_Open, &AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report the error
            ...
        }

        // done.
    }
```

3.1.9.5 PXCp_AddLaunchActionW

PXCp_AddLaunchActionW

Top Previous Next

PXCp_AddLaunchActionW specifies a rectangular area on a page that triggers execution of a specified application or document operation.

Note: This function is the UNICODE equivalent to the **PXCp_AddLaunchActionA** function.

```
HRESULT  PXCp_AddLaunchActionW(
    PDFDocument hDocument,
    DWORD PageNumber,
    LPCPXC_RectF rect,
    LPCWSTR lpwszFileName,
    LPCWSTR lpwszParams,
    PXC_LaunchOperation oper,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument
[in] Specifies an PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] Specifies the page number into which to add the annotation.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

lpwszFileName

[in] Pointer to a null-terminated string that specifies the full path and file name of the application to be launched or the document to opened or printed.

lpwszParams

[in] Pointer to a null-terminated string that specifies a parameter string to be passed to the application as designated by the *rect* parameter. This parameter may be `NULL`.

oper

[in] Specifies the operation to perform. May be any one of the following values:

Value	Description
LO_Open	Open a document.
LO_Print	Print a document.

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXCp_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to add a Launch annotation to the document

    void AddLaunchAnnotation(PDFDocument hDoc,  LPCWSTR PDFFileToOpen)
    {
        HRESULT hr = DS_OK;

        // Annotation rectangle

        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Common annotation information

        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Add lunch annotation onto the first page

        hr = PXCp_AddLaunchActionW(hDoc, 0, &rect, PDFFileToOpen, NULL,
LO_Open, &AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report the error
            ...
        }

        // done.
    }
```

3.1.9.6 PXCp_AddLineAnnotationA

PXCp_AddLineAnnotationA

Top Previous Next

PXCp_AddLineAnnotationA adds a *line annotation* to the specified page. This displays as a single straight line on the page.

```
HRESULT  PXCp_AddLineAnnotationA(
    PDFDocument hDocument,
    DWORD PageNumber,
    LPCPXC_RectF rect,
    LPCSTR pszTitle,
    LPCSTR pszAnnot,
    LPCPXC_PointF pntStart,
    LPCPXC_PointF pntEnd,
    PXC_LineAnnotsType sEndStyle,
    PXC_LineAnnotsType eEndStyle,
    COLORREF cInterior,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument
[in] Specifies an PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] Specifies the page number into which to add the annotation.

rect
: [in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

pszTitle
: [in] Pointer to a null-terminated string that specifies the title of the annotation's pop-up window.

pszAnnot
: [in] Pointer to a null-terminated string that specifies the text to be displayed for the annotation.

pntStart
: [in] Pointer to a `PXC_PointF` structure that specifies the starting coordinates of the line.

pntEnd
: [in] Pointer to a `PXC_PointF` structure that specifies the ending coordinates of the line.

sEndStyle
: [in] Specifies the line ending style for the starting point of the line.

eEndStyle
: [in] Specifies the line ending style for the ending point of the line.

cInterior
: [in] Specifies interior color for line endings. See **Comments**.

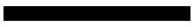
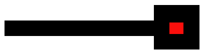
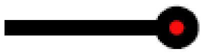
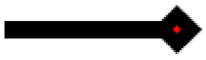
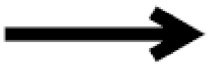
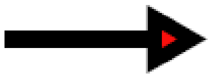
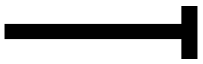
pInfo
: [in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXCp_SetAnnotsInfo**).

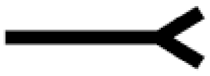
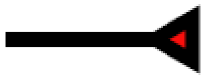
Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

Line ending styles table (`PXC_LineAnnotsType`):

Constant	Appearance	Description
`LAType_None`		No line ending.
`LAType_Square`		A square filled with the annotation's interior color.
`LAType_Circle`		A circle filled with the annotation's interior color.
`LAType_Diamond`		A diamond shape filled with the annotation's interior color.
`LAType_OpenArrow`		Two short lines meeting in an acute angle, forming an open arrowhead.
`LAType_ClosedArrow`		Two short lines meeting in an acute angle as in the `LAType_OpenArrow` style, connected by a third line to form a triangular closed arrowhead filled with the annotation's interior color.
`LAType_Butt`		(*PDF 1.5*) A short line at the endpoint perpendicular to the line itself.

`LAType_ROpenArrow`		(*PDF 1.5*) Two short lines in the reverse direction from `LAType_OpenArrow`.
`LAType_RClosedArrow`		(*PDF 1.5*) A triangular closed arrowhead in the reverse direction from `LAType_ClosedArrow`.

Example (C++).

```
// Example shows, how to add a line annotation to the document

    void AddLineAnnotation(PDFDocument hDoc)
    {
        HRESULT hr = DS_OK;

        // Common annotation information
        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Start and end points
        PXC_PointF        startPnt;
        PXC_PointF        endPnt;

        // Points data
        startPnt.x = 300.0;
        startPnt.y = 300.0;
        endPnt.x = 10.0;
        endPnt.y = 10.0;

        // Annotation rectangle
        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Add annotation
        hr = PXCp_AddLineAnnotationA(hDoc, 0, &rect, "Title", "Contents",
&startPnt, &endPnt, LAType_Square, LAType_Circle, RGB(0, 200, 150),
&AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // done.
    }
```

3.1.9.7 PXCp_AddLineAnnotationW

PXCp_AddLineAnnotationW

Top Previous Next

PXCp_AddLineAnnotationW adds a *line annotation* to the specified page. This displays as a single straight line on the page.

Note: This function is a UNICODE equivalent of the function **PXCp_AddLineAnnotationA**.

```
HRESULT  PXCp_AddLineAnnotationW(
    PDFDocument hDocument,
    DWORD PageNumber,
    LPCPXC_RectF rect,
    LPCWSTR pwszTitle,
    LPCWSTR pwszAnnot,
    LPCPXC_PointF pntStart,
    LPCPXC_PointF pntEnd,
    PXC_LineAnnotsType sEndStyle,
    PXC_LineAnnotsType eEndStyle,
    COLORREF cInterior,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument
[in] Specifies an PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] Specifies the page number into which to add the annotation.

rect
[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

pwszTitle
[in] Pointer to a null-terminated string that specifies the title of the annotation's pop-up window.

pwszAnnot
[in] Pointer to a null-terminated string that specifies the text to be displayed for the annotation.

pntStart
[in] Pointer to a `PXC_PointF` structure that specifies the starting coordinates of the line.

pntEnd
[in] Pointer to a `PXC_PointF` structure that specifies the ending coordinates of the line.

sEndStyle
[in] Specifies the line ending style for the starting point of the line.

eEndStyle
[in] Specifies the line ending style for the ending point of the line.

cInterior
[in] Specifies interior color for line endings. See **Comments**.

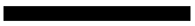
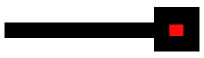
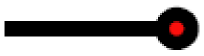
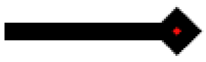
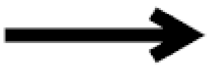
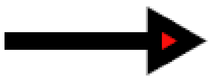
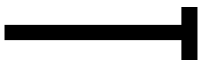
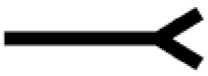
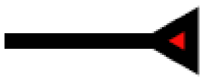
pInfo
[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXCp_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

Line ending styles table (`PXC_LineAnnotsType`):

Constant	Appearance	Description
`LAType_None`		No line ending.
`LAType_Square`		A square filled with the annotation's interior color.
`LAType_Circle`		A circle filled with the annotation's interior color.
`LAType_Diamond`		A diamond shape filled with the annotation's interior color.
`LAType_OpenArrow`		Two short lines meeting in an acute angle, forming an open arrowhead.
`LAType_ClosedArrow`		Two short lines meeting in an acute angle as in the `LAType_OpenArrow` style, connected by a third line to form a triangular closed arrowhead filled with the annotation's interior color.
`LAType_Butt`		(*PDF 1.5*) A short line at the endpoint perpendicular to the line itself.
`LAType_ROpenArrow`		(*PDF 1.5*) Two short lines in the reverse direction from `LAType_OpenArrow`.
`LAType_RClosedArrow`		(*PDF 1.5*) A triangular closed arrowhead in the reverse direction from `LAType_ClosedArrow`.

Example (C++).

```
// Example shows, how to add a line annotation to the document

    void AddLineAnnotation(PDFDocument hDoc)
    {
        HRESULT hr = DS_OK;

        // Common annotation information
        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Start and end points
        PXC_PointF        startPnt;
        PXC_PointF        endPnt;

        // Points data
        startPnt.x = 300.0;
        startPnt.y = 300.0;
        endPnt.x = 10.0;
        endPnt.y = 10.0;

        // Annotation rectangle
        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Add annotation
        hr = PXCp_AddLineAnnotationW(hDoc, 0, &rect, L"Title", L"Contents",
&startPnt, &endPnt, LAType_Square, LAType_Circle, RGB(0, 200, 150),
&AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // done.
    }
```

3.1.9.8 PXCp_AddLink

PXCp_AddLink

PXCp_AddLink adds a URL link to the specified page.

```
HRESULT  PXCp_AddLink(
    PDFDocument hDocument,
    DWORD PageNumber,
    LPCPXC_RectF rect,
    LPCSTR lpszURL,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument
[in] Specifies an PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] Specifies the page number into which to add the annotation.

rect
[in] Specifies the bounding rectangle of the link.

lpszURL
[in] Specifies the URL of the link. This parameter must be a null-terminated string.

pInfo
[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is NULL, the document's global settings will be used (for more information see **PXCp_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to add a link annotation to the document

   void AddLinkAnnotation(PDFDocument hDoc)
   {
       HRESULT hr = DS_OK;

       // Specify the rect to the annotation
       PXC_RectF rect;
       rect.left = 20.0;
       rect.top = 200.0;
       rect.right = 320.0;
       rect.bottom = 500.0;

       // Common annotation information
```

```
        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 5.0;
        AnnotInfo.m_Border.m_DashArray[1] = 10.0;
        AnnotInfo.m_Border.m_DashArray[2] = 3.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 5.0;

        // Add annotation to the first page
        hr = PXCp_AddLink(hDoc, 0, &rect, "http://www.google.com",
&AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // done.
    }
```

3.1.9.9 PXCp_AddTextAnnotationA

PXCp_AddTextAnnotationA

Top Previous Next

PXCp_AddTextAnnotationA adds a *text annotation* to the content of the PDF object. A text annotation represents a "sticky note" attached to a point in the PDF document. When closed, the annotation appears as an icon; when opened, it displays a pop-up window containing the text of the note, in a font and size chosen by the viewing application.

```
HRESULT  PXCp_AddTextAnnotationA(
    PDFDocument hDocument,
    DWORD PageNumber,
    LPCPXC_RectF rect,
    LPCSTR pszTitle,
    LPCSTR pszAnnot,
    PXC_TextAnnotsType type,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument

[in] Specifies a PDF object previously created by the function **PXCp_Init**.

PageNumber

[in] Specifies the page number into which to add the annotation.

rect

[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

pszTitle

[in] Pointer to a null-terminated string that specifies the title of the annotation's pop-up window.

pszAnnot

[in] Pointer to a null-terminated string that specifies text to be displayed for the annotation.

type

[in] Specifies the icon to be used in displaying the annotation. May be any one of the following values:

- `TAType_Note`
- `TAType_Comment`
- `TAType_Key`
- `TAType_Help`
- `TAType_NewParagraph`
- `TAType_Insert`

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the documents global settings will be used (for more information see **PXCp_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function has a UNICODE equivalent - **PXCp_AddTextAnnotationW**.

Example (C++).

```
// Example shows, how to add a text annotation to the document

   void AddTextAnnotation(PDFDocument hDoc)
   {
       HRESULT hr = DS_OK;

       // Specify the rect to the annotation
       PXC_RectF rect;
       rect.left = 20.0;
       rect.top = 200.0;
       rect.right = 320.0;
       rect.bottom = 500.0;

       // Specify annotation title
       LPCSTR pszTitle = "This is annotation title";
```

```
        // Specify annotation text
        LPCSTR pszAnnot = "This is annotation content";

        // The type of the annotation
        PXC_TextAnnotsType type = TAType_Note;

        // Common annotation information
        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;

        // Use dashed border
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 1.0;
        AnnotInfo.m_Border.m_DashArray[1] = 2.0;
        AnnotInfo.m_Border.m_DashArray[2] = 0.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 1.0;

        // Add annotation to the first page
        hr = PXCp_AddTextAnnotationA(hDoc, 0, &rect, pszTitle, pszAnnot, type,
&AnnotInfo);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // done.
    }
```

3.1.9.10 PXCp_AddTextAnnotationW

PXCp_AddTextAnnotationW

Top Previous Next

PXCp_AddTextAnnotationW adds a *text annotation* to the content of the PDF object. A text annotation represents a "sticky note" attached to a point in the PDF document. When closed, the annotation appears as an icon; when opened, it displays a pop-up window containing the text of the note, in a font and size chosen by the PDF viewing application.

Note: This function is a UNICODE equivalent of the function **PXCp_AddTextAnnotationA**.

```
HRESULT  PXCp_AddTextAnnotationW(
    PDFDocument hDocument,
    DWORD PageNumber,
    LPCPXC_RectF rect,
    LPCWSTR pwszTitle,
    LPCWSTR pwszAnnot,
    PXC_TextAnnotsType type,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument
[in] Specifies a PDF object previously created by the function **PXCp_Init**.

PageNumber
[in] Specifies the page number into which to add the annotation.

rect
[in] Pointer to a `PXC_RectF` structure that specifies the bounding rectangle of the annotation.

pwszTitle
[in] Pointer to a null-terminated string that specifies the title of the annotation's pop-up window.

pwszAnnot
[in] Pointer to a null-terminated string that specifies the text to be displayed for the annotation.

type
[in] Specifies the icon to be used in displaying the annotation. May be any one of the following values:

- `TAType_Note`
- `TAType_Comment`
- `TAType_Key`
- `TAType_Help`
- `TAType_NewParagraph`
- `TAType_Insert`

pInfo
[in] Pointer to a **PXC_CommonAnnotInfo** structure that describes attributes of the annotation. If this parameter is `NULL`, the document's global settings will be used (for more information see **PXCp_SetAnnotsInfo**).

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to add a text annotation to the document

    void AddTextAnnotation(PDFDocument hDoc)
    {
        HRESULT hr = DS_OK;

        // Specify the rect to the annotation
        PXC_RectF rect;
        rect.left = 20.0;
        rect.top = 200.0;
        rect.right = 320.0;
        rect.bottom = 500.0;

        // Specify annotation title
        LPCWSTR pwszTitle = L"This is annotation title";

        // Specify annotation text
        LPCWSTR pwszAnnot = L"This is annotation content";

        // The type of the annotation
        PXC_TextAnnotsType type = TAType_Note;

        // Common annotation information
        PXC_CommonAnnotInfo AnnotInfo;
        AnnotInfo.m_Color = RGB(200, 0, 100);
        AnnotInfo.m_Flags = 0;
        AnnotInfo.m_Opacity = 1.0;

        // Use dashed border
        AnnotInfo.m_Border.m_DashArray = new double[3];
        AnnotInfo.m_Border.m_DashArray[0] = 1.0;
        AnnotInfo.m_Border.m_DashArray[1] = 2.0;
        AnnotInfo.m_Border.m_DashArray[2] = 0.5;
        AnnotInfo.m_Border.m_DashCount = 3;
        AnnotInfo.m_Border.m_Type = ABS_Dashed;
        AnnotInfo.m_Border.m_Width = 1.0;

        // Add annotation to the first page
        hr = PXCp_AddTextAnnotationW(hDoc, 0, &rect, pwszTitle, pwszAnnot,
type, &AnnotInfo);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // done.
    }
```

3.1.9.11 PXCp_SetAnnotsInfo

PXCp_SetAnnotsInfo

Top Previous Next

PXCp_SetAnnotsInfo sets the general annotation parameters (color, border type and width etc.) for the document.

These general parameters are used when the appropriate parameter for the functions **PXCp_AddLink**,

PXCp_AddLineAnnotationW, **PXCp_AddLaunchActionW**, **PXCp_AddTextAnnotationW** have a value of `NULL`.

```
HRESULT  PXCp_SetAnnotsInfo(
    PDFDocument hDocument,
    const PXC_CommonAnnotInfo* pInfo
);
```

Parameters

hDocument

[in] Specifies a PDF object previously created by the function **PXCp_Init**.

pInfo

[in] Pointer to a **PXC_CommonAnnotInfo** structure describing attributes of the annotation.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to add several links
// using the same common annotation information to the document

   void AddSeveralLinks(PDFDocument hDoc,  LPCSTR LinkURL, DWORD LinkCount)
   {
       HRESULT hr = DS_OK;

       // Specify the 'start' rect to the annotation

       PXC_RectF rect;
       rect.left = 20.0;
       rect.top = 200.0;
       rect.right = 320.0;
       rect.bottom = 250.0;

       // Common annotation information

       PXC_CommonAnnotInfo AnnotInfo;
       AnnotInfo.m_Color = RGB(200, 0, 100);
       AnnotInfo.m_Flags = 0;
       AnnotInfo.m_Opacity = 1.0;
       AnnotInfo.m_Border.m_DashArray = new double[3];
       AnnotInfo.m_Border.m_DashArray[0] = 5.0;
       AnnotInfo.m_Border.m_DashArray[1] = 10.0;
       AnnotInfo.m_Border.m_DashArray[2] = 3.5;
       AnnotInfo.m_Border.m_DashCount = 3;
       AnnotInfo.m_Border.m_Type = ABS_Dashed;
       AnnotInfo.m_Border.m_Width = 5.0;
```

```
        // Set common information for all further annotation functions

        hr = PXCp_SetAnnotsInfo(hDoc, &AnnotInfo);

        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // Now place several links on to the page

        for (DWORD lc = 0; lc < LinkCount; lc++)
        {
            // Place link
            // the last argument is NULL as the 'global' annotation
information is used

            hr = PXCp_AddLink(hDoc, 0, &rect, LinkURL, NULL);
            if (IS_DS_FAILED(hr))
            {
                // report error
                ...
            }

            // Now 'move' the rect of the link

            rect.top += 51;
            rect.bottom = rect.top + 50;
        }

        // done.
    }
```

3.1.10 Text Extraction

3.1.10.1 PXCp_ET_AnalyzePageContent

PXCp_ET_AnalyzePageContent

Top Previous Next

PXCp_ET_AnalyzePageContent analyzes the content of the specified page and collects information about each portion of text on the page. All text element information is stored in an internal data structure and may be accessed using the function **PXCp_ET_GetElement**.

```
HRESULT  PXCp_ET_AnalyzePageContent(
    PDFDocument pDocument,
    DWORD pageNum
);
```

Parameters

pDocument
[in] *pDocument* specifies a PDF object previously created by the function **PXCp_Init**.

pageNum
[in] *pageNum* specifies zero-based page index.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
    // Example shows how to extract all text from the document
    // and save it to another document retaining formating

    void ExtractTextToOtherPDFDocument(PDFDocument hDoc, LPCWSTR
OtherPDFFileName)
    {
        HRESULT hr = DS_OK;

        // Preparsing document

        hr = PXCp_ET_Prepare(hDoc);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        DWORD        fontCount = 0;
        DWORD*        fontIDs = NULL;
        PXCDocument hDstDoc = NULL;

        do
        {
            // 1. Get all fonts from the doc
            hr = PXCp_ET_GetFontCount(hDoc, &fontCount);
            if (IS_DS_FAILED(hr))
                break;
            // 2. Create new doc
            hr = PXC_NewDocument(&hDstDoc, NULL, NULL);
            if(IS_DS_FAILED(hr))
                break;
            fontIDs = new DWORD[fontCount];
            ::ZeroMemory(fontIDs, fontCount * sizeof(DWORD));

            // buffer for font name + font style
            LPWSTR        fontName = NULL;
            DWORD        curFontNameLen = 0;

            for (DWORD i = 0; i < fontCount; i++)
            {
                DWORD bufLen = 0;
                // get font name length
                // if it is equal to 1 then there is no font name
                // "1" is null-terminator in this case
                hr = PXCp_ET_GetFontName(hDoc, i, NULL, &bufLen);
                if (IS_DS_FAILED(hr))
                    break;
                // Check if the font have any name set
                BOOL bNoFontNameSet = bufLen <= 1;
                // if there is no font name default 'Arial' will be used
                static LPCWSTR DefaultFontName = L"Arial";
                if (bNoFontNameSet)
                    bufLen = 6; // ::lstrlenW(DefaultFontName)
                // Get the length of font style
                DWORD StyleLen = 0;
                hr = PXCp_ET_GetFontStyle(hDoc, i, NULL, &StyleLen);
                // Check if there is font style s°õ
                if (StyleLen <= 1)
                    StyleLen = 0;
                if (IS_DS_SUCCESSFUL(hr) && StyleLen)
                {
                    // if there is font style set - then adjust the buffer
    length
                    bufLen += StyleLen;
                }
                // Check for necessary buffer
                if (bufLen > curFontNameLen)
                {
                    if (fontName)
                        delete[] fontName;
                    fontName = new WCHAR[bufLen];
                    curFontNameLen = bufLen;
                }
                if (bNoFontNameSet)
                {
                    // if there is no file name - copy default name
                    ::lstrcpy(fontName, DefaultFontName);
                }
                else
                {
                    // else aquire font name from the library
                    DWORD tempBufLen = bufLen;
                    hr = PXCp_ET_GetFontName(hDoc, i, fontName, &tempBufLen);
                    if (IS_DS_FAILED(hr))
```

```
                        break;
                }
                if (StyleLen)
                {
                    // if there is font style set - aquire it
                    hr = PXCp_ET_GetFontStyle(hDoc, i, fontName + (bufLen -
StyleLen) - 1, &StyleLen);
                }
                // add the font into library
                hr = PXC_AddFontW(hDstDoc, FW_NORMAL, FALSE, fontName, fontIDs
+ i);
                if (IS_DS_FAILED(hr))
                    break;
            }
            // clean unnecessary buffer
            if (fontName)
            {
                delete[] fontName;
                fontName = NULL;
            }
            if (IS_DS_FAILED(hr))
            {
                break;
            }
            DWORD PageCnt = 0;
            hr = PXCp_GetPagesCount(hDoc, &PageCnt);
            if (IS_DS_FAILED(hr) || !PageCnt)
                break;
            // 3. for each page
            for (DWORD CurPage = 0; CurPage < PageCnt; CurPage++)
            {
                // create new page in the new document
                PXC_RectF rcMediaBox;
                PXC_RectF rcCropBox;
                LONG nAngle;

                hr = PXCp_PageGetBox(hDoc, CurPage, PB_MediaBox, &rcMediaBox);
                if(IS_DS_FAILED(hr))
                    break;
                // add to the new page
                PXCPage hDstPage = NULL;
                hr = PXC_AddPage(hDstDoc, rcMediaBox.right - rcMediaBox.left,
rcMediaBox.top - rcMediaBox.bottom, &hDstPage);
                if(IS_DS_FAILED(hr))
                    break;

                hr = PXCp_PageGetBox(hDoc, CurPage, PB_CropBox, &rcCropBox);
                if(IS_DS_SUCCESSFUL(hr))
                {
                    hr = PXC_SetPageBox(hDstPage, PB_CropBox, &rcCropBox);
                }
                hr = PXCp_PageGetRotate(hDoc, CurPage, &nAngle);
                if(IS_DS_SUCCESSFUL(hr) && nAngle)
                {
                    hr = PXC_SetPageRotation(hDstPage, nAngle);
                }
                PXC_TextOptions pto = { sizeof(PXC_TextOptions) };
                PXC_GetTextOptions(hDstPage, &pto);
                pto.nTextPosition = TextPosition_Baseline;
                PXC_SetTextOptions(hDstPage, &pto);
                //        for each element
                hr = PXCp_ET_AnalyzePageContent(hDoc, CurPage);
                if(IS_DS_FAILED(hr))
                    break;

                DWORD TextElCount = 0;
                hr = PXCp_ET_GetElementCount(hDoc, &TextElCount);
                if(IS_DS_FAILED(hr) || TextElCount == 0)
                    continue;

                PXP_TextElement TextElement = {0};
                TextElement.cbSize = sizeof(PXP_TextElement);
                DWORD CurCount = 0;

                PXC_PointF ptTextOrg = {0};
                WCHAR buf[2];
                buf[0] = buf[1] = 0;

                for (DWORD t = 0; t < TextElCount; t++)
                {
                    TextElement.Count = 0;
                    TextElement.mask = 0;
                    hr = PXCp_ET_GetElement(hDoc, t, &TextElement, 0);
                    if(IS_DS_FAILED(hr) || (LONG)TextElement.Count <= 0)
                        continue;
                    TextElement.mask = PTEM_Text | PTEM_Offsets | PTEM_Matrix
|        PTEM_FontInfo | PTEM_TextParams;
                    if (CurCount < TextElement.Count)
                    {
                        if (TextElement.Characters != NULL)
                            delete TextElement.Characters;
                        if (TextElement.Offsets != NULL)
                            delete TextElement.Offsets;
                        TextElement.Characters = new WCHAR[TextElement.Count];
                        TextElement.Offsets = new double[TextElement.Count];
                        CurCount = TextElement.Count;
                    }
                    hr = PXCp_ET_GetElement(hDoc, t, &TextElement,
GTEF_IgnorePageRotation);
                    if (IS_DS_FAILED(hr))
                        continue;
                    // Now add this text element into new PDF document
                    hr = PXC_TCS_Transform(hDstPage, &TextElement.Matrix);

                    if (fontCount <= TextElement.FontIndex)
                        continue;

                    hr = PXC_SetCurrentFont(hDstPage, fontIDs[TextElement.
FontIndex], TextElement.FontSize);
                                    hr = PXC_SetFillColor(hDstPage,
TextElement.FillColor);
                    hr = PXC_SetStrokeColor(hDstPage, TextElement.
StrokeColor);
                    hr = PXC_SetTextRMode(hDstPage, TextElement.RenderingMode,
NULL);
                    hr = PXC_SetTextScaling(hDstPage, TextElement.Th, NULL);
                    hr = PXC_SetTextLeading(hDstPage, TextElement.Leading,
NULL);
                    hr = PXC_SetCharSpacing(hDstPage, TextElement.CharSpace,
NULL);
                    hr = PXC_SetWordSpacing(hDstPage, TextElement.WordSpace,
NULL);

                    for(DWORD j = 0; j < TextElement.Count - 1; j++)
                    {
                        ptTextOrg.x = TextElement.Offsets[j];
                        buf[0] = TextElement.Characters[j];
                        hr = PXC_TextOutW(hDstPage, &ptTextOrg, buf, 1);
                    }
                }
                if (TextElement.Characters != NULL)
                    delete TextElement.Characters;
                if (TextElement.Offsets != NULL)
                    delete TextElement.Offsets;
            }
            if (IS_DS_FAILED(hr))
                break;

            hr = PXC_WriteDocumentExW(hDstDoc, OtherPDFFileName, -1,
WEF_ShowSaveDialog | WEF_RunApp, NULL);

        } while(FALSE);

        // clear up
        if (hDstDoc)
        {
            PXC_ReleaseDocument(hDstDoc);
            hDstDoc = NULL;
        }
        if (fontIDs)
        {
            delete[] fontIDs;
            fontIDs = NULL;
        }
        PXCp_ET_Finish(hDoc);
    }
```

3.1.10.2 PXCp_ET_Finish

PXCp_ET_Finish

Top Previous Next

PXCp_ET_Finish is used to free internal data and temporary buffers used during text extraction processing.

```
HRESULT  PXCp_ET_Finish(
    PDFDocument hDocument
);
```

Parameters

hDocument

[in] *hDocument* specifies a PDF object previously created by **PXCp_Init**.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

This function should be called to correctly terminate text extraction processing and free all temporary data. No other text extraction functions should be called subsequently, except for **PXCp_ET_Prepare** should you need to restart text extraction.

Example (C++).

```
// Each text extraction ends with the 'Finish' functions call

    void DoSomeExtractTextStuff(PDFDocument hDoc)
    {
        HRESULT hr = DS_OK;

        ...

        // Now finish with the text extract

        hr = PXCp_ET_Finish(hDoc);
    }
```

3.1.10.3 PXCp_ET_GetCurrentComposeParams

PXCp_ET_GetCurrentComposeParams

Top Previous Next

PXCp_ET_GetCurrentComposeParams receive current text composition options.

```
HRESULT  PXCp_ET_GetCurrentComposeParams(
     PDFDocument pDocument,
     PXP_TETextComposeOptions* pOptions
);
```

Parameters

pDocument

[in] *pDocument* specifies the PDF object previously created by **PXCp_Init**.

pOptions

[out] *pOptions* specifies a pointer to a **PXP_TETextComposeOptions** structure to receive the composition information.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to retrieve current composition parameters

    PDFDocument hDoc;

    ...

    // Prepare PXP_TextElement structure

    PXP_TETextComposeOptions        txtComposeOpts;

    // Get text compose options

    HRESULT hr = PXCp_ET_GetCurrentComposeParams(hDoc, &txtComposeOpts);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }

    // Now txtComposeOpts.contains current composition parameters
```

```
...
```

3.1.10.4 PXCp_ET_GetElement

PXCp_ET_GetElement

PXCp_ET_GetElement retrieves information about the specified text element, including character positioning, text color, etc.

```
HRESULT  PXCp_ET_GetElement(
    PDFDocument pDocument,
    DWORD index,
    PXP_TextElement* pElement,
    DWORD flags
);
```

Parameters

pDocument
[in] *pDocument* specifies a PDF object previously created by **PXCp_Init**.

index
[in] *index* specifies zero-based element index.

pElement
[out] *pElement* specifies a pointer to a **PXP_TextElement** structure, which will receive the text element information.

flags
[in] *flags* specifies additional flags which determine how information in **PXP_TextElement** structure should be handled. it may be any combination of following flags:

Constant	Value	Meaning
GTEF_OriginalCodes	0x0001	Retrieve original character indexes instead of unicode values.
GTEF_OriginalDeltas	0x0002	Retrieve only additional deltas between characters instead of character positions from begining of the element. Note that instead of offsets additional deltas are in unscaled text units (see PDF Reference for details).
GTEF_IgnorePageRotation	0x0004	Do not include page rotation in the element matrix if the page is rotated.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to retrieve the text element information
// i.e. get text of the element

    PDFDocument hDoc;

    ...

    // Prepare PXP_TextElement structure

    PXP_TextElement TextElement = {0};
    TextElement.cbSize = sizeof(PXP_TextElement);
    TextElement.Count = 0;
    TextElement.mask = 0;

    // Retrieve number of characters in the element

    hr = PXCp_ET_GetElement(hDoc, t, &TextElement, 0);

    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }

    // Prepare buffer for the text

    TextElement.Characters = new WCHAR[TextElement.Count];

    // Retrive only text

    TextElement.mask = PTEM_Text;

    // Retrive element's text

    hr = PXCp_ET_GetElement(hDoc, t, &TextElement, GTEF_IgnorePageRotation);

    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }

    // Now the TextElement.Characters contains the element's text
    // Do not forget to free alocated buffer!

    ...
```

3.1.10.5 PXCp_ET_GetElementCount

PXCp_ET_GetElementCount

Top Previous Next

PXCp_ET_GetElementCount retrieves the count of text elements found by a prior call to **PXCp_ET_AnalyzePageContent**.

```
HRESULT  PXCp_ET_GetElementCount(
    PDFDocument pDocument,
    DWORD* count
);
```

Parameters

pDocument
: [in] *pDocument* specifies a PDF object previously created by **PXCp_Init**.

count
: [out]*count* specifies a pointer to a `DWORD` variable to receive the count of text elements found on the page.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to retrieve the number of text elements on the page

    DWORD GetTextElementsCountOnPage(PDFDocument hDoc, DWORD PageNumber)
    {
        HRESULT hr = DS_OK;

        // Preparsing document

        hr = PXCp_ET_Prepare(hDoc);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // Analyze page content

        hr = PXCp_ET_AnalyzePageContent(hDoc, PageNumber);

        DWORD TextElementsCount = 0;

        // Get number of text elements on page

        hr = PXCp_ET_GetElementCount(hDoc, &TextElementsCount);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // Finish and return the number in the document

        PXCp_ET_Finish(hDoc);
        return TextElementsCount;
    }
```

3.1.10.6 PXCp_ET_GetFontCount

PXCp_ET_GetFontCount

Top Previous Next

PXCp_ET_GetFontCount retrieves the number of fonts in the PDF document. **PXCp_ET_GetFontCount** may be used after successful call to **PXCp_ET_Prepare** function.

```
HRESULT  PXCp_ET_GetFontCount(
    PDFDocument pDocument,
    DWORD* count
);
```

Parameters

pDocument

[in] *pDocument* specifies a PDF object previously created by **PXCp_Init**.

count

[out] *count* specifies the pointer to the value of the `DWORD` variable which will receive the count of the fonts in the document.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieves the number of fonts in the PDF document

    DWORD GetNumberOfFonts(PDFDocument hDoc)
    {
        HRESULT hr = DS_OK;

        // Preparsing document

        hr = PXCp_ET_Prepare(hDoc);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        DWORD FontCount = 0;

        // Get number of all fonts from the doc

        hr = PXCp_ET_GetFontCount(hDoc, &FontCount);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // Finish and return the number of fonts in the document

        PXCp_ET_Finish(hDoc);
        return FontCount;
    }
```

3.1.10.7 PXCp_ET_GetFontInfo

PXCp_ET_GetFontInfo

Top Previous Next

PXCp_ET_GetFontInfo retrieves information about a PDF font specified by its index (the number of fonts may be obtained using **PXCp_ET_GetFontCount** function).

```
HRESULT  PXCp_ET_GetFontInfo(
    PDFDocument pDocument,
    DWORD index,
    PXP_TEFontInfo* pInfo
);
```

Parameters

pDocument
[in] *pDocument* specifies an PDF object previously created by the function **PXCp_Init**.

index
[in] *index* specifies the index of the font.

pInfo
[out] *pInfo* specifies the pointer to the **PXP_TEFontInfo** structure that receives the information about the specified font.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.

To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to retrieve information about a font

    PDFDocument        hDoc;
    DWORD                FontIndex;

    ...

    // Prepare PXP_TextElement structure

    PXP_TEFontInfo        FontInfo = {0};
    FontInfo.cbSize = sizeof(PXP_TextElement);

    // Retrieve information about the specified font

    hr = PXCp_ET_GetFontInfo(hDoc, FontIndex, &FontInfo);

    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }

    // Now FontInfo structure contains the information about the font

    ...
```

3.1.10.8 PXCp_ET_GetFontName

PXCp_ET_GetFontName

Top Previous Next

PXCp_ET_GetFontName retrieves the name of the PDF font.

```
HRESULT  PXCp_ET_GetFontName(
    PDFDocument pDocument,
    DWORD index,
    LPWSTR name,
    DWORD* length
);
```

Parameters

pDocument

[in] *pDocument* specifies a PDF object previously created by **PXCp_Init**.

index

[in] *index* specifies the index of the font.

name

[in, out] *name* specifies a pointer to a buffer where the name of the font should be inserted. \

Note: To determine the required buffer size you can pass `NULL` as *name*.

length

[in, out] *length* specifies an available buffer size in characters (including a null-terminating character).

Note: When *name* is set to `NULL` then *length* will contain the required buffer size

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

Total number of fonts in PDF document could be obtained be **PXCp_ET_GetFontCount**.

Example (C++).

```
// Example shows, how to retrieve a font name by its index

    ...

    // Buffer and it's length for the name of the font

    LPWSTR         FontName = NULL;
    DWORD         FontNameLen = 0;

    // Retrive the necesary buffer length

    HRESULT hr = PXCp_ET_GetFontName(hDoc, FontIndex, NULL, &FontNameLen);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }

    // Allocate the buffer

    FontName = new WCHAR[FontNameLen];

    // Retrive Font name

    hr = PXCp_ET_GetFontName(hDoc, FontIndex, FontName, &FontNameLen);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
```

3.1.10.9 PXCp_ET_GetFontObj

PXCp_ET_GetFontObj

Top Previous Next

PXCp_ET_GetFontObj retrieves the PDF object that corresponds to the specified font.

```
HRESULT  PXCp_ET_GetFontObj(
    PDFDocument pDocument,
    DWORD index,
    HPDFOBJECT* phObject
);
```

Parameters

pDocument
[in] *pDocument* specifies a PDF object previously created by **PXCp_Init**.

index
[in] *index* specifies the index of the font.

phObject
[out] *phObject* specifies the pointer to a variable of the `HPDFOBJECT` type that receives the object's handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to get PDF object that corresponds to the font

    ...

    HPDFOBJECT hFontObject;

    HRESULT hr = PXCp_ET_GetFontObj(hDoc, FontIndex, &hFontObject);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }

    // Now hFontObject is the valid handle of the PDF object
    // and could be used to edit the object directly
```

3.1.10.10 PXCp_ET_GetFontStyle

PXCp_ET_GetFontStyle

Top Previous Next

PXCp_ET_GetFontStyle retrieves the specified PDF font's style.

```
HRESULT  PXCp_ET_GetFontStyle(
    PDFDocument pDocument,
    DWORD index,
    LPWSTR style,
    DWORD* length
);
```

Parameters

pDocument
[in] *pDocument* specifies a PDF object previously created by **PXCp_Init**.

index
[in] *index* specifies the index of the font.

style
[in, out] *style* specifies a pointer to a buffer where the style of the font should be inserted.
Note: To determine the required buffer size you should pass `NULL` as *style*.

length
[in, out] *length* specifies an available buffer size in characters (including a null-terminating character).
Note: When *style* is set to `NULL` then *length* will contain the required buffer size

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to retrieve a font style by its index

    ...

    // Buffer and it's length for the style of the font

    LPWSTR        FontStyle = NULL;
    DWORD        FontStyleLen = 0;

    // Retrive the necesary buffer length

    HRESULT hr = PXCp_ET_GetFontStyle(hDoc, FontIndex, NULL, &FontStyleLen);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    // Allocate the buffer

    FontStyle = new WCHAR[FontStyleLen];

    // Retrive Font style

    hr = PXCp_ET_GetFontStyle(hDoc, FontIndex, FontStyle, &FontStyleLen);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
```

3.1.10.11 PXCp_ET_GetPageContentAsTextW

PXCp_ET_GetPageContentAsTextW

Top Previous Next

PXCp_ET_GetPageContentAsTextW recive all text from specified page as one block.

```
HRESULT  PXCp_ET_GetPageContentAsTextW(
    PDFDocument pDocument,
    DWORD pageNum,
    PXP_TETextComposeOptions* pOptions,
    LPCWSTR* buffer,
    DWORD* length
);
```

Parameters

pDocument
: [in] *pDocument* specifies the PDF object previously created by **PXCp_Init**.

pageNum
: [in] *pageNum* specifies zero-based page index.

pOptions
: [in] *pageNum* specifies a pointer to **PXP_TETextComposeOptions** structure. If this parameter is `NULL` then use the last specified options.

buffer
: [out]*buffer* specifies a pointer to a variable to receive the pointer to a read-only buffer containing all text from page, including the terminating null character.
Note that sometimes text may contain null characters inside, so refer to the `length` parameter to obtain the actual text length.

length
: [in/out]*length* specifies a pointer to variable which receives the length of *buffer* in characters (not bytes), including any terminating null character. As this is a Unicode string, the length in bytes

should be twice this value.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
    // Example shows how to extract all text from the document
    // and save it to the text file specified

void ExtractAllTextToFile(PDFDocument hDoc, LPCWSTR TxtFileName)
{
    HRESULT hr = DS_OK;

    // Preparsing document

    hr = PXCp_ET_Prepare(hDoc);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }

    // Buffer for the text and it's length
    DWORD           bufLen = 0;
    LPCWSTR          wBuf = NULL;

    // Compose options

    PXP_TETextComposeOptions        txtComposeOpts;

    // Get text composition options

    hr = PXCp_ET_GetCurrentComposeParams(hDoc, &txtComposeOpts);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }

    // Change some options

    txtComposeOpts.MinAddSpaceDistance = 0.5;
    txtComposeOpts.ComposeMethod = TETCM_PreservePositions;

    // retrive total page count in the document

    DWORD PageCnt = 0;
    PXCp_GetPagesCount(hDoc, &PageCnt);
```

```
        // Create file for the text

        HANDLE hTxtFile = ::CreateFileW(TxtFileName, GENERIC_WRITE,
FILE_SHARE_READ, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hTxtFile == INVALID_HANDLE_VALUE)
        {
            // report error
            ...
        }

        // Prepare for write

        DWORD written = 0;

        // Begining of the UNICODE text file

        BYTE w[2];
        w[0] = 0xff;
        w[1] = 0xfe;

        // write the header

        ::WriteFile(hTxtFile, w, 2, &written, NULL);

        // The loop for all pages

        for (DWORD i = 0; i < PageCnt; i++)
        {

            // Get text conten of the 'i' page

            hr = PXCp_ET_GetPageContentAsTextW(hDoc, i, &txtComposeOpts,
&wBuf, &bufLen);
            if (IS_DS_FAILED(hr))
            {
                // report error
                ...
            }

            // Write obtained text to the file

            ::WriteFile(hTxtFile, wBuf, bufLen * 2, &written, NULL);
        }

        // Finish with text etraction

        PXCp_ET_Finish(hDoc);

        // Close the handle of the text file
        ::CloseHandle(hTxtFile);

        // done.
    }
```

3.1.10.12 PXCp_ET_Prepare

PXCp_ET_Prepare

Top Previous Next

PXCp_ET_Prepare analyses the document structure and stores the collected information in an internal data storage area, which may then be used during actual text extraction. Be aware that this function should be called before any other functions related to text extraction, and must be be followed by calling **PXCp_ET_Finish** to free the internal data storage and temporary buffers.

```
HRESULT  PXCp_ET_Prepare(
    PDFDocument hDocument
);
```

Parameters

hDocument
: [in] *hDocument* specifies a PDF object previously created by **PXCp_Init**.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Comments

Mixing text extraction function calls with other XCPRO40 functions which may modify the PDF document content is **strongly discouraged** as unpredictable behavior may occur. Using functions which just retrieve information from the document without modification is permissible.

Example (C++).

```
// Each text extraction begins with preparsing

    void DoSomeExtractTextStuff(PDFDocument hDoc)
    {
        HRESULT hr = DS_OK;

        // Preparsing document

        hr = PXCp_ET_Prepare(hDoc);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        // Do the 'real' work next
        ...

    }
```

3.2 Low-Level API

Low-Level API

PDF-Tools (XCPRO40.DLL) Low-Level API

N.B. PLEASE! YOU HAVE BEEN WARNED! THIS PORTION OF THE XCPRO40 LIBRARY IS FOR ADVANCED USERS ONLY!!!!

An existing license for the PDF-Tools SDK allows evaluation access - but not live access to PDF generation using the Low-Level API functions, only developer's who have licensed the PDF-XChange 'PRO' SDK will have access for 'live' use, using their existing Version 3 licensing info. When requesting support regarding the Low-Level API functions you will be expected to have a thorough working knowledge of the PDF format and the associated documentation from Adobe - we regret we cannot and will not 'educate' developers in this vast and complex subject.

The Low-Level API is a set of powerful functions allowing developers to access the very heart of a PDF page/ file and make changes of the most fundamental nature - with this power comes a price, indiscriminate use of the Low-Level API functions will undoubtedly lead to file corruption and problems - these functions are not to be used in haste and we accept no responsibility whatsoever for developers use of these functions - use with caution !

The High level API functions have been provided as an alternative option for most functionality, specifically for developers who are not intimately familiar with the PDF format, structure and Adobe Documentation. We strongly reccomend that you use the High Level API functions if you do not feel your knowledge/skill set allows you to use the Low-Level API functions with confidence.

Consequently this documentation is intentionally limited in depth of content and not intended to impart extensive knowledge of the PDF format and its structure - see the Adobe PDF reference documentation for detailed information.

General Description of the Structure of a PDF document
A PDF document consists of a set of PDF objects of varying types (text, Images, Lines, etc) that represent different constituent parts of that document - here is a very brief and far from comprehensive outline of the way a PDF page/file is structured and works.

It is possible to refer to a PDF object from other objects specifying its ID number. Reading a PDF document begins with the document trailer also known as the dictionary. A Dictionary contains links to a PDF object known as the Catalog or Root object. Additionaly there are links to objects that represent general information about the PDF document (for example Title, Author, Creator etc), information about any document encryption and where to locate other specific objects.

Beginning with the Catalog object a PDF document can be regarded as a hierarchy of objects contained in the body section of a PDF file. At the root of this hierarchy is the document’s catalog dictionary. Most of the objects in the hierarchy are other dictionaries. For example, each page of a document is represented by a page object- a dictionary that includes references to the page contents and attributes, such as thumbnail image's and any annotations associated with it.

The individual page objects are tied together in a structure called the page tree, which in turn is located via an indirect reference within the document catalog. Parent, child, and sibling relationships within the hierarchy are defined and managed by dictionary entries whose values are indirect references to other dictionaries.

The PDF format supports eight basic and fundamental object types:

- **Boolean values.**

  PDF provides Boolean objects identified by the keywords true and false. Boolean objects can be used as the values of array elements and dictionary entries
- **Integer and real numbers**.

  PDF provides two types of numeric object: integer and real. Integer objects represent mathematical integers within a certain interval centered at 0. Real objects approximate mathematical real numbers, but with limited range and precision; they are typically represented in fixed-point, rather than floating-point, form. The range and precision of numbers are limited by the internal representations used in the machine on which the PDF viewer application is running.
- **Strings**.

  A string object consists of a series of bytes—unsigned integer values in the range 0 to 255. The string elements are not integer objects, but are stored in a more compact format. The length of a string is subject to an implementation limit.

  Not all strings may be converted into ASCII or UNICODE format. i.e. image palette values are usually stored in a PDF document as a string.

  In PDF-XChange Pro Library strings are represented by the **HPDFSTRING** handle.
- **Names**.

  A name object is an atomic symbol uniquely defined by a sequence of characters. Uniquely defined means that any two named objects made up of the same sequence of characters are identically the same object. Atomic means that a name has no internal structure; although it is defined by a sequence of characters, those characters are not “elements” of the name.
- **Arrays**.

  An array object is a one-dimensional collection of objects arranged sequentially. Unlike arrays in many other computer languages, PDF arrays may be heterogeneous; that is, an array’s elements may be any combination of numbers, strings, dictionaries, or any other objects, including other arrays. The number of elements in an array is subject to an implementation limit.

  In PDF-XChange Pro Library arrays are represented by the **HPDFARRAY** handle.
- **Dictionaries**.

  A dictionary object is an associative table containing pairs of objects, known as the dictionary’s entries. The first element of each entry is the key and the second element is the value. The key must be a name. The value can be any kind of object, including another dictionary. A dictionary entry whose value is null is equivalent to an absent entry. The number of entries in a dictionary is subject to an implementation limit.

  In PDF-XChange Pro Library dictionaries are represented by the **HPDFDICTIONARY** handle.

  Note: No two entries in the same dictionary should have the same key. If a key does appear more than once, its value is considered undefined.
- **Streams**.

A stream object, like a string object, is a sequence of bytes. However, a PDF application can read a stream incrementally, while a string must be read in its entirety. Furthermore, a stream can be of unlimited length, whereas a string is subject to an implementation limit. For this reason, objects with potentially large amounts of data, such as images and page descriptions, are represented as streams.

All streams must be indirect objects and the stream dictionary must be a direct object.

In PDF-XChange Pro Library streams are represented by **HPDFSTREAM** handle.

- **The null object**.

The null object has a type and value that are unequal to those of any other object. There is only one object of the type null, denoted by the keyword null. An indirect object reference to a nonexistent object is treated the same as a null object; specifying the null object as the value of a dictionary entry is the equivalent to omitting the entry entirely.

Objects may be labeled so that they can be referred to by other objects. A labeled object is called an indirect object.

PDF-XChange Pro Library uses its own type of variable - the PDF variant. This is specified by the handle HPDFVARIANT which represents the afore mentioned PDF objects. For the corresponding type of PDF variant see the PXCp_VariantGetType function description.

API functions that deal with the PDF variant begin with the **PXCp_Variant...** prefix.

3.2.1 General Functions

3.2.1.1 PXCp_llGetDocTrailer

PXCp_llGetDocTrailer Top Previous Next

PXCp_llGetDocTrailer retrieves a PDF document trailer.

For additional detailed information regarding a document trailer and the PDF format file structure please refer to Adobe's PDF format Reference documentation.

```
HRESULT  PXCp_llGetDocTrailer(
    PDFDocument hDocument,
    HPDFDICTIONARY* pDict
);
```

Parameters

hDocument
[in] *hDocument* specifies the PDF object previously created by the function **PXCp_Init**.

pDict
[out] specifies a pointer to the `HPDFDICTIONARY` which receives the trailer dictionary.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieves the dictionary which corresponds to the trailer of the document
```

```
PDFDocument hDoc

...

HPDFDICTIONARY hTrailer;

hr = PXCp_llGetDocTrailer(hDoc, &hTrailer);
if (IS_DS_FAILED(hr))
{
    // report error
    ...
}
...
```

3.2.1.2 PXCp_llGetObjectByIndex

PXCp_llGetObjectByIndex

Top Previous Next

PXCp_llGetObjectByIndex retrieves a PDF object as specified by its index value. The number of objects may be obtained by using the **PXCp_llGetObjectsCount** function.

```
HRESULT  PXCp_llGetObjectByIndex(
    PDFDocument hDocument,
    DWORD index,
    HPDFOBJECT* phObject
);
```

Parameters

hDocument
: [in] *hDocument* specifies the PDF object previously created by the function **PXCp_Init**.

index
: [in] *index* specifies the index of the object.

phObject
: [out] *phObject* specifies a pointer to a variable of the `HPDFOBJECT` type.
`HPDFOBJECT` is defined as:
`typedef void* HPDFOBJECT;`

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieves the object specified by it's index
```

```
    PDFDocument hDoc

    ...

    HPDFOBJECT        hObject;
    DWORD                ObjectIndex = 1;

    hr = PXCp_llGetObjectByIndex(hDoc, ObjectIndex, &hObject);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.1.3 PXCp_llGetObjectsCount

PXCp_llGetObjectsCount

Top Previous Next

PXCp_llGetObjectsCount retrieves the total object count within a PDF document. The number of objects may vary as new objects may be added to a document.

```
HRESULT  PXCp_llGetObjectsCount(
    PDFDocument hDocument,
    DWORD* pCnt
);
```

Parameters

hDocument

[in] *hDocument* specifies the PDF object previously created by the function **PXCp_Init**.

pCnt

[out] *pCnt* specifies a pointer to a variable of the `DWORD` type which receives the number of objects.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieves the number of objects in the PDF document

    DWORD GetNumberOfObjects(PDFDocument hDoc)
    {
        HRESULT hr = DS_OK;
        DWORD ObjsCount = 0;
        // Retrieve objects count

        hr = PXCp_llGetObjectsCount(hDoc, &ObjsCount);
        if (IS_DS_FAILED(hr))
        {
            // report error
            ...
        }

        return ObjsCount;
    }
```

3.2.1.4 PXCp_llGetPageByIndex

PXCp_llGetPageByIndex

Top Previous Next

PXCp_llGetPageByIndex retrieves a PDF object that represents a page in a PDF document.
For additional detailed information regarding the PDF format file structure please refer to Adobe's PDF format Reference documentation.

```
HRESULT  PXCp_llGetPageByIndex(
    PDFDocument hDocument,
    DWORD index,
    HPDFOBJECT* phObject
);
```

Parameters

hDocument
[in] *hDocument* specifies the PDF object previously created by the function **PXCp_Init**.

index
[in] *index* specifies a page number.

phObject
[out] *phObject* specifies a pointer to a variable of the `HPDFOBJECT` type that receives the page object.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieves the object which corresponds to the specified page

    PDFDocument hDoc

    ...
    HPDFOBJECT        hPage;
    DWORD                PageIndex = 0;

    hr = PXCp_llGetPageByIndex(hDoc, PageIndex, &hPage);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.1.5 PXCp_llGetPageCount

PXCp_llGetPageCount

Top Previous Next

PXCp_llGetPageCount retrieves the number of pages in a PDF document.

```
HRESULT  PXCp_llGetPageCount(
    PDFDocument hDocument,
    DWORD* pcnt
);
```

Parameters

hDocument
[in] *hDocument* specifies the PDF object previously created by the function **PXCp_Init**.

pcnt
[out] *pcnt* specifies a pointer to a variable which receives number of pages.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieves the number of pages in the document

    PDFDocument hDoc

    ...

    DWORD PageCount;

    hr = PXCp_llGetPageCount(hDoc, &PageCount);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.1.6 PXCp_llGetRootObject

PXCp_llGetRootObject

Top Previous Next

PXCp_llGetRootObject retrieves the PDF document Root (with reference to the PDF 'Catalog') object.

For additional detailed information regarding the PDF document Root and the PDF format file structure please refer to Adobe's PDF format Reference documentation..

```
HRESULT  PXCp_llGetRootObject(
    PDFDocument hDocument,
    HPDFOBJECT* phRoot
);
```

Parameters

hDocument
[in] *hDocument* specifies the PDF object previously created by the function **PXCp_Init**.

phRoot
[out] specifies a pointer to a variable of the `HPDFOBJECT` type that receives the root object.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
       // Retrieves the object which corresponds to the Root object, as
described
    // in the PDF Reference as a 'Catalog' object

    PDFDocument hDoc

    ...

    HPDFOBJECT hRoot;

    hr = PXCp_llGetRootObject(hDoc, &hRoot);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.2 Variant Functions

3.2.2.1 PXCp_VariantCopyFrom

PXCp_VariantCopyFrom

Top Previous Next

PXCp_VariantCopyFrom copies the content (data) of one PDF variant to another.

All original data in the target variant is replaced by that of the source variant.

```
HRESULT  PXCp_VariantCopyFrom(
    HPDFVARIANT hVariantDest,
    const HPDFVARIANT hVariantSrc
);
```

Parameters

hVariantDest
[in] specifies a `destination` variant handle.

hVariantSrc
[in] specifies a `source` variant handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Copy one variant value to another
    HPDFVARIANT         hVariantTarget;
    HPDFVARIANT         hVariantSource;

    HRESULT hr = PXCp_VariantCopyFrom(hVariantTarget, hVariantSource);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now hVariantTarget is the handle of the PDF variant that is
    // a copy of the hVariantSource
```

3.2.2.2 PXCp_VariantCreate

PXCp_VariantCreate

Top Previous Next

PXCp_VariantCreate creates a PDF variant object. This object is used to represent lexems in a PDF document.

For additional detailed information regarding the PDF format and its structure, please refer directly to the Adobe PDF Reference documentation.

```
HPDFVARIANT  PXCp_VariantCreate(

);
```

Parameters

None.

Return Values

If the function succeeds, the return value is not `NULL` and represents a valid PDF variant handle. If the function fails, the return value is `NULL`.

Example (C++).

```
// Create a PDF variant

    HPDFVARIANT hVariant = PXCp_VariantCreate();
    if (hVariant == NULL)
    {
        // Handle error
        ...
    }

    // Now hVariant is the valid handle of the PDF variant
    // and can be used in the further varianr functions
```

3.2.2.3 PXCp_VariantDelete

PXCp_VariantDelete

Top Previous Next

PXCp_VariantDelete deletes a PDF variant referenced by its handle when the variant is no longer required.

```
HRESULT  PXCp_VariantDelete(
    HPDFVARIANT hVariant
);
```

Parameters

hVariant
[in] specifies PDF variant handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Delete the PDF variant

    HPDFVARIANT hVariant;

    HRESULT hr = PXCp_VariantDelete(hVariant);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now hVariant is NOT the valid handle of the PDF variant
    // and can NOT be used in further variant functions
```

3.2.2.4 PXCp_VariantGetArray

PXCp_VariantGetArray

Top Previous Next

PXCp_VariantGetArray retrieves a PDF array value stored within a specified PDF variant.

```
HRESULT  PXCp_VariantGetArray(
    HPDFVARIANT hVariant,
    HPDFARRAY* phArray
);
```

Parameters

hVariant
[in] specifies variant handle.

phArray
[out] specifies a pointer to a variable of the `HPDFARRAY` type which recieves a PDF array value from a PDF variant.

Return Values

If the function succeeds, the return value is non-negative integer.
If a variant has an improper type then the function will return **`DPro_Err_InvalidVarType`**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve array value from the PDF variant
```

```
    HPDFVARIANT        hVariant;

    ...

    HPDFARRAY hArray;
    HRESULT hr = PXCp_VariantGetArray(hVariant, &hArray);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.2.5 PXCp_VariantGetBOOL

PXCp_VariantGetBOOL

Top Previous Next

PXCp_VariantGetBOOL retrieves a boolean value stored in a specified PDF variant.

```
HRESULT  PXCp_VariantGetBOOL(
    HPDFVARIANT hVariant,
    BOOL* pbVal
);
```

Parameters

hVariant

[in] specifies variant handle.

pbVal

[out] specifies a pointer to a variable of the `BOOL` type which receives the boolean value from the PDF variant.

Return Values

If the function succeeds, the return value is non-negative integer.
If a variant has an invalid type then the function returns **`DPro_Err_InvalidVarType`**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve boolean value from the PDF variant

    HPDFVARIANT        hVariant;

    ...

    BOOL bVal;
    HRESULT hr = PXCp_VariantGetBOOL(hVariant, &bVal);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.2.6 PXCp_VariantGetDictionary

PXCp_VariantGetDictionary

Top Previous Next

PXCp_VariantGetDictionary retrieves a PDF dictionary value stored in a specified PDF variant.

```
HRESULT  PXCp_VariantGetDictionary(
    HPDFVARIANT hVariant,
    HPDFDICTIONARY* phDictionary
);
```

Parameters

hVariant
[in] specifies a variant handle.

phDictionary
[out] specifies a pointer to a variable of the HPDFDICTIONARY type which receives the PDF dictionary value from the PDF variant.

Return Values

If the function succeeds, the return value is non-negative integer.
If a variant has an invalid type then the function returns **DPro_Err_InvalidVarType**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve dictionary value from the PDF variant

    HPDFVARIANT        hVariant;

    ...

    HPDFDICTIONARY hDictionary;
    HRESULT hr = PXCp_VariantGetDictionary(hVariant, &hDictionary);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.2.7 PXCp_VariantGetDouble

PXCp_VariantGetDouble

PXCp_VariantGetDouble retrieves the 'Real' number double value stored within a specified PDF variant.

```
HRESULT  PXCp_VariantGetDouble(
    HPDFVARIANT hVariant,
    double* pdVal
);
```

Parameters

hVariant
[in] specifies variant handle.

pdVal
[out] specifies the pointer to a variable of the `double` type which receives the value from a PDF variant.

Return Values

If the function succeeds, the return value is non-negative integer.
If a variant has an invalid type then the function returns **`DPro_Err_InvalidVarType`**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve a double value from the PDF variant

    HPDFVARIANT        hVariant;

    ...

    double dVal;
    HRESULT hr = PXCp_VariantGetDouble(hVariant, &dVal);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.2.8 PXCp_VariantGetInt

PXCp_VariantGetInt

Top Previous Next

PXCp_VariantGetInt retrieves the integer value stored in a specified PDF variant.

```
HRESULT  PXCp_VariantGetInt(
    HPDFVARIANT hVariant,
    __int64* piVal
);
```

Parameters

hVariant

[in] specifies variant handle.

piVal

[out] specifies a pointer to a variable of the `__int64` type which receives an integer value from a PDF variant.

Return Values

If the function succeeds, the return value is non-negative integer.
If a variant has an invalid type then the function returns **DPro_Err_InvalidVarType**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve integer value from the PDF variant

    HPDFVARIANT        hVariant;

    ...

    __int64 iVal;
    HRESULT hr = PXCp_VariantGetInt(hVariant, &iVal);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.2.9 PXCp_VariantGetName

PXCp_VariantGetName

Top Previous Next

PXCp_VariantGetName retrieves a PDF name value stored in a specified PDF variant.

```
HRESULT  PXCp_VariantGetName(
    HPDFVARIANT hVariant,
    HPDFSTRING* phString
);
```

Parameters

hVariant
[in] specifies a variant handle.

phString
[out] specifies a pointer to a variable of the `HPDFSTRING` type which receives the PDF name value from the PDF variant.

Return Values

If the function succeeds, the return value is non-negative integer.
If a variant has an invalid type then the function returns DPro_Err_InvalidVarType.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve name value from the PDF variant

    HPDFVARIANT        hVariant;

    ...

    HPDFSTRING hString;
    HRESULT hr = PXCp_VariantGetName(hVariant, &hString);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.2.10 PXCp_VariantGetObject

PXCp_VariantGetObject

Top Previous Next

PXCp_VariantGetObject retrieves a PDF object value stored in a specified PDF variant.

```
HRESULT  PXCp_VariantGetObject(
    HPDFVARIANT hVariant,
    HPDFOBJECT* phObject
);
```

Parameters

hVariant
[in] specifies a variant handle.

phObject

[out] specifies a pointer to a variable of the `HPDFOBJECT` type which receives the PDF object value from a PDF variant.

Return Values

If the function succeeds, the return value is non-negative integer.
If a variant has an invalid type then the function returns **`DPro_Err_InvalidVarType`**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve an object value from the PDF variant

    HPDFVARIANT        hVariant;

    ...

    HPDFOBJECT hObject;
    HRESULT hr = PXCp_VariantGetObject(hVariant, &hObject);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.2.11 PXCp_VariantGetString

PXCp_VariantGetString

Top Previous Next

PXCp_VariantGetString retrieves a PDF string value stored in a specified PDF variant.

```
HRESULT  PXCp_VariantGetString(
    HPDFVARIANT hVariant,
    HPDFSTRING* phString
);
```

Parameters

hVariant

[in] specifies a variant handle.

phString

[out] specifies a pointer to a variable of the `HPDFSTRING` type which receives the PDF string value from the PDF variant.

Return Values

If the function succeeds, the return value is non-negative integer.

If a variant has an invalid type then the function returns **`DPro_Err_InvalidVarType`**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve string value from the PDF variant

    HPDFVARIANT        hVariant;

    ...

    HPDFSTRING hString;
    HRESULT hr = PXCp_VariantGetString(hVariant, &hString);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.2.12 PXCp_VariantGetType

PXCp_VariantGetType

Top Previous Next

PXCp_VariantGetType retrieves the PDF variant type.

```
HRESULT  PXCp_VariantGetType(
    HPDFVARIANT hVariant,
    DPDFVariant_Type* pvarType
);
```

Parameters

hVariant
[in] specifies a variant handle.

pvarType
[out] specifies a pointer to a variable of the `DPDFVariant_Type` type which receives the variant type.
Possible values are:

Constant	Hex value	Meaning
PVT_EMPTY	0x0000	Empty variant, that has no value assigned to it.
PVT_NULL	0x0001	`null` value.
PVT_BOOL	0x0002	Boolean value.
PVT_INT	0x0003	Integer value. Correspoding to `__in64` type.
PVT_DOUBLE	0x0004	Double value.
PVT_NAME	0x0005	PDF `name` value.

PVT_STRING	0x0006	PDF `string` value.
PVT_ARRAY	0x0007	PDF `array` value.
PVT_DICTIONARY	0x0008	PDF `dictionary` value.
PVT_OBJREF	0x0009	Reference to the object.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

Note: For details about possible token types in a PDF document - refer to the Adobe PDF documentation.

Example (C++).

```
// Get PDF variant type

    HPDFVARIANT hVariant;

    ...

    DPDFVariant_Type        varType;

    HRESULT hr = PXCp_VariantGetType(hVariant, &varType);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now varType contains the information about variant type
```

3.2.2.13 PXCp_VariantGetType

PXCp_VariantGetType

Top Previous Next

PXCp_VariantSetArray sets a PDF array value to a PDF variant. All previous variant data will be deleted.

```
HRESULT  PXCp_VariantSetArray(
    HPDFVARIANT hVariant,
    HPDFARRAY hArray
);
```

Parameters

hVariant

[in] specifies variant handle.

hArray
[in] handle of a PDF array value to be set to the variant.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set array value to the PDF variant

    HPDFVARIANT         hVariant;
    HPDFARRAY                 hArray;

    ...

    HRESULT hr = PXCp_VariantSetArray(hVariant, hArray);
    if (IS_DS_FAILED(hr))
    {
        // Handle any error
        ...
    }
    ...
```

3.2.2.14 PXCp_VariantSetBOOL

PXCp_VariantSetBOOL

Top Previous Next

PXCp_VariantSetBOOL sets a boolean value to a PDF variant, all existing variant data is deleted.

```
HRESULT  PXCp_VariantSetBOOL(
    HPDFVARIANT hVariant,
    BOOL bVal
);
```

Parameters

hVariant
[in] specifies a variant handle.

bVal
[in] boolean value to be set to a variant.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes

page.

Example (C++).

```
// Set Boolean value to the PDF variant

    HPDFVARIANT         hVariant;
    BOOL                 bVal;

    ...

    HRESULT hr = PXCp_VariantSetBOOL(hVariant, bVal);
    if (IS_DS_FAILED(hr))
    {
        // Handle any error
        ...
    }
    ...
```

3.2.2.15 PXCp_VariantSetDictionary

PXCp_VariantSetDictionary

Top Previous Next

PXCp_VariantSetDictionary sets a PDF dictionary value to a PDF variant. All previous variant data will be deleted.

```
HRESULT  PXCp_VariantSetDictionary(
    HPDFVARIANT hVariant,
    HPDFDICTIONARY hDictionary
);
```

Parameters

hVariant
[in] specifies a variant handle.

hDictionary
[in] handle of the PDF dictionary value to be set to a variant.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set dictionary value to the PDF variant

    HPDFVARIANT         hVariant;
    HPDFDICTIONARY          hDictionary;
```

```
...

HRESULT hr = PXCp_VariantSetDictionary(hVariant, hDictionary);
if (IS_DS_FAILED(hr))
{
    // Handle any error
    ...
}
...
```

3.2.2.16 PXCp_VariantSetDouble

PXCp_VariantSetDouble

Top Previous Next

PXCp_VariantSetDouble sets to double a PDF variant representing a 'Real' mathmatical number value. All previous variant data will be deleted.

```
HRESULT  PXCp_VariantSetDouble(
    HPDFVARIANT hVariant,
    double dVal
);
```

Parameters

hVariant
: [in] specifies a variant handle.

dVal
: [in] double value to be set to variant.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set double value to the PDF variant

    HPDFVARIANT        hVariant;
    double                dVal;

    ...

    HRESULT hr = PXCp_VariantSetDouble(hVariant, dVal);
    if (IS_DS_FAILED(hr))
    {
        // Handle any error
        ...
    }
    ...
```

3.2.2.17 PXCp_VariantSetInt

PXCp_VariantSetInt

Top Previous Next

PXCp_VariantSetInt sets an integer value to a PDF variant, all existing variant data will be deleted.

```
HRESULT  PXCp_VariantSetInt(
    HPDFVARIANT hVariant,
    __int64 iVal
);
```

Parameters

hVariant
[in] specifies a variant handle.

iVal
[in] integer value to be set to the variant.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set integer value to the PDF variant

    HPDFVARIANT        hVariant;
    __int64                iVal;

    ...

    HRESULT hr = PXCp_VariantSetInt(hVariant, iVal);
    if (IS_DS_FAILED(hr))
    {
        // Handle any error
        ...
    }
    ...
```

3.2.2.18 PXCp_VariantSetName

PXCp_VariantSetName

Top Previous Next

PXCp_VariantSetName sets the PDF name value to a PDF variant. All previous variant data will be deleted.

```
HRESULT  PXCp_VariantSetName(
    HPDFVARIANT hVariant,
    HPDFSTRING hString
);
```

Parameters

hVariant
[in] specifies a variant handle.

hString
[in] handle of the PDF string value to be set to the variant.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set name to the PDF variant

    HPDFVARIANT        hVariant;

    ...

    // Create and set name

    HPDFSTRING hStr = PXCp_StringCreate();
    PXCp_StringSetA(hStr, "This is my name");

    HRESULT hr = PXCp_VariantSetName(hVariant, hStr);
    if (IS_DS_FAILED(hr))
    {
        // Handle any error
        ...
    }
    ...
```

3.2.2.19 PXCp_VariantSetNull

PXCp_VariantSetNull

Top Previous Next

PXCp_VariantSetNull sets to `null` all values within a variant, all variant data is deleted.

```
HRESULT  PXCp_VariantSetNull(
    HPDFVARIANT hVariant
);
```

Parameters

hVariant
[in] specifies a variant handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set 'null' value to the PDF variant

    HPDFVARIANT        hVariant;

    ...

    HRESULT hr = PXCp_VariantSetNull(hVariant);
    if (IS_DS_FAILED(hr))
    {
        // Handle any error
        ...
    }
    ...
```

3.2.2.20 PXCp_VariantSetObject

PXCp_VariantSetObject

Top Previous Next

PXCp_VariantSetObject sets a PDF object value to a PDF variant. All previous variant data will be deleted.

```
HRESULT  PXCp_VariantSetObject(
    HPDFVARIANT hVariant,
    HPDFOBJECT hObject
);
```

Parameters

hVariant

[in] specifies variant handle.

hObject

[in] handle of the PDF object value to be set to the variant.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set object value to the PDF variant

    HPDFVARIANT          hVariant;
    HPDFOBJECT          hObject;

    ...

    HRESULT hr = PXCp_VariantSetObject(hVariant, hObject);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.2.21 PXCp_VariantSetString

PXCp_VariantSetString

Top Previous Next

PXCp_VariantSetString sets a PDF string value to a PDF variant. All previous variant data will be deleted.

```
HRESULT  PXCp_VariantSetString(
    HPDFVARIANT hVariant,
    HPDFSTRING hString
);
```

Parameters

hVariant

[in] specifies a variant handle.

hString

[in] handle of the PDF string value to be set to the variant.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes

page.

Example (C++).

```
// Set string to the PDF variant

    HPDFVARIANT        hVariant;

    ...

    // Create and set string

    HPDFSTRING hStr = PXCp_StringCreate();
    PXCp_StringSetA(hStr, "This is my string");

    HRESULT hr = PXCp_VariantSetString(hVariant, hStr);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.3 Object Manipulation Functions

3.2.3.1 PXCp_ObjectCreateNew

PXCp_ObjectCreateNew

Top Previous Next

PXCp_ObjectCreateNew creates a new PDF object and adds it to the specified PDF document. Initially the object is created exclusive of a dictionary, body or stream.

```
HRESULT  PXCp_ObjectCreateNew(
     PDFDocument hDocument,
     HPDFOBJECT* phObject
);
```

Parameters

hDocument

[in] specifies the PDF object previously created by the function **PXCp_Init**.

phObject

[out] specifies a pointer to the `HPDFOBJECT` which receives the handle of the newly created object.
`typedef void* HPDFOBJECT;`

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Create new object

    PDFDocument hDoc

    ...

    HPDFOBJECT hObject;

    hr = PXCp_ObjectCreateNew(hDoc, &hObject);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.3.2 PXCp_ObjectGetBody

PXCp_ObjectGetBody

Top Previous Next

PXCp_ObjectGetBody retrieves a PDF objects's body.

If an object has no dictionary then its token(s) are packed into an PDF variant called the `'body'`.

The `'body'` handle received by this function should never be deleted!

```
HRESULT  PXCp_ObjectGetBody(
    HPDFOBJECT hObject,
    HPDFVARIANT* phVariant
);
```

Parameters

hObject
[in] specifies an object handle.

phVariant
[out] specifies a pointer to an `HPDFVARIANT` which receives the object's body.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve object's body
```

```
    HPDFOBJECT hObject;

    ...

    HPDFVARIANT hBody;

    hr = PXCp_ObjectGetBody(hObject, &hBody);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.3.3 PXCp_ObjectGetDictionary

PXCp_ObjectGetDictionary

Top Previous Next

PXCp_ObjectGetDictionary retrieves an object's dictionary.

```
HRESULT  PXCp_ObjectGetDictionary(
    HPDFOBJECT hObject,
    HPDFDICTIONARY* phDict
);
```

Parameters

hObject
[in] specifies an object handle.

phDict
[out] specifies a pointer to the `HPDFDICTIONARY` which receives the object's dictionary.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve object's dictionary

    HPDFOBJECT hObject;

    ...

    HPDFDICTIONARY hDict;

    hr = PXCp_ObjectGetDictionary(hObject, &hDict);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.3.4 PXCp_ObjectGetStream

PXCp_ObjectGetStream

Top Previous Next

PXCp_ObjectGetStream retrieves an object's stream (if present). If an object contains a stream then it must also contain a dictionary.

A Stream may not be simply `'set'` to an object as there may not be a stream without a corresponding object. should it be necessary to set a Stream to an object - the **PXCp_StreamCreate** function should be used.

```
HRESULT  PXCp_ObjectGetStream(
    HPDFOBJECT hObject,
    HPDFSTREAM* phStream
);
```

Parameters

hObject
[in] specifies an object handle.

phStream
[out] specifies a pointer to n `HPDFSTREAM` which receives the object's dictionary.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Retrieve an object's stream

    HPDFOBJECT hObject;

    ...

    HPDFSTREAM hStream;

    hr = PXCp_ObjectGetStream(hObject, &hStream);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.3.5 PXCp_ObjectSetBody

PXCp_ObjectSetBody

Top Previous Next

PXCp_ObjectSetBody applies a new body to an object. If the object already contains a body, then once the operation is complete, the original body is deleted automatically and replaced by that passed by **PXCp_ObjectSetBody**, extreme caution is required when managing body variant handles as the new body applied to an object must not be deleted - the body variant handle remains available to the library for further valid use and great care is required so as not to delete a body in error.

```
HRESULT  PXCp_ObjectSetBody(
    HPDFOBJECT hObject,
    HPDFVARIANT hVariant
);
```

Parameters

hObject
[in] specifies object handle.

hVariant
[in] specifies body variant handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set body to object

    HPDFOBJECT hObject;

    ...

    HPDFVARIANT hVariant;

    hr = PXCp_ObjectSetBody(hObject, hVariant);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.3.6 PXCp_ObjectSetDictionary

PXCp_ObjectSetDictionary

Top Previous Next

PXCp_ObjectSetDictionary applies a new dictionary to an object. If the object already contains a dictionary, then once the operation is complete, the original dictionary is deleted automatically and replaced by that passed by **PXCp_ObjectSetDictionary.**

Extreme caution is required when managing dictionary handles as the new dictionary applied to an object must not be deleted - the dictionary handle remains available to the library for valid use (i.e. the addition or modification of keys etc) and great care is required so as not to delete a dictionary in error.

```
HRESULT  PXCp_ObjectSetDictionary(
    HPDFOBJECT hObject,
    HPDFDICTIONARY hDict
);
```

Parameters

hObject
[in] specifies an object handle.

hDict
specifies a dictionary handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set dictionary to object

    HPDFOBJECT hObject;

    ...

    HPDFDICTIONARY hDict;

    hr = PXCp_ObjectSetDictionary(hObject, hDict);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.4 PDF Array Functions

3.2.4.1 PXCp_ArrayAppendFrom

PXCp_ArrayAppendFrom

Top Previous Next

PXCp_ArrayAppendFrom inserts items from an array specified by *hSrcArray* into a target array specified by *hDestArray* at the specified index.

```
HRESULT  PXCp_ArrayAppendFrom(
    HPDFARRAY hDestArray,
    const HPDFARRAY hSrcArray,
    LONG InsertBefore
);
```

Parameters

hDestArray
[in] *hDestArray* specifies the destination array handle.

hSrcArray
[in] *hSrcArray* specifies the source array handle.

InsertBefore
[in] *InsertBefore* specifies the index to which the new items should be inserted. If the index value is -1 then items are appended.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows, how to 'merge' two arrays

    HPDFARRAY                 hDestination;
    HPDFARRAY                 hSource;

    ...

    hr = PXCp_ArrayAppendFrom(hDestination, hSource, -1);

    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }

    // Now hDestination is the handle of the array which contains 'merged'
array

    ...
```

3.2.4.2 PXCp_ArrayClearAll

PXCp_ArrayClearAll

Top Previous Next

PXCp_ArrayClearAll removes all items from a specified array.

```
HRESULT  PXCp_ArrayClearAll(
    HPDFARRAY hArray
);
```

Parameters

hArray

[in] *hArray* specifies the array handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Removes all items from the array

    HPDFARRAY hArray;

    ...

    HRESULT hr = PXCp_ArrayClearAll(hArray);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
    // Now hArray contains no one item
```

3.2.4.3 PXCp_ArrayCreate

PXCp_ArrayCreate

Top Previous Next

PXCp_ArrayCreate creates a new array.

```
HPDFARRAY  PXCp_ArrayCreate(

);
```

Parameters

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows how to create a new array

    HPDFARRAY hArray = PXCp_ArrayCreate();

    ...

    // Now hArray is the valid array handle
    // that can be used for further array operations
```

3.2.4.4 PXCp_ArrayDelete

PXCp_ArrayDelete

Top Previous Next

PXCp_ArrayDelete destroys a specified array.

```
HRESULT  PXCp_ArrayDelete(
    HPDFARRAY hArray
);
```

Parameters

hArray
[in] *hArray* specifies the array handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Delete the array

    HPDFARRAY hArray;

    ...

    HRESULT hr = PXCp_ArrayDelete(hArray);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
    // Now hArray is not the valid array handle
    // and can not be used in any array operations
```

3.2.4.5 PXCp_ArrayDeleteAt

PXCp_ArrayDeleteAt

Top Previous Next

PXCp_ArrayDeleteAt removes an array item as specified by the index value, from the specified array.

```
HRESULT  PXCp_ArrayDeleteAt(
    HPDFARRAY hArray,
    DWORD index,
    BOOL bNonCopy
);
```

Parameters

hArray

[in] *hArray* specifies the array handle.

index

[in] *index* specifies the index of the array item to be removed.

bNonCopy

[in] *bNonCopy* specifies the required behaviour for the item being removed. If the parameter is TRUE then the item is removed and destroyed, if false the item is removed only from the array and remains available to copy or re-use as required, by reference to the array handle (HPDFVARIANT). It should also be noted that should the array subsequently be released, the item no longer remains available or valid

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Remove the item from the array
    // item is specified by it's index

    HPDFARRAY                hArray;
    DWORD                ItemIndex;

    ...

    HRESULT hr = PXCp_ArrayDeleteAt(hArray, ItemIndex, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    ...
```

3.2.4.6 PXCp_ArrayGetByIndex

PXCp_ArrayGetByIndex

Top Previous Next

PXCp_ArrayGetByIndex retrieves the value of an array item as specified by the index value.

```
HRESULT  PXCp_ArrayGetByIndex(
    HPDFARRAY hArray,
    DWORD index,
    HPDFVARIANT* phVariant
);
```

Parameters

hArray
[in] *hArray* specifies the array handle.

index
[in] *index* specifies an item's index.

phVariant
[out] *phVariant* is a pointer to a variable that receives the array item handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows how to go through all array items
    // and retrieve their values

    void GoThroughAllItems(HPDFARRAY hArray)
    {
        HRESULT         hr = DS_OK;
        DWORD          ItemCount = 0;

        // Get number of items in the array

        hr = PXCp_ArrayGetCount(hArray, &ItemCount);
        if (IS_DS_FAILED(hr))
        {
            // Handle error
            ...
        }

        // Check if array has any item

        if (ItemCount == 0)
            return;

        // Now get all items from the array

        for (DWORD ic = 0; ic < ItemCount; ic++)
        {
            HPDFVARIANT                hVariant;
            hr = PXCp_ArrayGetByIndex(hArray, ic, &hVariant);
            if (IS_DS_FAILED(hr))
            {
                // Handle error
                ...
            }

            // Now hVariant contains the value of the item

            ...
        }

    }
```

3.2.4.7 PXCp_ArrayGetCount

PXCp_ArrayGetCount

Top Previous Next

PXCp_ArrayGetCount retrieves a count for the items in a specified array.

```
HRESULT  PXCp_ArrayGetCount(
    HPDFARRAY hArray,
    DWORD* pCount
);
```

Parameters

hArray

[in] *hArray* specifies the array handle.

pCount

[out] *pCount* is a pointer to a variable that receives the number of items in a specified array.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Get number of items in the array

    HPDFARRAY hArray;

    ...

    DWORD ItemsCount = 0;

    HRESULT hr = PXCp_ArrayGetCount(hArray, &ItemsCount);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.4.8 PXCp_ArrayInsertAt

PXCp_ArrayInsertAt

Top Previous Next

PXCp_ArrayInsertAt inserts a new array item into a specified array for a specified index. Once complete, the new array item is accessible by the specified index.

```
HRESULT  PXCp_ArrayInsertAt(
    HPDFARRAY hArray,
    LONG index,
    HPDFVARIANT hVariant,
    BOOL bNonCopy
);
```

Parameters

hArray
[in] *hArray* specifies the array handle.

index
[in] *index* specifies an index value for the inserted item. If the index value is -1 then the new item is appended to the array.

hVariant
[in] *hVariant* specifies a value for the new item.

bNonCopy
[in] *bNonCopy* specifies the current and future behaviour for the item value. If this parameter is TRUE

then the value is destroyed on completion of the action and the item handle is invalid for future use. Otherwise, the handle remains available and the item and values may be re-used (i.e. for further array item insertion and duplication) until such time as the array itself is released and no longer available.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Insert new item into the begining of the array

   HPDFARRAY        hArray;

   ...

   // Create new item

   HPDFVARIANT        hVariant = PXCp_VariantCreate();
   PXCp_VariantSetInt(hVariant, 5);

   // Insert new item in the begining of hte array

   HRESULT hr = PXCp_ArrayInsertAt(hArray, 0, hVariant, TRUE);
   if (IS_DS_FAILED(hr))
   {
       // Handle error
       ...
   }

   // Note that hVariant handle should not be destroyed after the operation

   ...
```

3.2.4.9 PXCp_ArraySetValue

PXCp_ArraySetValue

Top Previous Next

PXCp_ArraySetValue sets a new value for array item accessed by a specified index.

```
HRESULT  PXCp_ArraySetValue(
    HPDFARRAY hArray,
    DWORD index,
    HPDFVARIANT hVariant,
    BOOL bNonCopy
);
```

Parameters

hArray
    [in] *hArray* specifies the array handle..

index
    [in] *index* specifies the item index.

hVariant
    [in] *hVariant* specifies the item value.

bNonCopy
    [in] *bNonCopy* specifies the current and future behaviour for the item value. If this parameter is TRUE then the value is destroyed on completion of the action and the item handle is invalid for future use. Otherwise, the handle remains available and the item and values may be re-used (i.e. for further array item value changes) until such time as the array itself is released and no longer available.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows how to change all item values in the array
    // for the integer type, i.e. [1, 2, 3, 4, ...]

    void ChangeToInteger(HPDFARRAY hArray)
    {
        HRESULT hr = DS_OK;

        DWORD        ItemCount = 0;

        // Get number of items in the array

        hr = PXCp_ArrayGetCount(hArray, &ItemCount);
        if (IS_DS_FAILED(hr))
        {
            // Handle error
            ...
        }

        // Check if array has any item

        if (ItemCount == 0)
            return;

        // Prepare PDF variant of the integer type

        HPDFVARIANT        hVariant = PXCp_VariantCreate();

        // Now get all items from the array and change them
        for (DWORD ic = 0; ic < ItemCount; ic++)
        {
            // Increase variant value

            PXCp_VariantSetInt(hVariant, ic + 1);

            hr = PXCp_ArraySetValue(hArray, ic, hVariant, FALSE);
            if (IS_DS_FAILED(hr))
            {
                // Handle error
                ...
            }
        }
        // Now hVariant should be deleted
        PXCp_VariantDelete(hVariant);
    }
```

3.2.5 PDF Dictionary Functions

3.2.5.1 PXCp_DictionaryAppendFrom

PXCp_DictionaryAppendFrom

Top Previous Next

PXCp_DictionaryAppendFrom appends items from one source dictionary as specified by *hSrcDict* to another specified target dictionary *hDestDict*.

```
HRESULT  PXCp_DictionaryAppendFrom(
    HPDFDICTIONARY hDestDict,
    const HPDFDICTIONARY hSrcDict
);
```

Parameters

hDestDict

hDestDict specifies the destination dictionary handle.

hSrcDict

hSrcDict specifies the source dictionary handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows how to 'merge' two dictionaries

    HPDFDICTIONARY          hDictSource;
    HPDFDICTIONARY          hDictTarget;
    ...

    HRESULT hr = PXCp_DictionaryAppendFrom(hDictTarget, hDictSource);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now hDictTarget consist of both it's own keys and the hDictSource keys

    ...
```

3.2.5.2 PXCp_DictionaryClearAll

PXCp_DictionaryClearAll

Top Previous Next

PXCp_DictionaryClearAll removes all items from a specified dictionary.

```
HRESULT  PXCp_DictionaryClearAll(
    HPDFDICTIONARY hDict
);
```

Parameters

hDict

[in] *hDict* specifies the required dictionary handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Removes all keys from the specified dictionary

    HPDFDICTIONARY hDict;

    ...

    HRESULT hr = PXCp_DictionaryClearAll(hDict);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
    // Now hDict contains no key
```

3.2.5.3 PXCp_DictionaryClearValue

PXCp_DictionaryClearValue

Top Previous Next

PXCp_DictionaryClearValue clears a value accessed by a specified key name from within a dictionary.

```
HRESULT  PXCp_DictionaryClearValue(
    HPDFDICTIONARY hDict,
    HPDFSTRING hString,
    BOOL bNonCopy
);
```

Parameters

hDict
: [in] *hDict* specifies the dictionary handle.

hString
: [in] *hString* specifies the handle of a PDF string with key name.

bNonCopy
: [in] *bNonCopy* specifies the required behaviour of a defined value within a keyname. If the parameter is TRUE then the value contents are cleared and the value handle is deemed invalid for future use. Otherwise, the handle is made available.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Remove the key from the dictionary

    HPDFDICTIONARY hDict;

    ...

    // Create string for the key name

    HPDFSTRING hStrName = PXCp_StringCreate();
    hr = PXCp_StringSetA(hStrName, "Type");

    HRESULT hr = PXCp_DictionaryClearValue(hDict, hStrName, TRUE);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now hDict has no 'Type' key

    ...

    // Clean up

    PXCp_StringDelete(hStrName);
```

3.2.5.4 PXCp_DictionaryCreate

PXCp_DictionaryCreate

Top Previous Next

PXCp_DictionaryCreate creates a new dictionary.

```
HPDFDICTIONARY  PXCp_DictionaryCreate(

);
```

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows how to create a new dictionary

    HPDFDICTIONARY hDict = PXCp_DictionaryCreate();

    ...

    // Now hDict is the valid dictionary handle
    // that can be used for further dictionary operations
```

3.2.5.5 PXCp_DictionaryDelete

PXCp_DictionaryDelete

Top Previous Next

PXCp_DictionaryDelete destroys a specified dictionary.

```
HRESULT  PXCp_DictionaryDelete(
    HPDFDICTIONARY hDict
);
```

Parameters

hDict
[in] *hDict* specifies the dictionary handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Delete the dictionary

    HPDFDICTIONARY hDict;

    ...

    HRESULT hr = PXCp_DictionaryDelete(hDict);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
    // Now hDict is not the valid dictionary handle
    // and can not be used in any further dictionary operations
```

3.2.5.6 PXCp_DictionaryGetCount

PXCp_DictionaryGetCount

Top Previous Next

PXCp_DictionaryGetCount retrieves a count of the items contained in a specified dictionary.

```
HRESULT  PXCp_DictionaryGetCount(
    HPDFDICTIONARY hDict,
    DWORD* pCount
);
```

Parameters

hDict
[in] *hDict* specifies the dictionary handle.

pCount
[out] *pCount* a pointer to a variable that receives a count of the items in a specified dictionary.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.

To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Get number of keys in the dictionary

    HPDFDICTIONARY hDict;

    ...

    DWORD KeyCount = 0;

    HRESULT hr = PXCp_DictionaryGetCount(hDict, &KeyCount);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.5.7 PXCp_DictionaryGetKeyByPDFStr

PXCp_DictionaryGetKeyByPDFStr

Top Previous Next

PXCp_DictionaryGetKeyByPDFStr retrieves a key value accessed in a dictionary by a key name, specified as a PDF string.

```
HRESULT  PXCp_DictionaryGetKeyByPDFStr(
    HPDFDICTIONARY hDict,
    const HPDFSTRING hKeyName,
    HPDFVARIANT* phVariant
);
```

Parameters

hDict
[in] *hDict* specifies the dictionary handle.

hKeyName
[in] *hKeyName* the handle of a PDF string specifying a key name.

phVariant
[out] *phVariant* is a pointer to a variable that receives the key value.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows how to get the specified dictionary key
// Example is similar to the example for the PXCp_DictionaryGetKeyByStr
function

    HPDFDICTIONARY        hDict;
    HPDFVARIANT        hVariant;

    // Create string for the key name

    HPDFSTRING hStrName = PXCp_StringCreate();
    hr = PXCp_StringSetA(hStrName, "Type");

    // Get key value

    hr = PXCp_DictionaryGetKeyByPDFStr(hDict, hStrName, &hVariant);

    ...
```

3.2.5.8 PXCp_DictionaryGetKeyByStr

PXCp_DictionaryGetKeyByStr

Top Previous Next

PXCp_DictionaryGetKeyByStr retrieves a key value accessed in a specified dictionary by the key name, specified as binary data.

```
HRESULT  PXCp_DictionaryGetKeyByStr(
    HPDFDICTIONARY hDict,
    LPCVOID buf,
    DWORD bufLen,
    HPDFVARIANT* phVariant
);
```

Parameters

hDict
[in] *hDict* specifies the dictionary handle.

buf
[in] *buf* is a constant pointer to a buffer with binary data representing the key name.

bufLen
[in] *bufLen* specifies the size in bytes of a buffer pointed by *buf*.

phVariant
[out] *phVariant* is a pointer to variable that receives the key value.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.

To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows how to get specified dictionary key

    HPDFDICTIONARY          hDict;
    HPDFVARIANT         hVariant;

    // Get key value

    hr = PXCp_DictionaryGetKeyByStr(hDict, "Type", 4, &hVariant);

    ...
```

3.2.5.9 PXCp_DictionaryGetPair

PXCp_DictionaryGetPair

Top Previous Next

PXCp_DictionaryGetPair retrieves both the name and value of a key in a specified dictionary accessed by the index.

```
HRESULT  PXCp_DictionaryGetPair(
    HPDFDICTIONARY hDict,
    DWORD index,
    HPDFSTRING hKeyName,
    HPDFVARIANT* phVariant
);
```

Parameters

hDict
[in] *hDict* specifies the dictionary handle.

index
[in] *index* specifies the zero based index of a key in a dictionary.

hKeyName
[in] *hKeyName* specifies the handle of a PDF string. After the function is completed successfully, call **PXCp_StringGetB** function to retrieve a string value.

phVariant
[out] *phVariant* is a pointer to a variable that receives the key value.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example shows how to go through all dictionary keys
// and retrieve their names and values

    void LookthroughAllKeys(HPDFDICTIONARY hDict)
    {
        HRESULT hr = DS_OK;
        DWORD        KeyCount = 0;

        // Get number of keys in the dictionry

        hr = PXCp_DictionaryGetCount(hDict, &KeyCount);
        if (IS_DS_FAILED(hr))
        {
            // Handle error
            ...
        }

        // Check if dictionary has any key

        if (KeyCount == 0)
            return;

        // Now get all keys from the dictionary

        for (DWORD kc = 0; kc < KeyCount; kc++)
        {
            HPDFSTRING                hKeyName = PXCp_StringCreate();
            HPDFVARIANT                hVariant;
            hr = PXCp_DictionaryGetPair(hDict, kc, hKeyName, &hVariant);
            if (IS_DS_FAILED(hr))
            {
                // Handle error
                ...
            }

            // Now hKeyName string contains key name
            // and hVariant contains it's value

            ...

        }

    }
```

3.2.5.10 PXCp_DictionarySetKeyByPDFStr

PXCp_DictionarySetKeyByPDFStr

Top Previous Next

PXCp_DictionarySetKeyByPDFStr sets the key value accessed for a specified dictionary by key name as specified by PDF string.

```
HRESULT  PXCp_DictionarySetKeyByPDFStr(
    HPDFDICTIONARY hDict,
    HPDFSTRING hString,
    HPDFVARIANT hVariant,
    BOOL bNonCopy
);
```

Parameters

hDict
[in] *hDict* specifies the dictionary handle.

hString
[in] *hString* specifies a key name as the PDF string.

hVariant
[in] *hVariant* specifies a key value.

bNonCopy
[in] *bNonCopy* [in] this parameter controls the way a key value is set to a dictionary. See remarks below.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

The value (as represented by a `PDF variant` handle) may be set to the dictionary by 2 alternate means.

- As a **copy** of the existing variant. = `FALSE`.
  This allows the library **PXCp_DictionarySetKeyByPDFStr** to make a copy of the current variant and apply this copy to a dictionary as the specified key's value. Any further modification of the variant will not affect the value applied to the dictionary (as the original copy is already stored within the dictionary).
  bNonCopy May also be used for other purposes i.e. update the values and apply to another key etc. *bNonCopy* should be deleted by the **PXCp_VariantDelete** function once no longer required.
- As a value. = `TRUE`.
  During the process the **PXCp_DictionarySetKeyByPDFStr** should `NOT` make a copy of the variant *bNonCopy*, and the variant is stored in a similar manner to a 'pointer'. When the value of the *bNonCopy* is amended after processing then any key value in the dictionary will also be updated as well.
  In this case the `PDF variant`*bNonCopy* should not be deleted!

Example (C++).

```
// Example shows how to create a dictionary and add to it
// the key 'Type' with the value 'Catalog'
// Example similar to example for the PXCp_DictionarySetKeyByStr function

    // Create dictionary

    HPDFDICTIONARY hDictCatalog = PXCp_DictionaryCreate();

    // Create PDF variant with the name value 'Catalog'

    HPDFVARIANT hV = PXCp_VariantCreate();
    HPDFSTRING hStr = PXCp_StringCreate();
    hr = PXCp_StringSetA(hStr, "Catalog");
    hr = PXCp_VariantSetName(hV, hStr);

    // Create string for the key name

    HPDFSTRING hStrName = PXCp_StringCreate();
    hr = PXCp_StringSetA(hStrName, "Type");


    // Set new key

    hr = PXCp_DictionarySetKeyByPDFStr(hDictCatalog, hStrName, hV, FALSE);

    // As the key was passed as copy it has to be deleted if unnecessary

    hr = PXCp_VariantDelete(hV);

    ...
```

3.2.5.11 PXCp_DictionarySetKeyByStr

PXCp_DictionarySetKeyByStr

Top Previous Next

PXCp_DictionarySetKeyByStr sets a key value (passed as binary data) in a specified dictionary. If a key of the same name already exists, the key data is replaced, otherwise a new key is added to the dictionary .

```
HRESULT  PXCp_DictionarySetKeyByStr(
    HPDFDICTIONARY hDict,
    LPCVOID buf,
    DWORD bufLen,
    HPDFVARIANT hVariant,
    BOOL bNonCopy
);
```

Parameters

hDict
[in] *hDict* specifies the dictionary handle.

buf
[in] *buf* is a constant pointer to a buffer with binary data to represent the key name.

bufLen
[in] *bufLen* specifies size in bytes of a buffer pointed by *buf*.

hVariant
[in] *hVariant* specifies the key value.

bNonCopy
[in] this parameter controls the way a key value is set to a dictionary. See remarks below.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

The value (as represented by a `PDF variant` handle) may be set to the dictionary by 2 alternate means.

- As a **copy** of the existing variant. *bNonCopy* = `FALSE`.
  This allows the library **PXCp_DictionarySetKeyByStr** to make a copy of the current variant and apply this copy to a dictionary as the specified key's value. Any further modification of the variant will not affect the value applied to the dictionary (as the original copy is already stored within the dictionary).
  hVariant May also be used for other purposes i.e. update the values and apply to another key etc.
  hVariant should be deleted by the **PXCp_VariantDelete** function once no longer required.
- As a value. *bNonCopy* = `TRUE`.
  During the process the **PXCp_DictionarySetKeyByStr** should `NOT` make a copy of the variant *hVariant*, and the variant is stored in a similar manner to a 'pointer'. When the value of the *hVariant* is amended after processing then any key value in the dictionary will also be updated as well.
  In this case the `PDF variant`*hVariant* should not be deleted!

Example (C++).

```
// Example shows how to create a dictionary and add to it
// the key 'Type' with the value 'Catalog'

    // Create dictionary

    HPDFDICTIONARY hDictCatalog = PXCp_DictionaryCreate();

    // Create PDF variant with the name value 'Catalog'

    HPDFVARIANT hV = PXCp_VariantCreate();
    HPDFSTRING hStr = PXCp_StringCreate();
    hr = PXCp_StringSetA(hStr, "Catalog");
    hr = PXCp_VariantSetName(hV, hStr);

    // Set new key
    hr = PXCp_DictionarySetKeyByStr(hDictCatalog, "Type", 4, hV, FALSE);

    // As the key was passed as copy it has to be deleted if no longer
required.

    hr = PXCp_VariantDelete(hV);

    ...
```

3.2.6 PDF String Functions

3.2.6.1 PXCp_StringCreate

PXCp_StringCreate

Top Previous Next

PXCp_StringCreate creates a new string.

```
HPDFSTRING  PXCp_StringCreate(

);
```

Parameters

None.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Create the PDF string

    HPDFSTRING hString = PXCp_StringCreate();
    if (hString == NULL)
    {
        // Handle error
        ...
    }

    // Now hString is the valid handle of the PDF string
    // and can be used in further string functions
```

3.2.6.2 PXCp_StringDelete

PXCp_StringDelete

Top Previous Next

PXCp_StringDelete destroys a specified string.

```
HRESULT  PXCp_StringDelete(
    HPDFSTRING hString
);
```

Parameters

hString
[in] *hString* specifies the string handle.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Delete the PDF string

    HPDFSTRING hString;

    HRESULT hr = PXCp_StringDelete(hString);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }

    // Now hString is NOT a valid handle of the PDF string
    // and can NOT be used in further string functions
```

3.2.6.3 PXCp_StringGetB

PXCp_StringGetB

Top Previous Next

PXCp_StringGetB retrieves a specified PDF string in binary data format.

```
HRESULT  PXCp_StringGetB(
    HPDFSTRING hString,
    BYTE* buf,
    DWORD* pbufLen
);
```

Parameters

hString

[in] *hString* specifies the PDF string handle.

buf

[in] *buf* is a pointer to a buffer that receives string content presented as binary data. This parameter may be NULL.

pbufLen

[in/out] *pbufLen* is a pointer to variable that specifies the size of the buffer in bytes. If *buf* is NULL then *pbufLen* creates a buffer sufficient to receive the binary data.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Get string data

    HPDFSTRING hString;

    BYTE*         buffer = NULL;
    DWORD         bufLength = 0;

    // Retrieve necesary buffer length

    hr = PXCp_StringGetB(hString, NULL, &bufLength);
    if (IS_DS_FAILED(hr) || (bufLength == 0))
    {
        // Handle error
        ...
    }

    // Allocate buffer

    buffer = new BYTE[bufLength];

    // Retrive string data
    hr = PXCp_StringGetB(hString, buffer, &bufLength);

    // Now buffer contains string data

    ...

    // Do not forget to free allocated buffer when unnecessary

    ...
```

3.2.6.4 PXCp_StringSetA

PXCp_StringSetA

Top Previous Next

PXCp_StringSetA sets ASCII data for a specified string.

```
HRESULT  PXCp_StringSetA(
    HPDFSTRING hString,
    LPCSTR aStr
);
```

Parameters

hString

[in] *hString* specifies the handle of a string.

aStr

[in] *aStr* is a constant pointer to a buffer that contains an ASCII string. The string must be NULL terminated.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set the PDF string value using null-terminated ASCII string

   HPDFSTRING        hString;

   ...

   hr = PXCp_StringSetA(hString, "This is ASCII string");
   if (IS_DS_FAILED()hr)
   {
       // Handle error
       ...
   }
   ...
```

3.2.6.5 PXCp_StringSetB

PXCp_StringSetB

Top Previous Next

PXCp_StringSetB sets binary data for a specified string.

```
HRESULT  PXCp_StringSetB(
    HPDFSTRING hString,
    const BYTE* buf,
    DWORD bufLen
);
```

Parameters

hString

[in] *hString* specifies the string handle.

buf

[in] *buf* is a constant pointer to a buffer that contains binary data.

bufLen

[in] *bufLen* specifies the size in bytes of binary data as pointed to by *buf*.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set the PDF string value using any binary data

   HPDFSTRING        hString;

   ...
   LPCSTR         aStr = "This is an example string";
   hr = PXCp_StringSetB(hString, (const BYTE*)aStr, ::lstrlenA(aStr));
   if (IS_DS_FAILED()hr)
   {
       // Handle error
       ...
   }
   ...
```

3.2.6.6 PXCp_StringSetW

PXCp_StringSetW

Top Previous Next

PXCp_StringSetW sets Unicode data for a specified string.

```
HRESULT  PXCp_StringSetW(
    HPDFSTRING hString,
    LPCWSTR wStr
);
```

Parameters

hString

[in] *hString* specifies the string handle.

wStr

[in] *wStr* is a constant pointer to a buffer that contains a Unicode string. The string must be NULL terminated.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Set the PDF string value using a null-terminated UNICODE string

    HPDFSTRING          hString;

    ...

    hr = PXCp_StringSetW(hString, L"This is a UNICODE string");
    if (IS_DS_FAILED()hr)
    {
        // Handle error
        ...
    }
    ...
```

3.2.7 PDF Stream Functions

3.2.7.1 PXCp_StreamCreate

PXCp_StreamCreate Top Previous Next

PXCp_StreamCreate creates a stream in a specified PDF object. A Stream may not be created without belonging to a specific object.

```
HRESULT  PXCp_StreamCreate(
    HPDFSTREAM* phStream,
    HPDFOBJECT hObject
);
```

Parameters

phStream
[out] *phStream* specifies the pointer to a stream handle.
The Stream handle is defined as:
`typedef void* HPDFSTREAM;`

hObject
[in] *hObject* specifies the handle of the PDF object.

Return Values

If the function succeeds, the return value is a non-negative integer.

If the object specified by *hObject* already contains a stream then the function returns **DPro_Err_ObjHasStream**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Create the stream

    HPDFOBJECT hObject;         // The object which will own the stream

    ...

    HPDFSTREAM hStream;

    hr = PXCp_StreamCreate(&hStream, hObject);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
    // Now hStream is the valid handle of the stream that can be used in
    // further operations.
```

3.2.7.2 PXCp_StreamDecode

PXCp_StreamDecode

Top Previous Next

PXCp_StreamDecode decodes the specified number of filters that are applied to the stream data.

In a PDF document each stream may be compressed using one or more filters. The number of filters used for a particular stream may be obtained by using the **PXCp_StreamGetFiltersCount** function. The parameters used for the compression stream may further be obtained by using the **PXCp_StreamGetFilterParameters** function.

```
HRESULT  PXCp_StreamDecode(
    HPDFSTREAM hStream,
    UINT FilterCount
);
```

Parameters

hStream

[in] *hStream* specifies the stream handle.
The Stream handle is defined as:
`typedef void* HPDFSTREAM;`

FilterCount

[in] *FilterCount* specifies the number of filters to decode.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example to decode a complete stream

    UINT        FilterCnt = 0;
    HRESULT hr = PXCp_StreamGetFiltersCount(hStream, &FilterCnt);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        // ...
    }
    hr = PXCp_StreamDecode(hStream, FilterCnt);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        // ...
    }

    // At this pojnt the stream uncompressed
```

3.2.7.3 PXCp_StreamEncode

PXCp_StreamEncode

Top Previous Next

PXCp_StreamEncode encodes the stream data with the specified filter & parameters.

```
HRESULT  PXCp_StreamEncode(
    HPDFSTREAM hStream,
    FilterType FltType,
    LPPXCp_CompressParam pCompressParam
);
```

Parameters

hStream

[in] *hStream* specifies the stream handle.

FltType

[in] *FltType* specifies filter type. The filter type may be one of the following:

Value	Parameters for encoding?	Meaning
ft_ASCIIHex	no	ASCII hexadecimal representation. Increases the length of

		the data.
`ft_ASCII85`	no	ASCII base-85 representation. Increases the length of the data.
`ft_LZW`	no	LZW (Lempel-Ziv-Welch) adaptive compression method.
`ft_Flate`	yes	zlib/deflate compression method.
`ft_RLE`	no	Byte-oriented run-length encoding algorithm.
`ft_CCITTFax`	yes	CCITT facsimile standard.
`ft_DCT`	yes	DCT (discrete cosine transform) algorithm.
`ft_JPX`	yes	Wavelet-based JPEG2000 standard.

pCompressParam

[in] *pCompressParam* specifies a pointer to the **PXCp_CompressParam** structure with the parameters for the required *FltType* filter.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Encode stream with the ZIP codec

    HPDFSTREAM hStream;

    // Parameters for the encoder

    PXCp_CompressParam cparam;
    // Compression level
    cparam.CompressionLevel = 9; // Z_BEST_COMPRESSION

    // Encoder stream

    hr = PXCp_StreamEncode(hStream, ft_Flate, &cparam);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
    // Now stream is 'ziped'
```

3.2.7.4 PXCp_StreamGetFilterParameters

PXCp_StreamGetFilterParameters

Top Previous Next

PXCp_StreamGetFilterParameters retrieves the specified parameters of a filter used to compress the stream. As the stream may be compressed by several filters, the filter is specified by its index value.

```
HRESULT  PXCp_StreamGetFilterParameters(
    HPDFSTREAM hStream,
    UINT FilterNumber,
    FilterType* pFltType,
    LPPXCp_FilterParam pFltParam
);
```

Parameters

hStream
[in] *hStream* specifies the stream handle.

FilterNumber
[in] *FilterNumber* specifies the filter index.

pFltType
[out] *pFltType* specifies a pointer to a variable of the `FilterType` type.
This may be one of the following values:

Value	Parameters for encoding?	Meaning
ft_ASCIIHex	no	ASCII hexadecimal representation. Increases the length of the data.
ft_ASCII85	no	ASCII base-85 representation. Increases the length of the data.
ft_LZW	no	LZW (Lempel-Ziv-Welch) adaptive compression method.
ft_Flate	yes	zlib/deflate compression method.
ft_RLE	no	Byte-oriented run-length encoding algorithm.
ft_CCITTFax	yes	CCITT facsimile standard.
ft_DCT	yes	DCT (discrete cosine transform) algorithm.
ft_JPX	yes	Wavelet-based JPEG2000 standard.
ft_JBIG2	yes	JBIG2 standard.

pFltParam
[out] *pFltParam* specifies a pointer to a **PXCp_FilterParam** structure that contains parameters used for the filter.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Get the first filter parameters

    HPDFSTREAM hStream;

    ...
    FilterType          fltType;
    PXCp_FilterParam          fltParam;

    hr = PXCp_StreamGetFilterParameters(hStream, 0, &fltType, &fltParam);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
    // Now fltType contains the type of the filter and
    // fltParam contains its parameters.
```

3.2.7.5 PXCp_StreamGetFiltersCount

PXCp_StreamGetFiltersCount

Top Previous Next

PXCp_StreamGetFiltersCount retrieves the number of filters used for stream compression.
The number is a variable value, when the stream is decoded by **PXCp_StreamDecode** or encoded by **PXCp_StreamEncode** then the number of filters may change.

```
HRESULT  PXCp_StreamGetFiltersCount(
    HPDFSTREAM hStream,
    UINT* pFiltersCount
);
```

Parameters

hStream

[in] *hStream* specifies the stream handle.

pFiltersCount

[out] *pFiltersCount* specifies a pointer to a variable of the `UINT` type which receives the number of filters.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Get the number of filters used to encode the stream

    HPDFSTREAM hStream;

    ...

    UINT        FiltersCount = 0;
    hr = PXCp_StreamGetFiltersCount(hStream, &FiltersCount);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.7.6 PXCp_StreamGetLength

PXCp_StreamGetLength

Top Previous Next

PXCp_StreamGetLength retrieves the current length of the stream data. This length may change as the stream may be decoded or encoded.

```
HRESULT  PXCp_StreamGetLength(
    HPDFSTREAM hStream,
    __int64* pLength
);
```

Parameters

hStream
[in] *hStream* specifies the handle of the stream.

pLength
[out] *pLength* specifies a pointer to a variable of the `__int64` type, which receives the data length.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Get the length of the stream data

    HPDFSTREAM hStream;

    ...

    __int64        DataLength = 0;

    hr = PXCp_StreamGetLength(hStream, &DataLength);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.2.7.7 PXCp_StreamReadData

PXCp_StreamReadData

Top Previous Next

PXCp_StreamReadData reads data from the specified stream, starting from the specified offset. If Encoded data exists - this will be passed to the buffer "as is". If data from the stream is required in a decoded form - first use the **PXCp_StreamDecode** function - prior to reading the data stream, using **PXCp_StreamReadData**.

```
HRESULT  PXCp_StreamReadData(
    HPDFSTREAM hStream,
    void* pBuffer,
    UINT BufLength,
    __int64 Offset
);
```

Parameters

hStream
[in] *hStream* specifies the stream handle.

pBuffer
[in, out] *pBuffer* specifies a pointer to a buffer where **PXCp_StreamReadData** will place the data.

BufLength
[in] *BufLength* specifies the length of the passed buffer in bytes.

Offset
[in] *Offset* specifies the offset from the stream start point from which to read the passed data.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example on how to read data from a stream and to write to a file

   void WriteDataFromStreamToFile(HPDFSTREAM hStream, LPCWSTR FileName)
   {
       // First decode data
       UINT        FilterCnt = 0;
       HRESULT hr = PXCp_StreamGetFiltersCount(hStream, &FilterCnt);
       if (IS_DS_FAILED(hr))
       {
           // Handle error
```

```
            // ...
        }
        hr = PXCp_StreamDecode(hStream, FilterCnt);
        if (IS_DS_FAILED(hr))
        {
            // Handle error
            // ...
        }
        // At this point the stream is uncompressed

        // Get the length of the data
        __int64         StreamDataLength = 0;
        hr = PXCp_StreamGetLength(hStream, &StreamDataLength);
        if (IS_DS_FAILED(hr))
        {
            // Handle error
            // ...
        }

        // Allocate the buffer for data

        BYTE*         pBuffer = new BYTE[StreamDataLength];

        // Read data from stream to buffer
        hr = PXCp_StreamReadData(hStream, pBuffer, StreamDataLength, 0);
        if (IS_DS_FAILED(hr))
        {
            // Handle error
            // ...
        }

        // Create file:
        HANDLE hFile = ::CreateFileW(FileName, GENERIC_WRITE, FILE_SHARE_READ,
NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hFile == INVALID_HANDLE_VALUE)
        {
            // Handle error
            ...
        }

        // Now write data to file

        DWORD written = 0;
        ::WriteFile(hFile, pBuffer, StreamDataLength, &written, NULL);
        ::CloseHandle(hFile);

        // Clean up
        delete[] pBuffer;

        // done.
```

```
    }
```

3.2.7.8 PXCp_StreamRemove

PXCp_StreamRemove

Top Previous Next

PXCp_StreamRemove removes a stream from an object and deletes that stream.

```
HRESULT  PXCp_StreamRemove(
    HPDFOBJECT hObject
);
```

Parameters

hObject

[in] *hObject* specifies the handle of the PDF object from which to delete the specified stream.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

Note: After this operation the stream handle is invalid and may not be used further.

Example (C++).

```
// Removes the stream from the object

   HPDFOBJECT hObject;         // The object which contains the stream

   ...

   HPDFSTREAM hStream;

   hr = PXCp_StreamRemove(hObject);
   if (IS_DS_FAILED(hr))
   {
       // report error
       ...
   }
   ...
   // Now hObject contains no stream
```

3.2.7.9 PXCp_StreamRevertToOriginal

PXCp_StreamRevertToOriginal

Top Previous Next

Function **PXCp_StreamRevertToOriginal** attempts the reversion of the specified stream to its original state.
Whilst reading a PDF document, streams are not read into memory, to minimize application memory use. Each stream is only aware of the location of its own data and not any other's. When acquiring data from a stream - data is only then read into memory. This will occur with each action necessitating that stream data is decoded or encoded. Once this stream data is no longer required **PXCp_StreamRevertToOriginal** should be called to release memory occupied by the stream.

The stream is then restored to its original state from the data recorded within the document file. The filters count is then used to compress the stream and update any changed parameters.

However, this is not permitted for all streams, once changed or created, no original state for the stream exists and therefore no reversion processing is possible - attempting to perform this action in such an event will result in an error being returned.

```
HRESULT  PXCp_StreamRevertToOriginal(
    HPDFSTREAM hStream
);
```

Parameters

hStream
[in] *hStream* specifies the handle of the stream.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Revert stream to its original state

    HPDFSTREAM hStream;

    // Revert to original

    hr = PXCp_StreamRevertToOriginal(hStream);
    if (IS_DS_FAILED(hr))
    {
        // report error
        ...
    }
    ...
```

3.2.7.10 PXCp_StreamWrite

PXCp_StreamWrite

Top Previous Next

PXCp_StreamWrite writes uncompressed data to a stream with the specified offset from the beginning of the stream data.

If a stream is compressed before the process - information relating to compression will be discarded and the stream is uncompressed. After this operation the stream is incapable of reversion to its original state.

```
HRESULT  PXCp_StreamWrite(
    HPDFSTREAM hStream,
    const void* pBuffer,
    UINT BufLength,
    __int64 Offset
);
```

Parameters

hStream
[in] *hStream* specifies a stream handle.

pBuffer
[in] *pBuffer* specifies a pointer to a buffer to where stream data should be written.

BufLength
[in] *BufLength* specifies the length of the data in bytes.

Offset
[in] *Offset* specifies the offset value from the stream data origin to be applied.
Note: *Offset* May not be greater than the current stream data length - data may be appended or re-written but may not be applied for an invalid offset.

Return Values

If the function succeeds, the return value is a non-negative integer.
If *Offset* is greater than the stream data length **PXCp_StreamWrite** returns **PS_ERR_InvalidStreamOffset**.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Example (C++).

```
// Example on how to append data to the stream

    UINT        FilterCnt = 0;
    HRESULT hr = PXCp_StreamGetFiltersCount(hStream, &FilterCnt);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        // ...
    }
    hr = PXCp_StreamDecode(hStream, FilterCnt);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        // ...
    }at this time the stream is uncompressed
    BYTE*        pOwnData;
    UINT        OwnDataLen;
    // Get data from somewere
    // ...
    // Now append data to the end of stream
    __int64 StreamLen = 0;
    hr = PXCp_StreamGetLength(hStream, &StreamLen);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        // ...
    }
    hr = PXCp_StreamWrite(hStream, pOwnData, OwnDataLen, StreamLen);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        // ...
    }
```

3.2.7.11 PXCp_StreamWriteEncodedData

PXCp_StreamWriteEncodedData

Top Previous Next

PXCp_StreamWriteEncodedData writes the already encoded data to the stream. All present stream data will be replaced by the specified data. Filter information is also passed as stored within the stream. After this operation, stream reversion to its original state is no longer possible.

All encoded data should be written in one pass !

```
HRESULT  PXCp_StreamWriteEncodedData(
    HPDFSTREAM hStream,
    const void* pBuffer,
    UINT BufLength,
    FilterType FltType,
    LPPXCp_FilterParam pFilterParameter
);
```

Parameters

hStream

[in] *hStream* specifies the stream handle.

pBuffer

[in] *pBuffer* specifies a pointer to the buffer that contains the encoded data.

BufLength

[in] *BufLength* specifies the length of the data in the buffer in bytes.

FltType

[in] *FltType* specifies the filter type used when compressing the data.
For possible values see **PXCp_StreamGetFilterParameters** function description.

pFilterParameter

[in] *pFilterParameter* specifies a pointer to the **PXCp_FilterParam** structure with the required filter options, as defined by *FltType*.
If no additional filter parameters are required, may be set to `NULL`.

Return Values

If the function succeeds, the return value is a non-negative integer.
If the function fails, the return value is error code.
To determine if the function was successful use the defined macro's as described here: error codes page.

Remarks

Note: Any error in data or filter parameter settings will result in the production of a corrupt PDF document and will result in an error when attempts to read/decode the document/stream are subsequently made.

Example (C++).

```
// Example shows, how to write encoded data to stream
    //  data in buffer is compressed by ZIP
    // The length of data is bufLen

    HPDFSTREAM hStream;
    BYTE*         buffer;
    UINT        bufLen;

    ...
    // Prepare filter parameters

    PXCp_FilterParam        fltParam = {0};

    // Fill filter parameters with the default values for the zip encoder

    fltParam.Predictor                  = 1;
    fltParam.Colors                  = 1;
    fltParam.Bpp                  = 1;
    fltParam.Columns                  = 8;
    fltParam.EarlyChange        = 1;

    hr = PXCp_StreamWriteEncodedData(hStream, buffer, bufLen, ft_Flate,
&fltParam);
    if (IS_DS_FAILED(hr))
    {
        // Handle error
        ...
    }
    ...
```

3.3 Common Structures

Common Structures

Top Previous Next

Enter topic text here.

3.3.1 PXC_3DView

PXC_3DView

Top Previous Next

The **PXC_3DView** structure specifies a 3D view to be used by the **PXCp_Add3DAnnotationW** and **PXCp_Add3DAnnotationA** functions.

```
typedef struct _PXC_3DView {
   DWORD m_cbSize;
   WCHAR m_ExtName[128];
   double m_C2W[12];
   double m_CO;
   double m_FOV;
   COLORREF m_BackColor;
} PXC_3DView;
```

Members

m_cbSize

Specifies the size of the structure.

m_ExtName

Specifies the buffer for the external name.

m_C2W

Specifies a matrix that determines the position and orientation of the camera in world coordinates.

m_CO

Specifies the distance to the center of orbit.

m_FOV

Specifies the view of the camera (in degrees).

m_BackColor

Specifies the background color.

3.3.2 PXC_CommonAnnotInfo

PXC_CommonAnnotInfo

The **PXC_CommonAnnotInfo** structure specifies a matrix for the coordinate system transformation.

```
typedef struct _PXC_CommonAnnotInfo {
   double m_Opacity;
   COLORREF m_Color;
   DWORD m_Flags;
   PXC_AnnotBorder m_Border;
} PXC_CommonAnnotInfo;
```

Members

m_Opacity

Defines the annotation opacity level in the document. Will be used only if the PDF specification version is `1.4` or higher (**PXCp_SetSpecVersion**).

m_Color

Defines the annotation color. This color will be used as the color for the following:

- The background of the annotation's icon when closed
- The title bar of the annotation's pop-up window
- The border of the annotation link

m_Flags

A set of flags specifying various characteristics for the annotation. May be combination of the following flags:

Flag	Meaning
AF_Invisible	If set, do not display the annotation.
AF_Hidden	(*PDF 1.2*) If set, do not display or print the annotation or allow it to interact with the user, regardless of its annotation type.
AF_Print	(*PDF 1.2*) If set, print the annotation when the page is printed. If clear, never print the annotation, regardless of whether it is displayed on the screen. This can be useful, for example, for annotations representing an interactive link, which would serve no meaningful purpose on the printed page.
AF_NoZoom	(*PDF 1.3*) If set, do not scale the annotation’s appearance to match the magnification of the page. The location of the annotation on the page (defined by the upper-left corner of its annotation rectangle) remains fixed, regardless of the page magnification.
AF_NoRotate	(*PDF 1.3*) If set, do not rotate the annotation’s appearance to match the rotation of the page. The upper-left corner of the annotation rectangle remains in a fixed location on the page, regardless of the page rotation.
AF_NoView	(*PDF 1.3*) If set, do not display the annotation on the screen or allow it to

	interact with the user. The annotation may be printed (depending on the setting of the Print flag), but should be considered hidden for purposes of on-screen display and user interaction.
`AF_ReadOnly`	(*PDF 1.3*) If set, do not allow the annotation to interact with the user. The annotation may be displayed or printed (depending on the settings of the NoView and Print flags), but should not respond to mouse clicks or change its appearance in response to mouse motions.
`AF_Locked`	(*PDF 1.4*) If set, do not allow the annotation to be deleted or its properties (including position and size) to be modified by the user. However, this does not restrict changes to the annotation's contents.
`AF_ToggleNoView`	(*PDF 1.5*) If set, invert the interpretation of the NoView flag for certain events. A typical use is to have an annotation that appears only when a mouse cursor is held over it.

m_Border

Border appearance structure. See **PXC_AnnotBorder** for more information.

3.3.2.1 PXC_AnnotBorder

PXC_AnnotBorder

Top Previous Next

The **PXC_AnnotBorder** structure specifies a matrix for the coordinate system transformation.

```
typedef struct _PXC_AnnotBorder {
   double m_Width;
   PXC_AnnotBorderStyle m_Type;
   DWORD m_DashCount;
   double* m_DashArray;
} PXC_AnnotBorder;
```

Members

m_Width

The border width in points. If this value is 0, no border is drawn.

m_Type

The border style. May be any one of the following values:

Value	Meaning
`ABS_Solid`	A solid rectangle surrounding the annotation.
`ABS_Dashed`	A dashed rectangle surrounding the annotation. The dash pattern is Specified by the *m_DashArray* field.
`ABS_Bevel`	A simulated embossed rectangle that appears to be raised above the surface of the page.

ABS_Inset	A simulated engraved rectangle that appears to be recessed below the surface of the page.
ABS_Underline	A single line along the bottom of the annotation rectangle.

Note: For links (PXCp_AddLink) only `ABS_Solid` or `ABS_Dashed` style can be used.

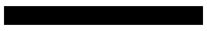
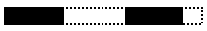
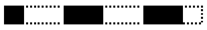
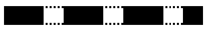
m_DashCount

Specifies the number of items in the *m_DashArray* array.

m_DashArray

Pointer to the `double` values array that define the dash pattern for border drawing.

Before beginning to stroke a path, the dash array is cycled through, adding up the lengths of dashes and gaps. When the accumulated length becomes equal or greater than the value Specifies by dash phase (), stroking of the path begins, using the dash array () cyclically from that point onward. The table below shows examples of line dash patterns.

Appearance	Dash Array And Phase	Description
	**{0, 0} 0**	No dash; solid, unbroken lines.
	**{3, 3} 0**	3 points on, 3 points off, ...
	**{2, 2} 1**	1 on, 2 off, 2 on, 2 off, ...
	**{2, 1} 0**	2 on, 1 off, 2 on, 1 off, ...

3.3.3 PXC_Watermark

PXC_Watermark

Top Previous Next

The **PXC_Watermark** structure describes the available watermark attributes and capabilities.

```
typedef struct _PXC_Watermark {
   DWORD m_Size;
   PXC_WaterType m_Type;
// Text part
   WCHAR m_FontName[64];
   DWORD m_FontWeight;
   BOOL m_bItalic;
   double m_FontSize;
   PXC_TextRenderingMode m_Mode;
   double m_LineWidth;
   COLORREF m_FColor;
   COLORREF m_SColor;
   WCHAR m_Text[256];
// Image Part
   WCHAR m_FileName[MAX_PATH];
   COLORREF m_TransColor;
   double m_Width;
   double m_Height;
   BOOL m_bKeepAspect;
// Commmon Part
   DWORD m_Align;
   double m_XOffset;
   double m_YOffset;
   double m_Angle;
   DWORD m_Opacity;
// Place Info
   PXC_WaterPlaceOrder m_PlaceOrder;
   PXC_WaterPlaceType m_PlaceType;
// Ranges
   DWORD m_NumRanges;
   LPDWORD m_Range;
// Available when library version is >= 3.30.0065
   DWORD m_ImagePageNumber;
} PXC_Watermark;
```

Members

m_Size

Specifies the size of the structure and is provided for compatibility with future versions of **PDF-XChange Pro Library** and **PDF-XChange Library** where this structure may be modified.

m_Type

Specifies the type of watermark. May have any one of the following values:

Constant	Meaning
WaterType_Text	Text watermark. All structural Image content will be ignored.
WaterType_Image	Image watermark. All structural text content will be ignored.

m_FontName

Specifies the font name for the text watermark. The font name must be a null terminated string and may not exceed 64 chars (including any terminating null-symbol). Please note that font name used must be defined as UNICODE characters.

m_FontWeight

Specifies the font weight for the text watermark. For more info see the `PXC_AddFontA` .

m_bItalic

If this parameter is `TRUE` the italic variant of the font will be used.

m_FontSize

Specifies the font size for a text watermark in points. If this value is 0, then the text watermark will be 'fitted' to the page.

m_Mode

Specifies the text drawing mode. For more information about text drawing modes, see the

`PXC_SetTextRMode` function in the **PDF-XChange Library**.

Note: Only modes from `TextRenderingMode_Fill` to `TextRenderingMode_None` are supported.

m_LineWidth

Specifies the border width for text drawing modes `TextRenderingMode_Stroke` and `TextRenderingMode_FillStroke`. For other modes this value is ignored. Line width is specified in points.

m_FColor

Specifies the color of the text (only for `TextRenderingMode_Fill` and `TextRenderingMode_FillStroke` drawing modes).

m_SColor

Specifies the color of the text border (only for `TextRenderingMode_Stroke` and `TextRenderingMode_FillStroke` drawing modes).

m_Text

A null-terminated string that specifies the text of the watermark.

m_FileName

A null-terminated string that specifies the image file name for use as a watermark.

m_TransColor

Specifies the transparent color for the image. If the high-order byte value of this member is not 0, then this member is ignored and a transparent color will not be used.

m_Width

Specifies the width of the image when placed on the page. This value is specified in points. If this member is 0, then the image will be resized to 'fit' to the page.

m_Height

Specifies height of the image when placed on the page. This value is specified in points. If this member is 0, then the image will be resized to 'fit' to the page.

m_bKeepAspect

Specifies how the **PDF-XChange Library** will scale the image during placement on the page. If this member is not 0 (zero), then the image will be resized to 'fit' in the rectangle with `m_Width` width and `m_Height` height and keeping its aspect ratio. Otherwise the aspect ratio will not be preserved.

m_Align

Specifies the horizontal and vertical alignment of the watermark text. The value is a combination of horizontal:

Value	Meaning
`TextAlign_Left`	Aligns text or image to the left.

`TextAlign_Center`	Centers text or image horizontally on the page.
`TextAlign_Right`	Aligns text or image to the right.

and vertical alignments:

Value	**Meaning**
`TextAlign_Top`	Aligns text or image to the top of the page.
`TextAlign_VCenter`	Centers text or image vertically.
`TextAlign_Bottom`	Aligns text or image to the bottom of the page.

m_XOffset

Specifies the offset of the watermark by `X` from the normal position.

m_YOffset

Specifies the offset of the watermark by `Y` from the normal position.

m_Angle

Specifies the rotation angle for the watermark text or image. *m_Angle* is the rotation angle in degrees, and its value must be in the range -90.0 to 90.0.

m_Opacity

Specifies the opacity level of the watermark (PDF specification must be 1.4 or higher to use transparency). Must be in range from 0 to 255.

m_PlaceOrder

Specifies how the watermark will be placed on the page(s) - In the foreground or background. May have any one of the following values:

Constant	**Meaning**
PlaceOrder_Background	Watermark will be placed in the background.
PlaceOrder_Foreground	Watermark will be placed in the foreground.

m_PlaceType

Specifies on which page(s) the watermark will be placed. May be a valid page number or one of the following constants:

Value	**Meaning**
PlaceType_AllPages	Watermark will be placed on all pages.
PlaceType_FirstPage	Watermark will be placed only on the first page of the document.
PlaceType_LastPage	Watermark will be placed only on the last page of the document.

PlaceType_EvenPages	Watermark will be placed only on even pages of the documentt
PlaceType_OddPages	Watermark will be placed only on odd pages of the document.
PlaceType_Range	Watermark will be placed only on pages specified by *m_Range* field.

m_NumRanges

Specifies the number of pairs in the array pointed to by *m_Range*.
This field must be 0 when *m_PlaceType* is not equal `PlaceType_Range`.

m_Range

Pointer to an array of paired `DWORD`'s values. The first element of a such pair, specifies the starting page number; The second specifies an ending page number where the watermark may be placed. This field must be `NULL` when *m_PlaceType* is not equal to `PlaceType_Range`. The array is one dimensional and should consist of `m_NumRanges * 2` elements.

m_ImagePageNumber

Specifies the page number (zero based) from an image file, which will be used as watermark. If this value points to a page which is absent within the image file, the last available page from the image will be used.

3.3.4 PXCp_BMInfo

PXCp_BMInfo

Top Previous Next

The **PXCp_BMInfo** structure displays information about a bookmark (outline) item.

```
typedef struct _PXCp_BMInfo {
   DWORD cbSize;
   DWORD Mask;
   LPWSTR TitleW;
   LPSTR TitleA;
   DWORD LengthOfTitle;
   BOOL bOpen;
   PXC_OutlineStyle Style;
   COLORREF Color;
   PXCp_BMDestination Destination;
   LPARAM UserData;
} PXCp_BMInfo;
```

Members

cbSize

Specifies the size of the structure. Usually assigned as follows:

```
PXCp_BMInfo bmInfo;
bmInfo.cbSize = sizeof(PXCp_BMInfo);
// ...
```

Mask

Specifies which fields in the structure are used during the current operation.
May be any logical combination of the following values:

Constant	Value	Meaning
BMIM_TitleA	0x0001	Title (ASCII variant).
BMIM_TitleW	0x0002	Title (UNICODE variant).
BMIM_Open	0x0004	Specifies if the item is opened when shown in a PDF viewer.
BMIM_UserData	0x0008	Specifies user data associated with the item.
BMIM_Style	0x0010	Item style.
BMIM_Color	0x0020	Item color.
BMIM_Destination	0x0040	Destination of the item (to which it points).

TitleW

Pointer to the UNICODE buffer for an item title.

TitleA

Pointer to the ASCII buffer for an item title.

LengthOfTitle

Length of the item in chars.

bOpen

Open the item when the PDF document is viewed?

Style

Specifies additional style options of the outline item entry. May be any one of the following:

Flag	Value	Meaning
OutlineStyle_Normal	0x0000	No additional styles.
OutlineStyle_Italic	0x0001	If set, item will be displayed in *italic*.
OutlineStyle_Bold	0x0002	If set, item will be displayed in **bold**.
OutlineStyle_BoldItalic	0x0003	If set, item will be displayed both in ***bold and italic***.

Color

Specifies the color of the item

Destination

Destination page (or URL) for the item. (See the PXCp_BMDestination structure description)

UserData

User defined value that is associated with the item.

3.3.4.1 PXCp_BMDestination

PXCp_BMDestination

The **PXCp_BMDestination** structure specifies the destination parameters for the outline item.

```
typedef struct _PXCp_BMDestination {
   PXC_OutlineDestination DestType;
   DWORD Mask;
   DWORD PageNumber;
   double Left;
   double Top;
   double Right;
   double Bottom;
   double Zoom;
   LPWSTR URL;
   DWORD LengthOfURL;
} PXCp_BMDestination;
```

Members

DestType

Specifies the mode in which the destination page, as specified by *PageNumber*, will be displayed, when viewed. Acceptable values are:

Constant	Meaning
Dest_Page	Retain current display location and zoom. Parameters *Left*, *Top*, *Right*, *Bottom*, and *Zoom* are not used.
Dest_XYZ	Display the page designated by *PageNumber*, with the coordinates (*Left*, *Top*) positioned at the top-left corner of the window and the contents of the page magnified by the *Zoom* factor. Parameters *Left* and *Top* are specified in points, and *Zoom* is specified in percentage points.
Dest_Fit	Display the page designated by *PageNumber*, with its contents magnified to fit the entire page within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are incompatible, the lesser of the two values is used, centering the page within the window. Parameters *Left*, *Top*, *Right*, *Bottom*, and *Zoom* are not used.
Dest_FitH	Display the page designated by *PageNumber*, with the vertical coordinate *Top*, specified in points, positioned at the top edge of the window and the contents of

	the page magnified just enough to fit the entire width of the page within the window available. Parameters *Left*, *Right*, *Bottom*, and *Zoom* are not used.
`Dest_FitV`	Display the page designated by *PageNumber*, with the horizontal coordinate *Left*, specified in points, positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire height of the page within the window. Parameters *Top*, *Right*, *Bottom*, and *Zoom* are not used.
`Dest_FitR`	Display the page designated by *PageNumber*, with its contents magnified just enough to fit the rectangle specified by the coordinates *Left*, *Top*, *Right*, and *Bottom* entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, the lesser of the two values is used, centering the rectangle within the window. Parameters *Left*, *Top*, *Right*, and *Bottom* are specified in points.
`Dest_FitB`	Display the page designated by *PageNumber*, with its contents magnified just enough to fit its bounding box entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are incompatible, the lesser of the two values is used, centering the bounding box within the window. Parameters *Left*, *Top*, *Right*, *Bottom*, and *Zoom* are not used.
`Dest_FitBH`	Display the page designated by *PageNumber*, with the vertical coordinate's top positioned at the *Top*, specified in points, to the edge of the window and the contents of the page magnified just enough to fit the entire width of its bounding box within the window. Parameters *Left*, *Right*, *Bottom*, and *Zoom* are not used.
`Dest_FitBV`	Display the page designated by *PageNumber*, with the horizontal coordinate left positioned at the *Left*, specified in points, to the edge of the window and the contents of the page magnified just enough to fit the entire height of its bounding box within the window. Parameters *Top*, *Right*, *Bottom*, and *Zoom* are not used.
`Dest_Y`	Same as DST_XYZ, but specifies only `Y` coordinate (*Top*, in points), leave others parameters unchanged. Parameters *Left*, *Right*, *Bottom*, and *Zoom* are not used.

Mask

Specifies strcuture parameters which are set to `null` or if the destination is set to a `URL` instead of pointing to the page in the document.

In PDF document destination parameters values `0` and `null` are different. For more information about the destination parameters, developers should see Adobe's comprehensive documentation for the PDF format freely available from the Adobe web site.

This parameter must be zero or a combination of the following values:

Flag	Value	Meaning
BMDF_DestIsURL	0x0001	Destination is URL.
BMDF_LeftIsNULL	0x0002	Left is set to `null`.
BMDF_TopIsNULL	0x0004	Top is set to `null`.
BMDF_RightIsNULL	0x0008	Rigth is set to `null`.
BMDF_BottomIsNULL	0x0010	Bottom is set to `null`.
BMDF_ZoomIsNULL	0x0020	Zoom is set to `null`.

PageNumber

Specifies page number.

Left

Specifies the left value for the page destination. See *DestType* for details.

Top

Specifies the top value for the page destination. See *DestType* for details.

Right

Specifies the right value for the page destination. See *DestType* for details.

Bottom

Specifies the bottom value for the page destination. See *DestType* for details.

Zoom

Specifies the zoom value for the page destination. See *DestType* for details.

URL

Specifies a pointer to the UNICODE buffer for `URL`.

LengthOfURL

Specifies the number of characters in the *URL* buffer.

Note: When retrieving a *URL* for the first time set the *URL* to `NULL` and obtain from the appropriate field the necessary number of characters for the *URL* buffer.

Comments

Parameters *Style* and *Color* are valid only if you have called and made use of the function **PXCp_SetSpecVersion** and set your PDF format specification to at least `SpecVersion14` or later.

3.3.5 PXCp_CompressParam

PXCp_CompressParam

Top Previous Next

The **PXCp_CompressParam** structure specifies parameters for compression filters applied by the **PXCp_StreamEncode** function.

```
typedef struct _PXCp_CompressParam {
   DWORD Width;
   DWORD Height;
   BYTE NumOfComponent;
   BYTE Quality;
   DWORD CompressionLevel;
   LONG K;
   BOOL bEndOfLine;
   BOOL bEncodedeLineAlign;
   BOOL bEndOfBlocks;
} PXCp_CompressParam;
```

Members

Width

Specifies the width of an image in pixels.
This parameter is used with the `ft_DCT`, `ft_JPX` and `ft_CCITTFax` filters.

Height

Specifies the height of an image in pixels.
This parameter is used with the `ft_DCT`, `ft_JPX` and `ft_CCITTFax` filters.

NumOfComponent

Specifies the number of color components for an image. Possible values are `1` (for grayscale) and `3` (for color) images.
This parameter is used with the `ft_DCT` and `ft_JPX`.

Quality

Specifies the quality of `JPEG` or `JPEG 2000` filters.
Quality set may be in the range from 1 (worst) to 100 (best), lower values improve file size and reduce image quality.
This parameter is used with the `ft_DCT` and `ft_JPX`.

CompressionLevel

Specifies the compression level for `ft_Flate` (ZIP) filter.
Maximum value is `9` which is usually set for this filter type.
This parameter is used only with the `ft_Flate` filter.

K

Specifies a code identifying the encoding scheme used.
Please refer to Adobe's comprehensive documentation for the PDF format, available free from the Adobe web site.
This parameter is used only with the `ft_CCITTFax` filter.

bEndOfLine

Specifies a flag indicating whether end-of-line bit patterns are required to be present in the encoding.
Please refer to Adobe's comprehensive documentation for the PDF format available free from the Adobe web site.

This parameter is used only with the `ft_CCITTFax` filter.

bEncodedeLineAlign

Specifies a flag indicating whether the filter expects extra 0 bits before each encoded line so that the line begins on a byte boundary.

Please refer to Adobe's comprehensive documentation for the PDF format available free from the Adobe web site.

This parameter is used only with the `ft_CCITTFax` filter.

bEndOfBlocks

Specifies a flag indicating whether the filter expects the encoded data to be terminated by an end-of-block pattern.

Please refer to Adobe's comprehensive documentation for the PDF format available free from the Adobe web site.

This parameter is used only with the `ft_CCITTFax` filter.

3.3.6 PXCp_ContentPlaceInfo

PXCp_ContentPlaceInfo

Top Previous Next

This **PXCp_ContentPlaceInfo** structure is used to determine which of the source PDF document pages should be combined with pages from the destination PDF document and determine how they should be combined when using the **PXCp_PlaceContents** function.

```
typedef struct _PXCp_ContentPlaceInfo {
   DWORD DestPage;
   DWORD SrcPage;
   DWORD Alignment;
} PXCp_ContentPlaceInfo;
```

Members

DestPage

Zero based index of the destination PDF page in the destination PDF document.

SrcPage

Zero based index of the source PDF page in the source PDF document.

Alignment

Combination of placement flags, that determine how the source page will be placed on the destination page. This should be a combination of one of the horizontal alignment flags (`CPA_Hor???`), and the vertical alignment flag (`CPA_Ver???`) and any additionally required set of available flags (see table below).

Flag	Value	Meaning
CPA_HorLeft	0x0000	Align placed content to the left.
CPA_HorCenter	0x0001	Center placed content.
CPA_HorRigth	0x0002	Align placed content to the right.
CPA_HorFit	0x0003	Fit source page width to destination page width.
CPA_VerBottom	0x0000	Align placed content to the bottom.
CPA_VerCenter	0x0010	Center placed content.
CPA_VerTop	0x0020	Align placed content to the top.
CPA_VerFit	0x0030	Fit source page height to destination page height.
CPA_Foreground	0x0100	Place source page content as foreground if set, otherwise placed as background
CPA_KeepAspect	0x0200	Keep source page aspect ratio.

If `CPA_KeepAspect` flag is not set the page may be distorted. If set, the source page dimensions will be as follows:
- if no Fit flags (`CPA_HorFit` or `CPA_VerFit`) are set the destination page sizes will match those of the original
- if only the horizontal alignment flag is `CPA_HorFit` the source page will be scaled to fit the destination page width, positioned using the prescribed vertical alignment flag
- if only the vertical alignment flag is `CPA_VerFit` set, the source page will be scaled to fit the destination page height and then positioned using the horizontal alignment flag
- if both Fit flags (`CPA_HorFit` and `CPA_VerFit`) are set the source page will be scaled to fit the destination page and then centered.

3.3.7 PXCp_CopyPageRange

PXCp_CopyPageRange

Top Previous Next

The **PXCp_CopyPageRange** structure specifies a Range of page's to be used by the **PXCp_InsertPagesTo** function.

```
typedef struct _PXCp_CopyPageRange {
   DWORD StartPage;
   DWORD EndPage;
   LONG InsertBefore;
   DWORD Reserved;
} PXCp_CopyPageRange;
```

Members

StartPage

Specifies the first page in the required Range.

EndPage

Specifies the last page in the required Range to be copied.

InsertBefore

Specifies the page in the target document before which the Range should be inserted.
If this field is set to `-1` then the Range is appended to the end of the target document.

Reserved

This is a reserved field and should be set to `0`.

Remarks

`Important!` PDF file page numbering is Zero based - the initial page being numbered `0`, the second page value being page `1` and so on.

3.3.8 PXCp_FilterParam

PXCp_FilterParam

Top Previous Next

The **PXCp_FilterParam** structure details parameters used when the data stream was originally compressed. This structure is used in the functions **PXCp_StreamGetFilterParameters** and **PXCp_StreamWriteEncodedData**.

Note: Please refer to Adobe's comprehensive documentation for the PDF format available free from the Adobe web site for more details on filters and their parameters.

```
typedef struct _PXCp_FilterParam {
   DWORD Predictor;
   DWORD Colors;
   DWORD Bpp;
   DWORD Columns;
   DWORD EarlyChange;
   int K;
   BOOL EndOfLine;
   BOOL EncodedByteAlign;
   DWORD ccitt_Columns;
   DWORD Rows;
   BOOL EndOfBlock;
   BOOL BlackIs1;
   DWORD DamagedRowsBeforeError;
   void* hObject;
   DWORD ColorTransform;
} PXCp_FilterParam;
```

Members

// --------- Optional parameters for LZWDecode and FlateDecode filters

Predictor

A code that selects the predictor algorithm, if any.
If the value of this entry is 1, the filter assumes that the normal algorithm was used to encode the data, without prediction. If the value is greater than 1, the filter assumes that the data was altered before being encoded, and the Predictor selects the predictor algorithm.
Default value: 1.

Colors

(Used only if *Predictor* is greater than 1).
The number of interleaved color components per sample. Valid values are 1 to 4 in PDF 1.2 or earlier, and 1 or greater in PDF 1.3 or later.
Default value: 1.

Bpp

The number of bits used to represent each color component in a sample.
Valid values are 1, 2, 4, 8, and (in PDF 1.5) 16.
Default value: 8.

Columns

The number of samples in each row.
Default value: 1

EarlyChange

(LZWDecode only)
An indication of when to increase the code length. If the value of this entry is 0, code length increases are postponed as long as possible. If it is 1, they occur one code early. This parameter is included because LZW sample code distributed by some vendors increases the code length one code earlier than necessary.
Default value: 1.

// --------- Optional parameters for the CCITTFaxDecode filter

K

A code identifying the encoding scheme used:

- less then 0 Pure two-dimensional encoding (Group 4)
- equal to 0 Pure one-dimensional encoding (Group 3, 1-D)
- greater then 0 Mixed one and two-dimensional encoding (Group 3, 2-D), in which a line encoded one-dimensionally can be followed by at most K - 1 lines encoded two-dimensionally.

The filter distinguishes among negative, zero, and positive values of K to determine how to interpret the encoded data; however, it does not distinguish between different positive K values.
Default value: 0.

EndOfLine

A flag indicating whether end-of-line bit patterns are required to be present in the encoding. The CCITTFax filter always accepts end-of-line bit patterns, but requires them only if EndOfLine is true.
Default value: false.

EncodedByteAlign

A flag indicating whether the filter expects extra 0 bits before each encoded line so that the line begins on a byte boundary. If true, the filter skips over encoded bits to begin decoding each line at a byte boundary. If false, the filter does not expect extra bits in the encoded representation.
Default value: false.

ccitt_Columns

The width of the image in pixels. If the value is not a multiple of 8, the filter adjusts the width of the unencoded image to the next multiple of 8, so that each line starts on a byte boundary.
Default value: 1728.

Rows

The height of the image in scan lines. If the value is 0 or absent, the image's height is not predetermined, and the encoded data must be terminated by an end-of-block bit pattern or by the end of the filter's data.
Default value: 0.

EndOfBlock

A flag indicating whether the filter expects the encoded data to be terminated by an end-of-block pattern, overriding the Rows parameter. If false, the filter stops when it has decoded the number of lines indicated by Rows or when its data has been exhausted, whichever occurs first. The end-of-block pattern is the CCITT end-of-facsimile- block (EOFB) or return-to-control (RTC) appropriate for the K parameter.
Default value: true.

BlackIs1

A flag indicating whether 1 bits are to be interpreted as black pixels and 0 bits as white pixels, the reverse of the normal PDF convention for image data.
Default value: false.

DamagedRowsBeforeError

The number of damaged rows of data to be tolerated before an error occurs. This entry applies only if EndOfLine is true and K is non negative. Tolerating a damaged row means locating its end in the encoded data by searching for an EndOfLine pattern and then substituting decoded data from the previous row if the previous row was not damaged, or a white scan line if the previous row was also damaged.
Default value: 0.

// --------- Optional parameter for the JBIG2Decode filter

hObject

A stream containing the JBIG2 global (page 0) segments. Global segments must be placed in this stream even if only a single JBIG2 image XObject refers to it.

ColorTransform

A code specifying the transformation to be performed on the sample values:

- 0 No transformation.
- 1 If the image has three color components, transform RGB values to YUV before encoding and from YUV to RGB after decoding. If the image has four components, transform CMYK values to YUVK before encoding and from YUVK to CMYK after decoding. This option is ignored if the image has one or two color components.

Note: The RGB and YUV used here have nothing to do with the color spaces defined as part of the Adobe imaging model. The purpose of converting from RGB to YUV is to separate luminance and chrominance information (see below).

The default value of ColorTransform is 1 if the image has three components and 0 otherwise. In other words, conversion between RGB and YUV is performed for all three-component images unless explicitly disabled by setting ColorTransform to 0. Additionally, the encoding algorithm inserts an Adobe defined marker code in the encoded data indicating the ColorTransform value used. If present, this marker code overrides the ColorTransform value given to DCTDecode. Thus it is necessary to specify ColorTransform only when decoding data that does not contain the Adobe-defined marker code.

3.3.9 PXCp_SaveImageOptions

PXCp_SaveImageOptions

Top Previous Next

The **PXCp_SaveImageOptions** structure describes the available options for saving an extracted image file to a disk file.

```
typedef struct _PXCp_SaveImageOptions {
   DWORD fmtID;
   DWORD imgType;
   BOOL bConvertToGray;
   DWORD bDither;
   BOOL bWriteAlpha;
   DWORD xDPI;
   DWORD yDPI;
   DWORD CompressionMethod;
   DWORD CompressionLevel;
   BOOL bAppendToExisting;
} PXCp_SaveImageOptions;
```

Members

fmtID

Specifies the image format identifier. Possible values are:

Format identifier	Hex Value	Meaning
PRO_FMT_BMP_ID	0x424d5020	Windows Bitmap Files
PRO_FMT_PNG_ID	0x504e4720	Portable Network Graphic

`PRO_FMT_GIF_ID`	0x47494620	Compuserve GIF
`PRO_FMT_PGM_ID`	0x50474d20	Portable Graymap
`PRO_FMT_PBM_ID`	0x50424d20	Portable Bitmap
`PRO_FMT_PPM_ID`	0x50504d20	Portable Pixelmap
`PRO_FMT_JP2K_ID`	0x4a50324b	Jpeg 2000
`PRO_FMT_JPEG_ID`	0x4a504547	Jpeg
`PRO_FMT_JNG_ID`	0x4a4e4720	Jpeg Network Graphic
`PRO_FMT_TGA_ID`	0x54474120	Truevision Targa
`PRO_FMT_TIFF_ID`	0x54494646	TIFF Files
`PRO_FMT_WBMP_ID`	0x57424d50	Wireless mono bitmap
`PRO_FMT_PCX_ID`	0x50435820	PC Paintbrush File Format
`PRO_FMT_DCX_ID`	0x44435820	Multipage PCX

imgType

Specifies image type (bits per pixel, color component(s))

Image type	**Value**	**Meaning**	**Comments**
`ImType_bw_1bpp`	1	black and white, 1 bpp	
`ImType_index_1bpp`	2	indexed, 1 bpp	
`ImType_index_2bpp`	3	indexed, 2 bpp	
`ImType_index_3bpp`	4	indexed, 3 bpp	May be used only with `GIF` images
`ImType_index_4bpp`	5	indexed, 4 bpp	
`ImType_index_5bpp`	6	indexed, 5 bpp	May be used only with `GIF` images
`ImType_index_6bpp`	7	indexed, 6 bpp	May be used only with `GIF` images
`ImType_index_7bpp`	8	indexed, 7 bpp	May be used only with `GIF` images
`ImType_index_8bpp`	9	indexed, 8 bpp	
`ImType_rgb_15`	10	RGB, 15 bpp	`BMP` and `TGA` only
`ImType_rgb_16_565`	11	RGB, 16 bpp	R:5, G:6, B:5, `BMP` only
`ImType_rgb_16_5551`	12	RGB, 16 bpp	R:5, G:5, B:5, `BMP` and `TGA` only

`ImType_rgb_24bpp`	13	RGB, 24 bpp	
`ImType_rgb_32bpp`	14	RGB, 32 bpp	
`ImType_rgb_36bpp`	15	RGB, 36 bpp	`JPEG` and `JNG` only
`ImType_rgb_48bpp`	16	RGB, 48 bpp	`PNG` only
`ImType_gray_8bpp`	17	grayscale, 8 bpp	
`ImType_gray_12bpp`	18	grayscale, 8 bpp	`JPEG` and `JNG` only
`ImType_gray_16bpp`	19	grayscale, 8 bpp	`PNG` only

bConvertToGray

Boolean value that specifies whether the image should be converted to grayscale or not.

bDither

Value that specifies if the image is to be saved as dithered or not. (`1` - dither, `0` - don't dither)

bWriteAlpha

Boolean value that specifies whether to write an alpha channel. Alpha channel may be specified for `PNG`, `JNG` and `TIFF` formats.

xDPI

Specifies the `X` resolution of the image.

yDPI

Specifiec the `Y` resolution of the image.

CompressionMethod

Specifies comression method.

Compression	Value	Meaning
ImCompression_None	0	No compression
ImCompression_ZIP	1	ZIP
ImCompression_JPEG	2	JPEG
ImCompression_LZW	3	LZW
ImCompression_RLE	4	Run Length Encoding
ImCompression_CCITT3_1d	5	CCITT modified Huffman RLE
ImCompression_CCITT3_2d	6	CCITT Group 3 fax
ImCompression_CCITT4	7	CCITT Group 4 fax
ImCompression_CCITT_RLE	8	CCITT RLE with word alignment

W

ImCompression_JPEG2k	9	Jpeg 2000 compression
ImCompression_ASCII	10	ASCII compression (only for `PBM, PGM, PPM`)
ImCompression_Binary	11	Binary compression (only for `PBM, PGM, PPM`)

CompressionLevel

Specifies compression level.

bAppendToExisting

Boolean value that specifies whether to append to an existing file or overwrite an existing image file should it exist.

Comments

Image formats may have the following image types:

Image format	**Image type**
PRO_FMT_BMP_ID	ImType_index_1bpp, ImType_index_4bpp, ImType_index_8bpp, ImType_rgb_15, ImType_rgb_16_565, ImType_rgb_16_5551, ImType_rgb_24bpp, ImType_rgb_32bpp
PRO_FMT_GIF_ID	ImType_index_1bpp, ImType_index_2bpp, ImType_index_3bpp, ImType_index_4bpp, ImType_index_5bpp, ImType_index_6bpp, ImType_index_7bpp, ImType_index_8bpp
PRO_FMT_PNG_ID	ImType_index_1bpp, ImType_index_2bpp, ImType_index_4bpp, ImType_index_8bpp, ImType_gray_8bpp, ImType_gray_16bpp, ImType_rgb_24bpp, ImType_rgb_48bpp
PRO_FMT_PBM_ID	ImType_bw_1bpp
PRO_FMT_PGM_ID	ImType_gray_8bpp
PRO_FMT_PPM_ID	ImType_rgb_24bpp
PRO_FMT_JPEG_ID	ImType_gray_8bpp, ImType_gray_12bpp, ImType_rgb_24bpp, ImType_rgb_36bpp
PRO_FMT_JP2K_ID	ImType_gray_8bpp, ImType_rgb_24bpp
PRO_FMT_JNG_ID	ImType_gray_8bpp, ImType_gray_12bpp, ImType_rgb_24bpp, ImType_rgb_36bpp
PRO_FMT_TGA_ID	ImType_index_8bpp, ImType_gray_8bpp, ImType_rgb_15, ImType_rgb_16_5551, ImType_rgb_24, ImType_rgb_32
PRO_FMT_TIFF_ID	ImType_bw_1bpp, ImType_index_4bpp, ImType_index_8bpp, ImType_gray_8bpp, ImType_rgb_24bpp, ImType_rgb_32bpp

PRO_FMT_WBMP_ID	ImType_bw_1bpp
PRO_FMT_PCX_ID	ImType_bw_1bpp, ImType_index_4bpp, ImType_index_8bpp, ImType_rgb_24bpp
PRO_FMT_DCX_ID	ImType_bw_1bpp, ImType_index_4bpp, ImType_index_8bpp, ImType_rgb_24bpp

Image formats may use the following compression methods:

Image format	Compression
PRO_FMT_BMP_ID	ImCompression_None
PRO_FMT_PNG_ID	ImCompression_ZIP
PRO_FMT_GIF_ID	ImCompression_None
PRO_FMT_PGM_ID	ImCompression_ASCII, ImCompression_Binary
PRO_FMT_PBM_ID	ImCompression_ASCII, ImCompression_Binary
PRO_FMT_PPM_ID	ImCompression_ASCII, ImCompression_Binary
PRO_FMT_JP2K_ID	ImCompression_JPEG2k
PRO_FMT_JPEG_ID	ImCompression_JPEG
PRO_FMT_JNG_ID	ImCompression_JPEG
PRO_FMT_TGA_ID	ImCompression_None, ImCompression_RLE
PRO_FMT_TIFF_ID	ImCompression_LZW, ImCompression_JPEG, ImCompression_RLE, ImCompression_CCITT3_1d, ImCompression_CCITT3_2d, ImCompression_CCITT4, ImCompression_CCITT_RLEW
PRO_FMT_WBMP_ID	ImCompression_None
PRO_FMT_PCX_ID	ImCompression_None
PRO_FMT_DCX_ID	ImCompression_None

For those image formats that may have only one or only `ImCompression_None` compression methods, this need not be set.

Remarks

This structure is used by the **PXCp_SaveDocImageIntoFileW** and **PXCp_PageSaveThumbnailToFile** functions.

3.3.10 PXP_TEFontInfo

PXP_TEFontInfo

Top Previous Next

The **PXP_TEFontInfo** structure is used by PXCp_ET_GetFontInfo and contains PDF font information.

```
typedef struct _PXP_TEFontInfo {
   DWORD cbSize;
   DWORD Flags;
   PXP_TE_FontQuality Quality;
   PXP_TE_FontType Type;
   double ItalicAngle;
   double Ascent;
   double Descent;
} PXP_TEFontInfo;
```

Members

cbSize

Contains the size of the structure in bytes.

Flags

The value is an unsigned 32-bit integer containing flags specifying various characteristics of the font. Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order). The table shows the meanings of the flags; all undefined flag bits are reserved and must be set to 0.

Bit Position	Hex values	Name	Meaning
1	0x00000001	**FixedPitch**	All glyphs have the same width (as opposed to proportional or variable-pitch fonts, which have differing widths).
2	0x00000002	**Serif**	Glyphs have serifs, which are short strokes drawn at an angle on the top and bottom of glyph stems (as opposed to sans serif fonts, which do not).
3	0x00000004	**Symbolic**	Font c ontains glyphs outside the Adobe standard Latin character set. This flag and the `Nonsymbolic` flag cannot both be set or both be clear.
4	0x00000008	**Script**	Glyphs resemble cursive handwriting.
6	0x00000020	**Nonsymbolic**	Font uses the Adobe standard Latin character set or a subset of it.
7	0x00000040	**Italic**	Glyphs have dominant vertical strokes that are slanted.
17	0x00010000	**AllCap**	Font contains no lowercase letters; typically used for display purposes such as titles or

			headlines.
18	0x00020000	**`SmallCap`**	Font contains both uppercase and lowercase letters. The uppercase letters are similar to ones in the regular version of the same typeface family. The glyphs for the lowercase letters have the same shapes as the corresponding uppercase letters, but they are sized and their proportions adjusted so that they have the same size and stroke weight as lowercase glyphs in the same typeface family.
19	0x00040000	**`ForceBold`**	Determines whether bold glyphs are painted with extra pixels even at very small text sizes. Typically, when glyphs are painted at small sizes on very low-resolution devices such as display screens, features of bold glyphs may appear only 1 pixel wide. Because this is the minimum feature width on a pixel-based device, ordinary (nonbold) glyphs also appear with 1-pixel-wide features, and so cannot be distinguished from bold glyphs. If the `ForceBold` flag is set, features of bold glyphs may be thickened at small text sizes.

Quality

Defines the accuracy of the text extraction for elements using this PDF font. PDF documents store text as a sequence of indexes that depend on the used font (not unicode). The text library must know how to translate such indexes into unicode values for each used font. There are several possibilities to determine translation:

1. The PDF font has a special ToUnicode table, which defines mapping from indexes to unicode values. This is the best case.

2. The PDF font uses a standard encoding or a custom encoding method which uses only standard glyph names. In this case there is a defined method to map character indexes into correspondong unicode values.

3. The PDF font uses custom encoding with non-standard glyph names or built-in encoding (typical situation for symbolic fonts). In this case there is no defined method to reconstruct mapping from indexes to unicode values, although there may be possiblities.

Please do not forget that PDF fonts MUST contain info to display characters correctly (actually mapping from indexes into glyph numbers), but MAY NOT contain info about mapping indexes to unicode values, and character indexes depend only on the PDF file creator. Could be any one of the following value:

Font Quality	Value	Meaning

TEFQ_ToUnicode	0	Using ToUnicode table.
TEFQ_Encoding	1	Using font encoding.
TEFQ_BuiltIn	2	Using built-in font data (reserved).
TEFQ_Approximated	3	Reserved for future usage
TEFQ_NotKnown	-1	There is no known way to determine how to obtain unicode codes for characters in this font.

Type

Defines the type of the PDF font. Could be one of the following value:

Font Type	Value	Meaning
TEFT_Unknown	0	Unknown font type
TEFT_TrueType	1	TrueType font
TEFT_Type1	2	Type 1 font
TEFT_Type3	3	Type 3 font

ItalicAngle

The angle, expressed in degrees counter clockwise from the vertical, of the dominant vertical strokes of the font. (For example, the 9-o'clock position is 90 degrees, and the 3-o'clock position is –90 degrees.) The value is negative for fonts that slope to the right, as almost all italic fonts do.

Ascent

The maximum height above the baseline reached by glyphs in this font, excluding the height of glyphs for accented characters.

Descent

The maximum depth below the baseline reached by glyphs in this font. The value is a negative number.

3.3.11 PXP_TETextComposeOptions

PXP_TETextComposeOptions

Top Previous Next

The **PXP_TETextComposeOptions** structure contains text extraction/composing options.

```
typedef struct _PXP_TETextComposeOptions {
   DWORD cbSize;
   DWORD Flags;
   double MergeDistanceX;
   double MergeDeltaY;
   PXP_TE_TextComposeMethod ComposeMethod;
   PXP_TE_AddSpaces AddSpaces;
   double MinAddSpaceDistance;
   PXP_TE_UnecodedCharacters Undecoded;
   WCHAR ReplaceBy;
   WORD Reserved;
} PXP_TETextComposeOptions;
```

Members

cbSize

Specifies the size of the structure. Usually assigned as follows:

```
PXP_TETextComposeOptions tco;
tco.cbSize = sizeof(PXP_TETextComposeOptions);
// ...
```

Flags

Reserved for future usage. Must be set to zero for compatibility.

MergeDistanceX

Specifies the maximum X axis distance to merge sequential text elements. If this parameter is negative then all text elements on one line will be merged.

MergeDeltaY

Specifies the maximum Y delta to merge sequential text elements. This parameter is multiplied by font size and cannot be negative.

ComposeMethod

Specifies the text composition method. May be one of the following values:

Constant	Value	Meaning
TETCM_PreserveOrder	0	Preserve text block drawing order as in PDF document.
TETCM_PreservePositions	1	Preserve text position on page.

AddSpaces

Specifies how much additional space should be inserted between text elements. May be one of the following values:

Constant	Value	Meaning
TEAS_None	0	Adds no additional space.
TEAS_Single	1	Adds a single space.
TEAS_Proportional	2	Add several spaces, depending on distance between blocks.

MinAddSpaceDistance

Specifies minimal distance at which point at least one space will be added.

To determine whether to add spacing or not it is multiplied by the width of a single space and then compared with the actual distance. If the distance is greater or equal then one or more spaces will be added.
This parameter must be greater than zero.

Undecoded

Specifies how to handle characters that have no valid unicode representations. May be any one of the following values:

Constant	Value	Meaning
TEUC_Remove	0	Remove undecoded characters from resulting text.
TEUC_KeepOriginal	1	Leave original character indexes as unicode values for undecoded characters (default).
TEUC_Replace	2	Replace undecoded characters with specified character.

ReplaceBy

Specifies character code for which all undecoded characters will be replaced when `Undecoded` is `TEUC_Replace`.

Reserved

Reserved. Please set to zero for compatibility.

3.3.12 PXP_TextElement

PXP_TextElement

Top Previous Next

The **PXP_TextElement** structure contains detailed information about a text element.

```
typedef struct _PXP_TextElement {
   DWORD cbSize;
   DWORD mask;
   WCHAR* Characters;
   double* Offsets;
   DWORD Count;
   DWORD FontIndex;
   double FontSize;
   PXC_Matrix Matrix;
   double CharSpace;
   double WordSpace;
   double Th;
   double Leading;
   double Rise;
   COLORREF FillColor;
   COLORREF StrokeColor;
   PXC_TextRenderingMode RenderingMode;
} PXP_TextElement;
```

Members

cbSize

Specifies the size of the structure. Usually assigned as follows:

```
PXP_TextElement te;
te.cbSize = sizeof(PXP_TextElement);
// ...
```

mask

Specifies which fields in the structure are used during the current operation.
May be any logical combination of the following values:

Constant	Value	Meaning
`PTEM_Text`	`0x0001`	Character codes (unicode or original PDF indexes).
`PTEM_Offsets`	`0x0002`	Character offsets (or character deltas).
`PTEM_Matrix`	`0x0004`	Element matrix.
`PTEM_FontInfo`	`0x0008`	Font index and font size.
`PTEM_TextParams`	`0x0010`	Other pdf-specific text showing parameters (`CharSpace`, `WordSpace`, `Th`, `Leading`, `Rise`, `FillColor`, `StrokeColor`, `RenderingMode` fields).

Characters

Pointer to the buffer where character codes will be stored (including terminating null-character).

Offsets

Pointer to the buffer where character offsets will be stored. Offset of the last character (terminating null-character) is element length.

Count

Count of characters, including terminating null character.
If any of following conditions is true this field will receive the required count and the `Characters` and `Offsets` buffers will be not filled:
1. `Count` is zero.
2. Both `Characters` and `Offsets` are `null`.
3. `mask` field does not have `PTEM_Text` and `PTEM_Offsets` flags set.

FontIndex

Index of font. You can retrieve the font info using **PXCp_ET_GetFontInfo**, **PXCp_ET_GetFontName**, and **PXCp_ET_GetFontStyle**.

FontSize

Font size.

Matrix

[out] Matrix which defines element transformation parameters. It is a combination of all transformations (position, scaling, rotation and so on) that have been used to produce the resulting output. The `e` and `f` parameters contain the starting location of the text element on the page, and are based on the mediabox coordinates.

See PDF Reference 1.6, section 4.2.2 Common Transformations for more information.

CharSpace

Additional spacing between characters.

WordSpace

Additional spacing between words; adds after each space character (code 32).

Th

Horizontal text scaling, as a percentage. For non-scaled text this parameter is 100.0.

Leading

Interval between lines of text. Informative only, as all text elements are one line.

Rise

Vertical text displacement. If the `GTEF_OriginalDeltas` flag is specified in **PXCp_ET_GetElement**, this displacement is not included in the `Matrix` field, otherwise `Matrix` will be corrected to include it.

FillColor

Text fill color.

StrokeColor

Txt stroke color (outline color).

RenderingMode

Text rendering mode.

4 Error Handling

Error Handling

Top Previous Next

Most functions return an `HRESULT` value representing an error code. This code may indicate success or failure & provides a simple means to determine success or otherwise of a function: If the most significant bit or result is set to 1 then the specified error occurred, otherwise the function succeeded. Here are two simple macros for C/C++ which apply these checks:

```
#define IS_DS_SUCCESSFUL(x)                 (((x) & 0x80000000) == 0)
#define IS_DS_FAILED(x)                         (((x) & 0x80000000) != 0)
```

Note: It is strongly recommended to always use the specified (or equivalent macro's) to establish if the function call was successful or otherwise. A simple comparison with `0` (`zero`) will usually provide eroneous and unreliable results described in the following example scenario's.

1. A Function may return a warning with a code that is not equal to `zero` (and is also not negative!). This

usually means that the function has not failed and is providing additional information about the call. i.e. The function returns a default value etc. For more information see the description provided for each particular function.

To determine if the return value is generating a warning we provide the `IS_DS_WARNING` macro's.

2. Some functions will return a meaningful, positive integer. i.e. **PXC_GetImageColors** returns the number of colors in the image (this number is always integer!), otherwise (when and if an error occurs) a (negative) error value is returned.

   This would be the correct syntax to check for the error status of the **PXC_GetImageColors** function :

```
    DWORD ColorCount = PXC_GetImageColors(doc, image);

    if (IS_DS_FAILED(ColorCount))
    {
        // An error occurred!
        // Manage the error accordingly to provide an orderly exit from the
function call
        ...
    }
    else
    {
        // 'ColorCount' contains number of colors in the image
        // and it is positive number
        ...
    }
```

The example code below demonstrates how NOT to provide error checking in your code :

```
    DWORD ColorCount = PXC_GetImageColors

    if (ColorCount == 0)
    {
        // treat as success
        ...

        (this is not true as a positive return value was received!)
        ...
    }
    else
    {
        // treat as error
        (Incorrect as the retrun value has not been adequately identified and
this is unreliable!)
        ...
    }
```

4.1 LIB Error Handling

LIB Error Handling

Top Previous Next

The PDF-XChange Tools Library (pxclib40.dll) has built in error handling functions:

Topic	**Description**
LIB Error Codes	Common errors returned by all Library (PXC_...) functions.
PXC_Err_FormatErrorCode	Returns the Error Text for any `HRESULT` returned by Library functions.
PXC_Err_FormatFacility	Returns the Facility for any `HRESULT` returned by Library functions.
PXC_Err_FormatSeverity	Returns the Error Severity for any `HRESULT` returned by Library functions.
Example (C++)	Example using the PXC_Err_... functions

4.1.1 LIB Error Codes

LIB Error Codes

Top Previous Next

Most frequently returned error codes are listed in the table below, however functions may return additional codes which are not listed here. There are 3 further functions available for dealing with error's and may give additional information relating to a specific error: **PXC_Err_FormatSeverity**, **PXC_Err_FormatFacility**, **PXC_Err_FormatErrorCode**. A code example of how to use this table is provided below the table itself. Please note that this function will provide information about **all** error codes which may be returned by the PDF-XChange Tools Components libraries.

Constant	**Value**	**Description**
PXC_ERR_NOTIMPL	0x820404b0	Not implemented
PXC_ERR_INV_ARG	0x82040001	Invalid argument
PXC_ERR_MEMALLOC	0x820403e8	Insufficient memory
PXC_ERR_USER_BREAK	0x820401f4	Operation aborted by user
PXC_ERR_INTERNAL	0x82040011	Internal error
PXC_ERR_NOT_INITIALIZED	0x82042710	Not initialized
PXC_ERR_BAD_SPEC_VERSION	0x82042711	Not supported in the designated PDF Specification
PXC_ERR_INVALIDIMAGEFORMAT	0x82042712	Operation is not supported for this image format
PXC_ERR_IMAGE_CLOSED	0x82042713	Image already closed
PXC_ERR_METAPARSE	0x82042714	An error occured during metafile parsing
PXC_ERR_NOMETAFILE	0x82042715	Invalid metafile specified
PXC_ERR_INVOBJ	0x82042716	Invalid Object ID
PXC_ERR_NOPATH	0x82042717	Path is not defined
PXC_ERR_NOTEXT	0x82042718	Text must be drawn
PXC_ERR_TEXT	0x82042719	Not allowed when text drawing
PXC_ERR_NOSTATE	0x8204271a	Invalid save state level
PXC_ERR_INVALIDFONT	0x8204271b	Invalid font

PXC_ERR_INCOMPATIBLEFONT	`0x8204271c`	Font is not supported
PXC_ERR_INVALIDFONTNAME	`0x8204271d`	Invalid font Name
PXC_ERR_INVALIDFONTNUM	`0x8204271e`	Invalid font ID
PXC_ERR_ATM_INVALIDFONT	`0x8204271f`	Invalid ATM font
PXC_ERR_ATM_INVALIDFONTNAME	`0x82042720`	Invalid ATM font Name
PXC_ERR_INVALIDFONTDATA	`0x82042721`	Invalid font data
PXC_ERR_ALREADY_HAS_DC	`0x82042722`	HDC already allocated
PXC_ERR_HAS_NO_DC	`0x82042723`	HDC was not allocated
PXC_ERR_INVALID_PAGE	`0x82042724`	Invalid page index specified
PXC_ERR_ALREADY_SIGNED	`0x82042725`	Document already has a digital signature
PXC_ERR_NOT_AVAIL_IN_PDFA	`0x82042726`	Operation is not permitted in PDF/A mode

4.1.2 PXC_Err_FormatErrorCode

PXC_Err_FormatErrorCode

Top Previous Next

PXC_Err_FormatErrorCode fills information regarding a passed error code. See Error Handling for additional information

```
LONG  PXC_Err_FormatErrorCode(
    HRESULT errorcode,
    LPSTR buf,
    LONG maxlen
);
```

Parameters

errorcode

[in] *errorcode* specifies the error code obtained from one of the library functions.

buf

[out] *buf* specifies a pointer to a buffer where the error description will be filled.

Note: To determine the required buffer size you should pass `NULL` as *buf*.

maxlen

[in] *maxlen* specifies an available buffer size in characters (including a null-terminating character).

Return Values

If the function fails to recognize an error code the return value is negative.
If the function cannot find information about the error code the return value is zero.
If the function successfully found information and the parameter *buf* is `NULL` the return value is the number of characters required to store the description (including a null-terminating character).
If the function successfully found information and the parameter *buf* is not `NULL` the return value is the number of characters written to buffer (including a null-terminating character).

4.1.3 PXC_Err_FormatFacility

PXC_Err_FormatFacility (COPY)

Top Previous Next

PXC_Err_FormatFacility returns information where an error occurred using the available error code's. See Error Handling for additional information

```
LONG  PXC_Err_FormatFacility(
    HRESULT errorcode,
    LPSTR buf,
    LONG maxlen
);
```

Parameters

errorcode

[in] *errorcode* specifies the error code obtained from one of the functions.

buf

[out] *buf* specifies a pointer to a buffer where the error description will be filled.

Note: To determine the required buffer size you should pass `NULL` as *buf*.

maxlen

[in] *maxlen* specifies the available buffer size in characters (including null-terminating character).

Return Values

If the function fails to recognize an error code the return value is negative.
If the function cannot find information about an error code the return value is zero.
If the function successfully found information and the parameter *buf* is `NULL` the return value is the number of characters required to store the description (including null-terminating character).
If the function successfully found information and the parameter *buf* is not `NULL` the return value is the number of characters written to the buffer (including null-terminating character).

4.1.4 PXC_Err_FormatSeverity

PXC_Err_FormatSeverity

Top Previous Next

PXC_Err_FormatSeverity returns information regarding the error "weight". See Error Handling for additional information

```
LONG  PXC_Err_FormatSeverity(
    HRESULT errorcode,
    LPSTR buf,
    LONG maxlen
);
```

Parameters

errorcode

[in] *errorcode* specifies an error code obtained from one of the library functions.

buf

[out] *buf* specifies a pointer to a buffer where the error description will be filled.

Note: To determine the required buffer size you should pass `NULL` as *buf*.

maxlen

[in] *maxlen* specifies the available buffer size in characters (including a null-terminating character).

Return Values

If the function fails to recognize an error code the return value is negative.
If the function cannot find information about an error code the return value is zero.
If the function successfully finds information and the parameter *buf* is `NULL` the return value is the number of characters required to store the description (including a null-terminating character).
If the function successfully found information and the parameter *buf* is not `NULL` the return value is the number of characters written to the buffer (including a null-terminating character).

4.1.5 Example (C++)

Example (C++)

Top Previous Next

```
// Using of PXC_Err_FormatSeverity, PXC_Err_FormatFacility,
PXC_Err_FormatErrorCode functions
char* err_message = NULL;
char* buf = NULL;
_PXCPage* p = NULL;
    // Code below should always return an error and never work
HRESULT dummyError = PXC_AddPage(NULL, 0, 0, &p);
LONG sevLen = PXC_Err_FormatSeverity(dummyError, NULL, 0);
LONG facLen = PXC_Err_FormatFacility(dummyError, NULL, 0);
LONG descLen = PXC_Err_FormatErrorCode(dummyError, NULL, 0);
if ((sevLen > 0) && (facLen > 0) && (descLen > 0))
{
    // Total length of the formated text is the sum of the length for each
description
    // plus some additional characters for formating
    LONG total = sevLen + facLen + descLen + 128;
    // allocate buffer for message
    err_message = new char[total];
    err_message[0] = '\0';
    // allocate temporary buffer
    buf = new char[total];
    // get error severity and append to message
    if (PXC_Err_FormatSeverity(dummyError, buf, total) > 0)
        lstrcat(err_message, buf);
    lstrcat(err_message, " [");
    // get error facility and append to message
    if (PXC_Err_FormatFacility(dummyError, buf, total) > 0)
        lstrcat(err_message, buf);
    lstrcat(err_message, "]: ");
    // and error code description and append to message
    if (PXC_Err_FormatErrorCode(dummyError, buf, total) > 0)
        lstrcat(err_message, buf);
    ::MessageBox(NULL, err_message, "Test error", MB_OK);
    delete[] buf;
    delete[] err_message;
}
```

4.2 PRO Error Handling

PRO Error Handling

Top Previous Next

The PDF-XChange Tools PRO (XCPRO40.dll) has built in error handling functions:

Topic	Description
LIB Error Codes	Common errors returned by all Library (PXCp_...) functions.
PXC_Err_FormatErrorCode	Returns the Error Text for any `HRESULT` returned by PRO functions.
PXC_Err_FormatFacility	Returns the Facility for any `HRESULT` returned by PRO functions.
PXC_Err_FormatSeverity	Returns the Error Severity for any `HRESULT` returned by PRO functions.
Example (C++)	Example using the PXCp_Err_... functions

4.2.1 PRO Error Codes

PRO Error Codes

Top Previous Next

The most frequent error codes are listed in the table below; however, functions may return additional codes which are not listed here.There are three additional functions dealing with error's and may give additional information relating to error's: **PXCp_Err_FormatSeverity**, **PXCp_Err_FormatFacility**, **PXCp_Err_FormatErrorCode**. You may see a code example for use below the table. Please note that this function will provide information about **all** error codes which may be returned by the **PDF-XChange Pro Library** library.

Constant	Value	Description
DPro_ERR_NOTIMPL	0x820804b0	Not implemented
DPro_ERR_INVARG	0x82080001	Invalid argument
DPro_ERR_MEMALLOC	0xc20803e8	Insufficient memory
DPro_ERR_USER_BREAK	0xc20801f4	Operation was aborted by user

DPro_Err_DocNotRead	0x82082710	Doc is not read
DPro_Err_WrongPageNumber	0x82082711	Incorrect page number
DPro_Err_DocHasNoPages	0x82082712	Document contains no pages
DPro_Err_DocAlreadyRead	0x82082713	Document already read
DPro_Err_Internal	0x82082714	Internal error
DPro_Err_DocHasNoBookmarks	0x82082715	Document contains no bookmarks
DPro_Err_InvalidBMHandler	0x82082716	Invalid bookmark item handle
DPro_Err_BMItemNotPresent	0x82082717	Bookmark item is not present
DPro_Err_InvalidDestType	0x82082718	Invalid destination type
DPro_Err_InvalidDestiation	0x82082719	Invalid destination
DPro_Err_InvalidReadSeq	0x8208271a	Invalid read sequence
DPro_Err_InvalidImageObj	0x8208271b	Invalid image object
DPro_Err_InvalidImageNumber	0x8208271c	Invalid image number
DPro_Err_InvalidImageFormat	0x8208271d	Invalid image format
DPro_Err_WatermarkProcess	0x8208271e	Error whilst processing watermark(s)
DPro_Err_InvalidChunkIndex	0x8208271f	Invalid chunk index
DPro_Err_InvalidVarType	0x82082720	Invalid variable type
DPro_Err_InvalidIndex	0x82082721	Invalid index
DPro_Err_InvalidPageObj	0x82082722	Invalid page object
DPro_Err_InvalidContent	0x82082723	Invalid content
DPro_Err_LibMissImage	0x82082724	Required library (Image-Xchange) missing or located in wrong folder
DPro_Err_InvalidFont	0x82082725	Invalid font
DPro_Err_InvalidObj	0x82082726	Invalid object
DPro_Err_LibMissPdfXchange	0x82082727	Required library (PDF-Xchange) missing or located in wrong folder
DPro_Err_UnableToResolveDest	0x82082728	Unable to resolve destination
DPro_Err_InvalidOutlineItem	0x82082729	Invalid outline item
DPro_Err_InvalidThread	0x8208272a	Invalid thread
DPro_Err_ObjHasStream	0x8208272b	Object contains stream
DPro_Err_InvalidSaveOpts	0x8208272c	Invalid save image options
DPro_Err_AnnotationProcess	0x8208272d	Error whilst processing annotations

Possible values of warnings for the `Pro` library:

Constant	Value	Description
DPro_Wrn_InfoTagNotSet	0x42082773	Inforamtion tag is not set
DPro_Wrn_NeedGreaterBuffer	0x42082774	Larger buffer required
DPro_Wrn_ItemNotSetRetDefVal	0x42082775	Required item is not set. Return value is set to default
DPro_Wrn_DocNotEncrypted	0x42082776	Document not encrypted
DPro_Wrn_PageHasNoImages	0x42082777	Page contains no images
DPro_Wrn_PageHasNoThumbnail	0x42082778	Page has no thumbnail set
DPro_Wrn_InfoNotSet	0x42082779	Information not set
DPro_Wrn_BMMoveToItself	0x4208277a	Bookmark item may not be moved to

		the same position

Possible values of errors of `PDF parser/`structure:

Constant	Value	Description
PS_ERR_NOTIMPLEMENTED	0x820f04b0	Not implemented
PS_ERR_INVALID_ARG	0x820f0001	Invalid argument
PS_ERR_MEMALLOC	0x820f03e8	Insufficient memory
PS_ERR_USER_BREAK	0x820f01f4	Operation aborted by user
PS_ERR_INTERNAL	0x820f0011	Internal error
PS_ERR_INVALID_FILE_FORMAT	0x820f0002	Invalid file format
PS_ERR_UnsupportedDecodeFilter	0x820f2710	Unsupported decode filter
PS_ERR_UnsupportedPredictor	0x820f2711	Unsupported predictor
PS_ERR_InavidInBufferSize	0x820f2712	Invalid buffer size
PS_ERR_WriteInavidObject	0x820f0011	Invalid object whilst writing
PS_ERR_REQUIRED_PROP_NOT_SET	0x820f2716	Required property is not set
PS_ERR_INVALID_PROP_TYPE	0x820f2717	Invalid property type
PS_ERR_INVALID_PROP_VALUE	0x820f2718	Invalid property value
PS_ERR_INVALID_OBJECT_NUM	0x820f2719	Invalid object number
PS_ERR_INVALID_PS_OPERATOR	0x820f271c	Invalid `PS` operator
PS_ERR_NOT_ENOUGH_DATA_IN_STREAM	0x820f271d	Insufficient data in stream
PS_ERR_INVALID_COLORSPACE	0x820f271e	Invalid color space
PS_ERR_INVALIDACROFORM	0x820f271f	Invalid acro form
PS_ERR_INVALIDFORMFIELD	0x820f2720	Invalid form field
PS_ERR_INVALID_FONT_STRUCTURE	0x820f2773	Invalid font structure
PS_ERR_UNSUPPORTED_FONT_TYPE	0x820f2774	Unsupported font type
PS_ERR_UNKNOWN_OPERATOR	0x820f2787	Unknown operator
PS_ERR_INVALID_CONTENT_STATE	0x820f2788	Invalid content state
PS_ERR_INVALID_OP_ARGUMENTS	0x820f2788	Invalid operator argument(s)
PS_ERR_TokenParseError	0x820f27a5	Parse token error
PS_ERR_IONOTINITALIZED	0x820f27a7	Input/output system is not initialized
PS_ERR_NoPassword	0x820f27a8	No password
PS_ERR_UnknowCryptFlt	0x820f27a9	Unknown crypt filter
PS_ERR_WrongPassword	0x820f27aa	Wrong password
PS_ERR_InvlaidObjStruct	0x820f27ab	Invalid object structure
PS_ERR_WrongEncryptDict	0x820f27ac	Invalid encryption dictionary
PS_ERR_DocEncrypted	0x820f27ad	Document encrypted
PS_ERR_DocNOTEncrypted	0x820f27ae	Document not encrypted
PS_ERR_WrongObjStream	0x820f27af	Invalid object stream
PS_ERR_WrongTrailer	0x820f27b0	Invalid document trailer
PS_ERR_WrongXRef	0x820f27b1	Invalid xref table
PS_ERR_WrongDecodeParms	0x820f27b2	Invalid decode parameter(s)
PS_ERR_XRefNotFounded	0x820f27b3	xref table is not foud
PS_ERR_DocAlreadyRead	0x820f27b4	Document is already read

PS_ERR_DocNotRead	0x820f27b5	Document is not read
PS_ERR_WrongObjNum	0x820f27b6	Invlaid object number
PS_ERR_InvalidFilterIndex	0x820f27b7	Invalid filter index
PS_ERR_InvalidStreamOffset	0x820f27b8	Invalid stream offset

Possible values of warnings of `PDF parser/structure`:

Constant	Value	Description
PS_WRN_INVALID_PROP_VALUE	0x420f271b	Invalid property value
PS_WRN_PROP_NOT_SET	0x420f271a	Property not set
PS_Wrn_TokenParseEOD	0x420f27a6	End of data reached whilst parsing token

4.2.2 PXCp_Err_FormatErrorCode

PXCp_Err_FormatErrorCode

Top Previous Next

PXCp_Err_FormatErrorCode fills information regarding a passed error code. See Error Handling for additional information

```
LONG  PXCp_Err_FormatErrorCode(
    HRESULT errorcode,
    LPSTR buf,
    LONG maxlen
);
```

Parameters

errorcode

[in] *errorcode* specifies the error code obtained from one of the library functions.

buf

[out] *buf* specifies a pointer to a buffer where the error description will be filled.

Note: To determine the required buffer size you should pass `NULL` as *buf*.

maxlen

[in] *maxlen* specifies an available buffer size in characters (including a null-terminating character).

Return Values

If the function fails to recognize an error code the return value is negative.
If the function cannot find information about the error code the return value is zero.
If the function successfully found information and the parameter *buf* is `NULL` the return value is the number of characters required to store the description (including a null-terminating character).
If the function successfully found information and the parameter *buf* is not `NULL` the return value is the number of characters written to buffer (including a null-terminating character).

4.2.3 PXCp_Err_FormatFacility

PXCp_Err_FormatFacility

Top Previous Next

PXCp_Err_FormatFacility returns information where an error occurred using the available error code's. See Error Handling for additional information

```
LONG  PXCp_Err_FormatFacility(
    HRESULT errorcode,
    LPSTR buf,
    LONG maxlen
);
```

Parameters

errorcode

[in] *errorcode* specifies the error code obtained from one of the functions.

buf

[out] *buf* specifies a pointer to a buffer where the error description will be filled.

Note: To determine the required buffer size you should pass `NULL` as *buf*.

maxlen

[in] *maxlen* specifies the available buffer size in characters (including null-terminating character).

Return Values

If the function fails to recognize an error code the return value is negative.
If the function cannot find information about an error code the return value is zero.
If the function successfully found information and the parameter *buf* is `NULL` the return value is the number of characters required to store the description (including null-terminating character).
If the function successfully found information and the parameter *buf* is not `NULL` the return value is the number of characters written to the buffer (including null-terminating character).

4.2.4 PXCp_Err_FormatSeverity

PXCp_Err_FormatSeverity

Top Previous Next

PXCp_Err_FormatSeverity returns information regarding the error "weight". See Error Handling for additional information

```
LONG  PXCp_Err_FormatSeverity(
    HRESULT errorcode,
    LPSTR buf,
    LONG maxlen
);
```

Parameters

errorcode

[in] *errorcode* specifies an error code obtained from one of the library functions.

buf

[out] *buf* specifies a pointer to a buffer where the error description will be filled.

Note: To determine the required buffer size you should pass `NULL` as *buf*.

maxlen

[in] *maxlen* specifies the available buffer size in characters (including a null-terminating character).

Return Values

If the function fails to recognize an error code the return value is negative.
If the function cannot find information about an error code the return value is zero.
If the function successfully finds information and the parameter *buf* is `NULL` the return value is the number of characters required to store the description (including a null-terminating character).
If the function successfully found information and the parameter *buf* is not `NULL` the return value is the number of characters written to the buffer (including a null-terminating character).

4.2.5 Example (C++)

Example (C++)

Top Previous Next

Example (C++).

```
/ Using of PXCp_Err_FormatSeverity, PXCp_Err_FormatFacility,
PXCp_Err_FormatErrorCode functions

char* err_message = NULL;
char* buf = NULL;
PDFDocument p = NULL;
    // Code below should always return error and never work
HRESULT dummyError = PXCp_ReadDocumentW(p, NULL, 0);
LONG sevLen = PXCp_Err_FormatSeverity(dummyError, NULL, 0);
LONG facLen = PXCp_Err_FormatFacility(dummyError, NULL, 0);
LONG descLen = PXCp_Err_FormatErrorCode(dummyError, NULL, 0);
if ((sevLen > 0) && (facLen > 0) && (descLen > 0))
{
    // Total length of formated text is sum of length for each description
    // plus some characters for formating
    LONG total = sevLen + facLen + descLen + 128;
    // allocate buffer for message
    err_message = new char[total];
    err_message[0] = '\0';
    // allocate temporary buffer
    buf = new char[total];
    // get error severity and append to message
    if (PXCp_Err_FormatSeverity(dummyError, buf, total) > 0)
        lstrcat(err_message, buf);
    lstrcat(err_message, " [");
    // get error facility and append to message
    if (PXCp_Err_FormatFacility(dummyError, buf, total) > 0)
        lstrcat(err_message, buf);
    lstrcat(err_message, "]: ");
    // and error code description and append to message
    if (PXCp_Err_FormatErrorCode(dummyError, buf, total) > 0)
        lstrcat(err_message, buf);
    ::MessageBox(NULL, err_message, "Test error", MB_OK);
    delete[] buf;
    delete[] err_message;
}
```

4.3 DSErrorLookUp Utility

DSErrorLookUp Utility

Top Previous Next

There is an additional utility included with this library which provides valuable additional data regarding all known error codes - **DSErrorLookUp.exe**. This can be found in your PDF-XChange/Tools installation folders and is extremly useful during your application development process - we strongly reccomend ALL developers utilise **DSErrorLookUp.exe** during the debugging of their applications and prior to support requests relating to Error Code return values and their meaning.

5 Tracker Software Products

Tracker Software Products

Top Previous Next

Who are we and what do we do?

We at Tracker Software Products take great pride in the software products we create and distribute. We sell our products directly, via Distributors, Resellers and OEM partners - in some cases with our products and larger partners these products are sold under different labels and names than those we sell directly, this is to allow our partners to build a following for their own brand and protect their future, our printer drivers however, whilst allowing you to rename them in the user's Printers list - do not allow 100% rebranding.

No matter how our products reach you, we want you to experience the best results possible - please do contact us if for any reason you are dissatisfied or have a suggestion how we can improve our product offerings.

You may also be interested in related products available from us - in the following brief topics you will find details of how to contact us, request support and summary details of the products available from us for both 'End Users' and Software Development tools for other Software developers to utilise in their product offerings.

Please do contact us if you cannot find the information you require within this manual/help file.

5.1 Our Products

Our Products

Top Previous Next

Products Offered By Tracker Software Products Updates Can be downloaded from our Update's page at our Web site

End User/Retail Products

You can Purchase Direct from our web site and be using any of our products the same day!

- PDF-XChange - Create fully native Adobe compatible PDF Files from virtually any Windows 32 Bit software application
- PDF-Tools - Create and manipulate Adobe PDF Files and batch Convert Images to PDF Files and more...
- Raster-XChange - Create and manipulate Image files from any Windows document or application.
- TIFF-XChange Create and manipulate TIFF files from any Windows document or application.
- PDF-XChange Viewer - Free, fully featured ADVANCED PDF Viewing application allowing users to read and add/modify PDF page and file content, ideal replacement for Adobe's Free Reader.
- PDF-XChange Viewer PRO - Extend the functionality currently available in the Free PDF-XChange Viewer.

Software Developers SDK's and other Products

You can Purchase Developer SDK's from our web site and be using any of our products the same day!

- PDF-XChange - Create fully native Adobe compatible PDF Files from your application output.
- PDF-Tools - Create and manipulate Adobe PDF Files and batch Convert Images to PDF Files and more...
- PDF-XChange Viewer SDK - Embed PDF viewing directly within your software applications.
- Image-XChange SDK - Print, Convert, Scan and View Imaging formats!
- Raster-XChange SDK - Create and manipulate Image files from any Windows document or application.
- TIFF-XChange SDK - Create and manipulate TIFF files from any Windows document or application.
- **OCR-XChange - coming Summer 2008**

Trial Versions

All of our products are available as fully functional evaluation downloads for you to try before you buy - usually printing a demo watermark/stamp to differentiate between output created with the evaluation or licensed versions. We recommend that all users test the product they wish to buy first - thus ensuring you only buy when you are satisfied that the product meets your needs.

Trial versions are available from our web site:

For more details visit our web site or contact us by email.

5.2 Contact Us

Contact Us

Top Previous

How to Contact Us ...

-

Head Office

-

Tracker Software Products Ltd.

Units 1-3 Burleigh Oaks

East Street Turners Hill,

RH10 4PZ Sussex England.

Tel: +44 020 8123 4934 Sales/Administration
(pls do not use for support issue's)

Fax:+44(0)1342-718060

North America

Tracker Software Products
18326-D Minnetonka Blvd,
Wayzata,MN 55391
USA.

Tel: +001 (952) 232-0414
(pls do not use for support issue's)
++++++++++++++++

Our Web Site : http://www.docu-track.com

We also have offices and representatives in several other locations including : United States, France, Germany and Ukraine - in some instances after an initial contact with our UK office you may be referred to one of these locations if appropriate.

To contact us for support related issues:

Please see this FAQ page before contacting our support department - it may save you the task !

We recommend you use our Web Based User Support Forums and scan the existing library of questions and answers, if you don't find a suitable response then feel free to post your own - all questions receive an answer within 1 business day at worst!

Due to excessive spam/junk mail - our support email address is no longer atcive - please see below for other valid contact info

If for any reason you have difficulty linking to the forum or feel it is inappropriate for your needs then please email sales@docu-track.com , we regret we cannot answer support requests via telephone without a valid support contract. The number above is answered by administration staff who are not trained to assist with technical problems.

To Contact us for Sales/Administration related issues:

sales@docu-track.com End User, Developer and OEM.

admin@docu-track.com

upgrades@docu-track.com All Licensing info requests (including lost license info)

All this information and a good deal more is available via our web site and the links provided.

-

Magazine reviews and press requests.

We are keen to assist in any way possible - please contact our sales department for any information or help you may require.

Index

- C -

Contact 578
Contact Us 578

- P -

PXC_AddCheckBox 245
PXC_AddPage 70
PXC_GetImageColors 565
PXC_NewDocument 169

- T -

Tracker Software Products 578