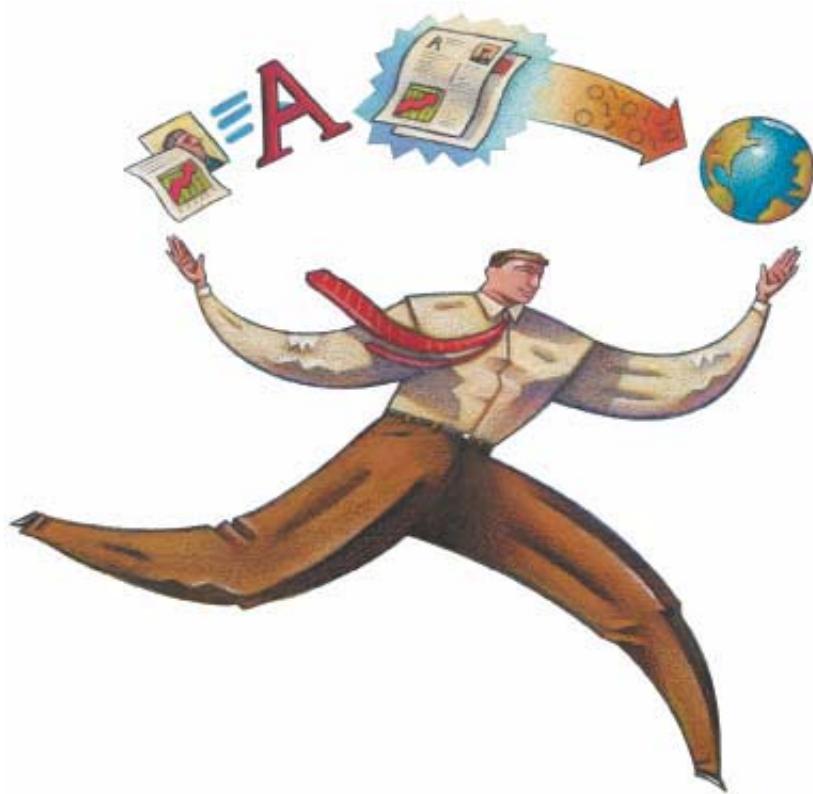




Acrobat Core API Overview

Technical Note #5190

Version : Acrobat 5.0



ADOBESYSTEMS INCORPORATED

Corporate Headquarters

345 Park Avenue
San Jose, CA 95110-2704
(408) 536-6000
<http://partners.adobe.com>

June 25, 2001

Copyright 2001 Adobe Systems Incorporated. All rights reserved.

NOTICE: All information contained herein is the property of Adobe Systems Incorporated. No part of this publication (whether in hardcopy or electronic form) may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the Adobe Systems Incorporated.

PostScript is a registered trademark of Adobe Systems Incorporated. All instances of the name PostScript in the text are references to the PostScript language as defined by Adobe Systems Incorporated unless otherwise stated. The name PostScript also is used as a product trademark for Adobe Systems' implementation of the PostScript language interpreter.

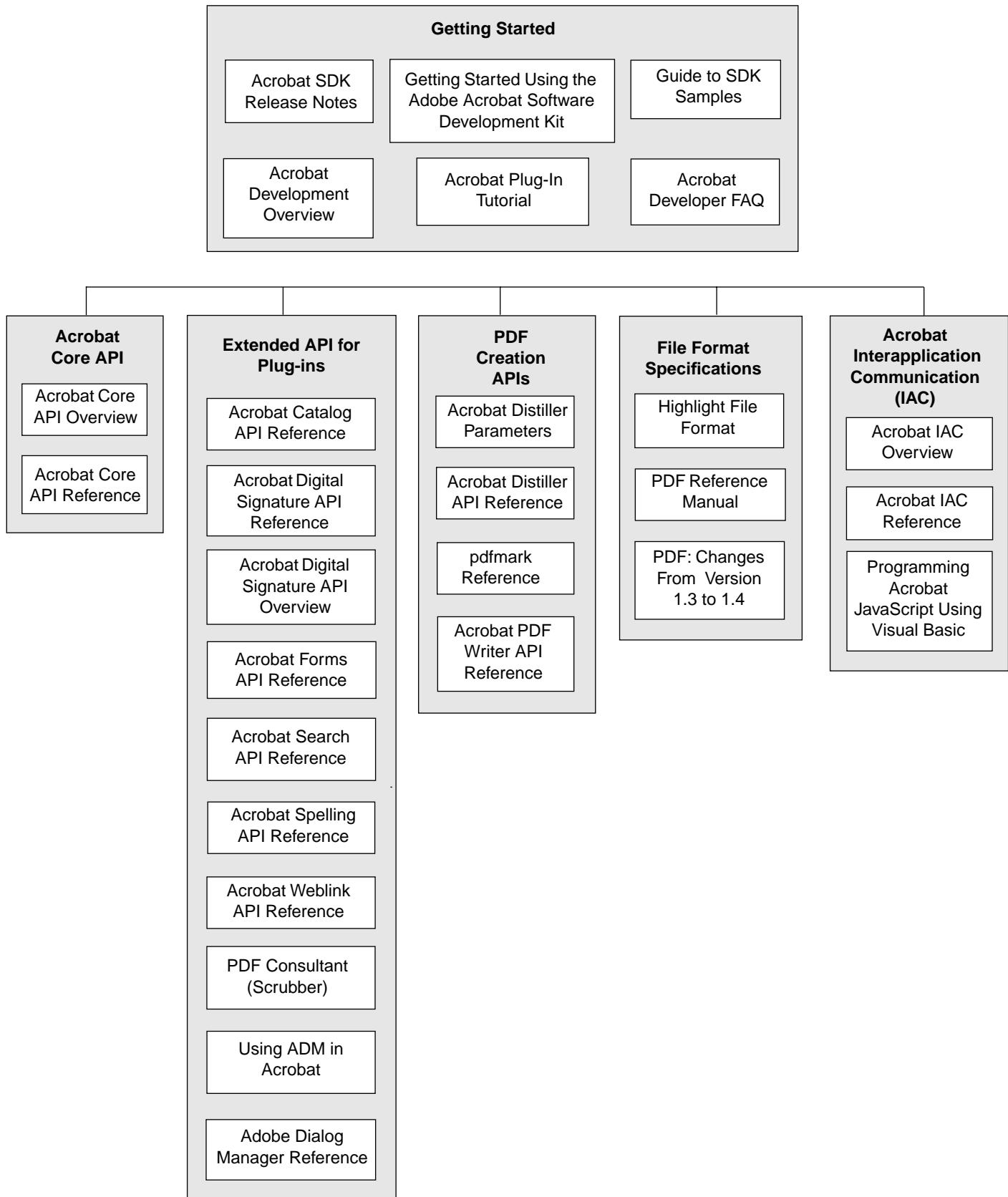
Except as otherwise stated, any reference to a "PostScript printing device," "PostScript display device," or similar item refers to a printing device, display device or item (respectively) that contains PostScript technology created or licensed by Adobe Systems Incorporated and not to devices or items that purport to be merely compatible with the PostScript language.

Adobe, the Adobe logo, Acrobat, the Acrobat logo, Acrobat Capture, Acrobat Catalog, Acrobat Reader, Acrobat Search, Distiller, PostScript, and the PostScript logo are trademarks of Adobe Systems Incorporated.

Apple, Macintosh, and Power Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries. PowerPC is a registered trademark of IBM Corporation in the United States. ActiveX, Microsoft, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and other countries. UNIX is a registered trademark of The Open Group. All other trademarks are the property of their respective owners.

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

Acrobat SDK Documentation Roadmap



Contents

Preface	15
Introduction	15
Audience	15
Assumptions	15
How This Document Is Organized	16
Related Documentation	17
Conventions Used In This Document	18
Chapter 1 Core API Overview	21
Ways to Integrate With the Acrobat Viewers	21
Acrobat Core API	22
Core API Objects	23
Core API Methods	24
Data Types	26
Scalar Types	26
Simple Types	27
Complex Types	27
Opaque Types	28
Cos Objects	28
Understanding Coordinate Systems	28
User Space	28
Device Space	29
Translating between User Space and Device Space	30
Machine Port Space	31
Using Rectangles and Quadrilaterals	32
Handling Exceptions	32
Adding New Object Types	33
Storing Private Data in PDF Files	33
Chapter 2 Core API Mechanics	35
Host Function Tables	35
Using HFTs	36

HFT Servers	36
Creating a New HFT	37
Replacing Built-In Methods	37
Interaction Between Plug-ins and the Acrobat Viewer	38
Locating Plug-ins	38
Handshaking and Initialization	39
Exporting HFTs	40
Importing HFTs and Registering for Notifications	40
Initialization	41
Unloading	42
Callbacks	42
Notifications	44
Enumeration	45
Handling Events	45
Mouse Clicks	45
Adjust Cursor	45
Key Presses	46
Adding Message Handling	46
Plug-in Prefixes	46
Acrobat and Reader Differences	46
Changing the Acrobat Viewer User Interface	47
Adding or Removing Menus and Menu Items	47
Modifying the Toolbar	48
Controlling the “About” Box and Splash Screen	48
Placing Plug-in Help Files In a Standard Location	48
Page View Layers	48
Reducing Conflicts Among Plug-ins	49
Chapter 3 Plug-in Applications	51
Controlling the Acrobat Viewers	51
Drawing Into Another Window	51
Indexed Searching	52
Steps in the Acrobat Product’s Indexed Searching	52
Extracting Text	53
Providing Document Security	54
Modifying File Access	54
Creating New Annotation Types	54
Accessing the Info Dictionary	54

Adding Private Data To PDF Files	55
Chapter 4 Acrobat Support	57
ASAtom	57
ASCab	57
ASCab Method Naming	58
Handling Pointers	58
ASCab Methods	58
ASCallback	59
ASExtension	59
ASFile	60
ASFileSys	60
ASPathName	62
ASStm	62
ASText	62
Configuration	64
Errors	64
Fixed-point Math	65
Fixed-point Utility Macros	65
Fixed-point Mathematics Methods	66
Fixed-point Matrix Methods	66
HFT Methods	66
Memory Allocation	67
Platform-specific Utilities	67
Macintosh	67
UNIX	67
Windows	68
Chapter 5 Acrobat Viewer Layer	69
General	70
AVActionHandler	70
AVAlert	71
AVAnnotHandler	71
AVApp	71
AVCommand	72
Invoking AVCommands Programmatically	72
AVCommand Methods	74

AVConversion	75
AVCrypt	75
AVDoc	76
AVGrafSelect	76
AVMenu	76
AVMenubar	77
AVMenuItem	78
AVPageView	79
AVSweetPea	79
AVSys	80
AVTool	80
AVToolBar	80
AVToolButton	81
AVWindow	82
Chapter 6 Portable Document Layer	85
General PD Layer Methods	86
Metadata	86
New Metadata Features in PDF 1.4	86
Metadata APIs in Acrobat 5.0	87
PDAction	88
PDAnnot	88
PDBead	89
PDBookmark	89
PDCharProc	90
PDDoc	91
Querying PDDoc Permissions	91
PDDoc Methods	92
PDFFileSpec	93
PDFFont	94
PDForm	96
PDGraphic	97
PDImage	97
PDInlineImage	98
PDLINKAnnot	98
PDNameTree	99

PDNumTree	99
PDPage	99
PDPageLabel	100
PDPath	101
PDStyle	101
PDText	101
PDTextAnnot	102
PDTextSelect	102
PDThread	104
PDThumb	104
PDTrans	104
PDViewDestination	104
PDWord	105
PDWordFinder	106
PDXObject	107

Chapter 7 PDFEdit—Creating and Editing Page Content 109

Introduction	109
Overview of PDFEdit	109
Why PDFEdit?	109
What is PDFEdit?	110
PDFEdit Paradigm	110
PDFEdit Classes	111
Basic Classes	111
PDEElement Classes	113
PDEElement Attribute Classes	113
Example	114
Comparing PDFEdit to Other Core API Methods	115
Classes	115
Mapping Between PDF Operators and PDFEdit	115
Page Contents Stream and PDFEdit Object List Correspondence	115
Enumerating Page Objects	116
Using PDFEdit versus PDWordFinder	117
Using PDFEdit Versus PDPageAddCosContents	117
Hit Testing	117
Using PDFEdit Methods	118
Reference Counting	118

Matrix Operations	119
Clip Objects and Sharing	119
Marked Content	119
Cos Objects and Documents	120
XObjects and PDEObjects.	120
Resources	120
Client Identifiers	121
Guide to Page Creation	121
Common Code Sequence	121
Ways To Modify a Page's Content	122
Debugging Tools and Techniques	126
Object Dump.	126
PDFEdit Methods	128
Dump Methods.	128
General Methods	129
PDEClip	129
PDEColorSpace	129
PDEContainer	130
PDEContent	131
PDEDeviceNColors	132
PDEElement	132
PDEExtGState	133
Setting the Opacity of an Object.	133
PDEExtGState Methods	133
PDEFont	134
PDEForm.	135
PDEGroup	136
PDEImage	136
PDEObject.	137
PDEPath	137
PDEPattern	137
PDEPlace	138
PDEPS.	138
PDEShading	139
PDESoftMask	139
PDEText	139
PDEUnknown	140

PDEXGroup	140
PDEXObject	141
PDSysEncoding	141
PDSysFont	141

Chapter 8 PDSEdit—Creating and Editing Logical Structure. 143

Introduction.	143
Why Have Logical Structure?	143
Logical Structure in a PDF Document	144
The Structure Tree	145
Navigating a PDF Document	145
Extracting Data From a PDF Document	145
Adding Structure Data To a PDF Document	145
Using pdfmark to Add Structure Data to PDF	146
PDSEdit Classes.	146
PDSTreeRoot	146
PDSElement	146
PDSAttrObj	146
PDSMC	147
PDSOBJR	147
PDSClassMap	147
PDSRoleMap	147
Relationship of PDSEdit and PDFEdit	148
Using the PDSEdit API: Examining Structure	148
Structure Tree Root	148
Structure Elements	148
Traversing Elements in a Subtree	149
Object Attributes	150
Other Object Characteristics.	151
Element Types and the Role Map	151
Classes and the Class Map	151
Using the PDSEdit API: Creating Structure	152
Structure Tree Root	152
Structure Elements	152
Adding Marked Content to an Element	153
Adding an Object Reference to an Element	153
Class Map	154
Role Map.	154

Chapter 9 Cos Layer	155
Cos Objects: Direct and Indirect	155
File structure	156
Cos Objects in the Core API	156
CosDoc.	157
CosObj	157
CosArray	157
CosBoolean	158
CosDict.	158
CosFixed	158
CosInteger	159
CosName	159
CosNull.	159
CosStream	159
CosString.	160
Encryption/Decryption	160
Chapter 10 Handlers	161
Action Handlers	162
Annotation Handlers	162
AVCommand Handlers.	163
Creating an AVCommand Handler.	163
Exposing AVCommands to the Batch Framework	164
File Format Conversion Handlers	166
File Specification Handlers.	166
Security Handlers	167
Selection Servers	167
Tools	168
Window Handlers	168
File Systems	169
Progress Monitors	170
Transition Handlers.	171
Chapter 11 Document Security	173
Encryption and Decryption.	173

Security Handlers	174
Adding a Security Handler.	175
Security Handler Callbacks	176
New Security Features in Acrobat 5.0	176
Opening a File	177
Acrobat's Built-in Authorization Procedure.	178
Saving a File	179
Setting a Document's Security.	180
Implementation Examples	180
Saving a File With a New Encryption Dictionary.	180
Opening an Encrypted File	181
Utility Methods	181
Chapter 12 Handling Errors	183
Exception Handlers	183
Handling an Exception Later	186
Returning From an Exception Handler.	186
API Methods That Raise Exceptions.	188
Exception Handler Caveats	188
Don't Use goto In a DURING...HANDLER Block	188
Don't Nest Exception Handlers In a Single Function	189
Be Careful About Register Usage	190
Chapter 13 Changes For This Revision	191
New Features in Acrobat 5.0.	191
New Core API Objects	191
Other Changes in This Document	192
Appendix A Object Interrelationships	193
Appendix B Portable Document Format	195
Relationship of Acrobat and PDF Versions	195
Introduction To PDF	195
PDF Objects	196
File Structure.	196
Document Structure	198
Page Contents	199

Index.	201
-----------------------	------------

Preface

Introduction

This document provides a conceptual overview of the Acrobat core application programming interface (API). It is intended to familiarize you with the core API, as described in detail in the [Acrobat Core API Reference](#), and the Acrobat conventions for using this interface.

The core API is used primarily by Acrobat *plug-ins*. Plug-ins can be created for the viewers: Acrobat® and Acrobat Reader®.

NOTE: Users of the Adobe PDF Library will also find much of the information in this document helpful. The information that does *not* apply to the PDF library are primarily the sections on plug-in mechanics and on the AcroView (AV) layer of the API.

Using the API, a plug-in can perform functions such as:

- Controlling an Acrobat session
- Customizing the Acrobat user interface
- Augmenting existing Acrobat functions
- Displaying Portable Document Format (PDF) documents in an application-supplied window, without using the Acrobat viewer user interface
- Manipulating the contents of a PDF file
- Adding private data to PDF files

NOTE: See [Chapter 13, “Changes For This Revision,”](#) for references to features that are new to Acrobat 5.0.

Audience

The primary audience of this document is Acrobat and Acrobat Reader plug-in developers. Developers of PDF Library applications and interapplication communication (IAC) applications will also find much of the information helpful.

Assumptions

This document assumes that you are familiar the Acrobat product family and that you are an experienced user of Acrobat products. You should understand ANSI-C or C++

and be familiar with programming on your development platform. If you plan to manipulate data in PDF files, you should be familiar with the contents and structure of PDF files, as described in the [PDF Reference](#). For an overview of PDF structures, see [Appendix B](#) in this document.

If you are new to writing plug-ins, work through some sample plug-ins in the [Acrobat Plug-In Tutorial](#). Then you can explore other sample plug-ins provided with the SDK.

How This Document Is Organized

This document is organized as follows:

- [Chapter 1, “Core API Overview,”](#) describes the structure of the core API, its objects, methods, and data types.
- [Chapter 2, “Core API Mechanics,”](#) discusses a number of topics basic to plug-in development, including the host function table mechanism, the plug-in handshake sequence, and creating callbacks.
- [Chapter 3, “Plug-in Applications,”](#) describes some of the things that plug-ins can do.
- [Chapter 4, “Acrobat Support,”](#) describes the methods for manipulating objects in the Acrobat support (AS) layer, as well as platform-specific methods.
- [Chapter 5, “Acrobat Viewer Layer,”](#) describes the Acrobat Viewer (AV) layer methods for controlling the Acrobat viewer application and modifying its user interface.
- [Chapter 6, “Portable Document Layer,”](#) describes the portable document (PD) layer of object methods that enable plug-ins to access and manipulate most data in a PDF file.
- [Chapter 7, “PDFEdit—Creating and Editing Page Content,”](#) describes PDFEdit, a collection of objects that enable your plug-in to treat a page’s contents as a list of objects rather manipulating content stream marking operators.
- [Chapter 8, “PDSEdit—Creating and Editing Logical Structure,”](#) describes PDSEdit, a collection of objects that enable your plug-in to create and examine the logical structure in PDF files.
- [Chapter 9, “Cos Layer,”](#) describes the Cos object methods, which provide access to the low-level object types and file structure in PDF files.
- [Chapter 10, “Handlers,”](#) describes handlers, a collection of methods that expand the number of object types Acrobat supports by adding new types of tools, annotations, actions, file systems, and so on.
- [Chapter 11, “Document Security,”](#) describes the core API document security features: security handlers, encryption and decryption methods, and utility methods.

- [Chapter 12, “Handling Errors,”](#) covers Acrobat’s error system, providing advice on how to write exception handlers.
- [Chapter 13, “Changes For This Revision,”](#) lists the additions and modifications to this document for the Acrobat SDK, Revision 5.0.
- [Appendix A](#) illustrates object interrelationships.
- [Appendix B](#) provides an overview of PDF structures.

Related Documentation

For more information, see the following SDK documents, which are referenced in this overview:

- [Getting Started Using the Adobe Acrobat Software Development Kit](#) provides an overview of the Acrobat SDK and the supporting documentation.
- [Acrobat Core API Reference](#) contains the method prototypes and details on arguments. By using this reference online, you can copy prototypes directly into your plug-in as you are coding it.
- [Acrobat Plug-In Tutorial](#) explains how to use the Acrobat core API to write plug-ins for Acrobat and Acrobat Reader. It describes basic Acrobat development concepts, provides an overview of how Acrobat interacts with plug-ins at load-time and initialization, and includes chapters that explain and show by example how to code various tasks that your plug-in can perform to manipulate and enhance the Acrobat viewer user interface as well as manipulate the contents of underlying PDF documents.
- [Acrobat Development Overview](#) provides guidelines for developing plug-ins, including registering plug-in names and development environment requirements.
- [Acrobat Forms API Reference](#) describes the Acrobat Forms plug-in API methods.
- [PDF Reference, second edition, version 1.3](#) describes PDF version 1.3 format in detail, including PDF object types, file format, and document structure.
- [PDF: Changes From Version 1.3 to 1.4](#) supplements the *PDF Reference, second edition, version 1.3*, by providing PDF 1.4 format details.
- [Using ADM in Acrobat](#) describes how to create platform-independent dialogs for your plug-in.

NOTE: In this document, references to documents that appear online (in blue, italics) are live links. However, to activate these links, you must install the documents on your local file system in the same directory structure in which they appear in the Acrobat SDK. This happens automatically when you install the SDK.

If you did not install the entire SDK and you do not have all the documents, please visit the [Adobe Solutions Network Web site](#) to find the documents you need. Then install them in the appropriate directories. You can use the Acrobat

SDK Documentation Roadmap located at the beginning of this document as a guide.

Additional documents that you should have available for reference are listed below. These documents are available on the Adobe Solutions Network Web site:

- *PostScript Language Reference, third edition* describes the syntax and semantics of the PostScript® language and the Adobe imaging model.

Conventions Used In This Document

Item	Character Definition	Use and Examples
File names	Courier, 12-point	C:\templates\Acrobat_docs
Code items within plain text; parameter names in reference documents	Courier, 12-point, bold	The GetExtensionID method returns an ASAtom object
Code examples set off from plain text	Courier, 10-point, plain	These are variable declarations: AVMenu commandMenu, helpMenu;
Code values within plain text	Helvetica, 11-point, plain; (same as plain text)	The method returns true or false A null pointer
Pseudocode	Helvetica, 11-point, italic	ACCB1 void ACCB2 ExeProc(void) { do something }
Cross references to Web pages	Blue text; everything else “as-is”	The Acrobat Solutions Network URL is: http://partners.adobe.com/asn
Cross references to titles of other Acrobat SDK documents	Blue text; Helvetica, 11-point, italic	See the <i>Acrobat Core API Overview</i> .
Cross references within a document	Blue text; everything else “as-is”	See Section 3.1, “Using the SDK.” Test whether an ASAtom exists.
PostScript language operators, PDF operators, keywords, dictionary key names; user interface names	Helvetica, 11-point, bold	The setpagedevice operator The File menu

Item	Character Definition	Use and Examples
Document titles that are not cross-reference links to other Acrobat SDK documents, new terms, PostScript variables	Helvetica, 11-point, italic	<i>Acrobat Core API Overview</i> <code>filename deletefile</code>

Preface

Conventions Used In This Document

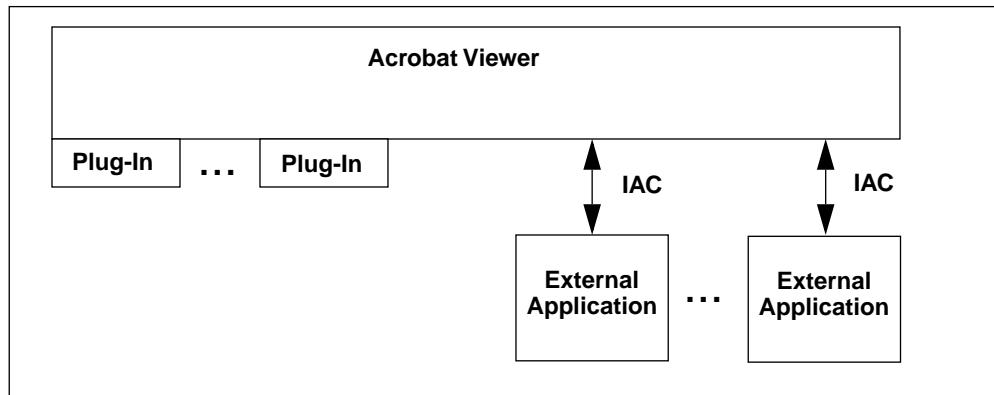
The Acrobat core API is a set of interfaces you can use to write plug-ins that integrate with Acrobat and Acrobat Reader. This chapter introduces the core API, describing its object orientation and organization, and a number of other concepts fundamental to understanding the API.

Ways to Integrate With the Acrobat Viewers

You can develop software that integrates with Acrobat and Acrobat Reader in two ways:

- By creating plug-ins that are dynamically linked to the Acrobat viewer and extend the viewer's functionality
- By writing a separate application process that uses interapplication communication (IAC) to control Acrobat functionality. DDE and OLE are supported on Windows and Apple events / AppleScript on the Macintosh .

FIGURE 1.1 *Ways to integrate with Acrobat and Acrobat Reader*



Through IAC interfaces, an application can control the viewer in ways the interactive user can. A plug-in can control the viewer in the same way, but, in addition it can extend the viewer using the much broader range of core API methods.

Your project's scope determines which of these methods better meets your needs. You can also use a combination approach, by creating a plug-in and a separate application, where the application sends messages to the plug-in, and the plug-in manipulates the Acrobat viewer.

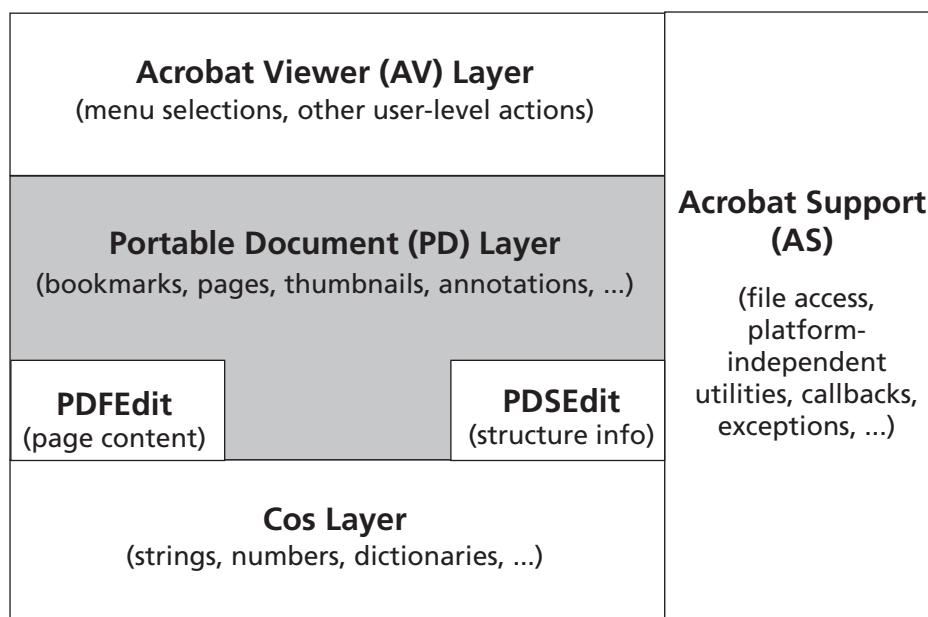
To learn more about using IAC, refer to the Acrobat SDK documents [Acrobat IAC Overview](#) and [Acrobat IAC Reference](#)

Acrobat Core API

The core API consists of a set of *methods* that operate on *objects*. The objects have types and encapsulate their data. This object orientation is a conceptual model, implemented using a standard ANSI C-programming interface. Methods are C functions; objects are opaque data types. The Core API is supported on Microsoft 32-bit Windows®, Apple Macintosh, and UNIX® platforms.

The core API methods are organized into the hierarchy shown in [Figure 1.2](#).

FIGURE 1.2 Overview of Core API



Acrobat Viewer

The Acrobat Viewer (AV) layer (also known as AcroView or AV Model) deals with the Acrobat viewer. Its methods allow plug-ins to manipulate components of the Acrobat viewer application itself, such as menus and menu items.

NOTE: The AV layer is not available to users of the PDF Library.

Portable Document

The Portable Document (PD) layer (also known as PDModel) provides access to components of PDF documents. Its methods allow plug-ins to manipulate document components such as document pages and annotations. Closely related to the PD layer are two method groups, each of which controls a different aspect of a PDF document:

- PDFEdit methods deal with physical representation of a PDF document. More specifically, PDFEdit methods treat page content as a list of objects whose values and attributes a plug-in can modify. The methods allow your plug-in to read, write,

edit, and create page contents and page resources, which may contain fonts, images, and so on.

- PDSEdit methods deal with the logical structure of a PDF document. A PDF document's logical structure is built independently of its physical representation, with pointers from the logical structure to the physical representation, and vice versa. PDSEdit methods store the logical structure information. They allow your plug-in to access PDF files by means of a structure tree. Having logical structure in PDF files facilitates navigating, searching, and extracting data from PDF documents. For example, PDSEdit methods can obtain logically-ordered content, independently of drawing order.

Acrobat Support

The Acrobat Support (AS) layer provides platform-independent utility functions and allows plug-ins to override the built-in file-handling mechanisms.

Cos Layer

The Cos Object System (Cos) layer provides access to the building blocks used to construct documents. Cos methods allow plug-ins to manipulate low-level data in a PDF file, such as dictionary and string objects.

Platform-Specific Methods

In addition to the method groups represented in [Figure 1.2](#), the core API includes platform-specific plug-in utilities to handle issues that are unique to Macintosh, Windows and UNIX platforms.

Core API Objects

Most objects accessible by AV and PD layer methods are opaque. They are, in general, neither pointers nor pointers to pointers. They provide equivalent functionality in that they merely reference an object's data rather than containing it. They cannot reliably be syntactically considered a `void *`. If you assign one object to another variable, both variables affect the *same* internal object.

Typically objects are named using the following conventions. There are exceptions. For example, not all pointers to structures end in `P`. However, your familiarity with the conventions described here should help you recognize the types when you encounter them in the Acrobat SDK documentation.

- The name of the concrete definition for a complex type ends in `Rec`, for record.
- Typically, a pointer to simple or complex type ends in `P`, for pointer.
- Opaque types do not contain a `P` suffix. For example, a `PDDoc` object references a PDF document.
- Three names identify complex types that provide callback methods:

- **Monitor**: A set of callbacks for an enumeration method (also used for **ProgressMonitor**)
- **Server**: An implementation for a service added by a plug-in
- **Handler**: An implementation for a subtype of object handled by a plug-in

Callback method names typically contain the suffix **Proc**, for procedure.

Core API Methods

There are several types of methods in the core API. See the [Acrobat Core API Reference](#) for complete information on all methods.

Method names generally are of the form:

`<layer><object><verb><thing>`,

where

- *layer* identifies the method's layer (**AV** for Acrobat Viewer, **PD** for Portable Document, **Cos** for Cos, and **AS** for Acrobat Support)
- *object* identifies the object upon which the method acts (for example, menu, window, font, bookmark, annotation, dictionary, string, or file)
- *verb* specifies an action such as **Get**, **Set**, **Acquire**, **Release**, **Create**, **New**, and **Destroy**. See [Table 1.1](#) for a list of the most common verbs in method names.
- *thing* is specific to each method, usually an object of the operation. May not always be present.

TABLE 1.1 *Verbs in API method names*

New	Creates a new unattached object. Example: AVMenuNew .
AddNew	Creates a new object using the specified parameters and adds the new object to the current object. Example: PDBBookmarkAddNewChild .
Add	Adds the second object as a child to the current object. Example: PDBBookmarkAddChild .
Create	Creates a new object of a given type. Example: PDDocCreatePage .
Destroy	Destroys the specified object and releases its resources immediately. Example: PDBBookmarkDestroy .
Open	Opens an object from storage or a stream. Example: AVDocOpenFromFile .
Close	Destroys an object that was opened. Closes the underlying storage or stream. Example: ASFfileClose .

TABLE 1.1 *Verbs in API method names*

Acquire	Obtains a shareable resource from a parent object. Or, increments a reference counter for an object. The shared object isn't destroyed until all acquirers have released it. Example: AVMenuItemAcquire .
Release	Releases a shared object. Example: PDPAGERelease .
Delete	Removes the second object from the current object and destroys the second object. Example: PDDocDeletePages .
Remove	Removes the second object from the current object but doesn't destroy it. Example: AVMenuRemove .
Get	Retrieves an attribute of the object. Example: AVWindowGetTitle
Set	Sets an attribute of the object. Example: PDAnnotSetFlags . (Note: Cos uses the verb Put).
Is	Retrieves a boolean attribute of the object. Example: PDBookmarkIsOpen .
Enum	Enumerates the specified descendant objects of the current object. Example: PDDocEnumFonts .

While many of the API method names follow this form, there are exceptions.

Conversion methods, for example, are of the form:

`<layer><object><source_object>to<dest_object>`

An example is [AVPageViewPointToDevice](#).

Get and **Set** methods are used for getting and setting object attributes. Each object type has zero or more attributes. For example, an annotation object ([PDAnnot](#)) contains attributes such as color and date. You can obtain and modify the value of an object's attributes using methods such as [PDAnnotGetColor](#) and [PDAnnotSetDate](#).

In some cases, the return value of a **Get** method is another object. For example, [AVDocGetAVWindow](#) returns an [AVWindow](#) object corresponding to a specified [AVDoc](#).

Other methods that return objects have the word **Acquire** in their name. These methods are always paired with a corresponding **Release** method, and have the additional side effect of incrementing or decrementing a reference count. The core API uses **Acquire/Release** methods, for example, to determine whether or not it is safe to free a memory structure representing an object.

If you use an **Acquire** method to obtain an object, you must subsequently use the **Release** method to correctly update the reference counter, as shown here:

```
PDDoc doc;
PDPAGE page;

doc = PDDocOpenFromASFFile ("AFILE.PDF", NULL, TRUE);
```

```
page = PDDocAcquirePage (doc, 42);

/* Now we're done with page */
PDPageRelease (page);
```

In the code above, note that the **PDPage** is acquired through the document that contains it (using **PDDocAcquirePage**), and is released using **PDPageRelease**. Failure to match **Acquire/Release** pairs generally results in Acrobat complaining that a document cannot be closed due to non-zero reference counts.

Because the core API does not keep track of objects that do not have **Acquire/Release** methods, there is no way for Acrobat plug-ins to know when such objects are being used, and when they can be deleted safely. For this reason, the API provides validity testing methods your plug-in can use to determine whether or not an object previously obtained using a **Get** method is still usable. **IsValid** typically is included in the name of a validity testing method, for example **PDAnotIsValid**. You can check if an object has an associated validity testing method by looking up the object in the “Objects” section in the *Acrobat Core API Reference*.

Data Types

The core API uses five types:

- Scalar Types
- Simple Types
- Complex Types
- Opaque Types
- Cos Objects

Scalar Types

Scalar (non-pointer) types are based on underlying C language types, but have platform-independent bit sizes. They are defined in the header file **CoreExpT.h**. All scalar types are AS layer types.

For portability, enumerated types are defined using a type of known size such as **ASEnum16**.

Table 1.2 lists and describes the scalar types.

TABLE 1.2 *Scalar types*

Type	Size (in bytes)	Description
ASBool	2	boolean

TABLE 1.2 *Scalar types*

ASUns8	1	unsigned char
ASUns16	2	unsigned short
ASUns32	4	unsigned long
ASInt8	1	char
ASInt16	2	signed short
ASInt32	4	signed long
ASEnum8	1	enum (127 values)
ASEnum16	2	enum (32767 values)
ASFixed	4	fixed point integer, 16 bits for mantissa and 16 bits for fractional part
ASSize_t	4	size of objects (as in size_t)

Simple Types

Simple types represent abstractions such as a rectangle or matrix. These objects have well-known fields that are not expected to change.

Examples of simple types are:

- **ASFixedRect**
- **ASFixedMatrix**
- **AVRect32**

NOTE: Two different coordinate systems are used for rectangles. See “[Understanding Coordinate Systems](#)” on page 28 for details.

Complex Types

Complex types are structures that contain one or more fields. They are used in situations such as the following:

- To transfer a large number of parameters to or from a method. For example, the API method **PDFFontGetMetrics** returns font metrics by filling out a complex structure (**PDFFontMetrics**).
- To define a data handler or server. For example, your plug-in must provide a complex structure filled out with callback methods (**AVAnnotHandlerRec**) when it intends to register an annotation handler.

Because a complex type may change over time (by adding new fields or obsoleting old ones), the size of the type is specified either as the first field of the type or as a separate parameter to a method. A core API method can examine this field to determine whether a new callback method is available, or whether a new data field should be filled out.

Opaque Types

Many methods in the core API hide the concrete C-language representation of data structures from plug-ins. Most of these methods take an object and perform an action on that object. The objects are represented as opaque types.

Examples of opaque objects are `PDDoc` and `AVPageView`.

Cos Objects

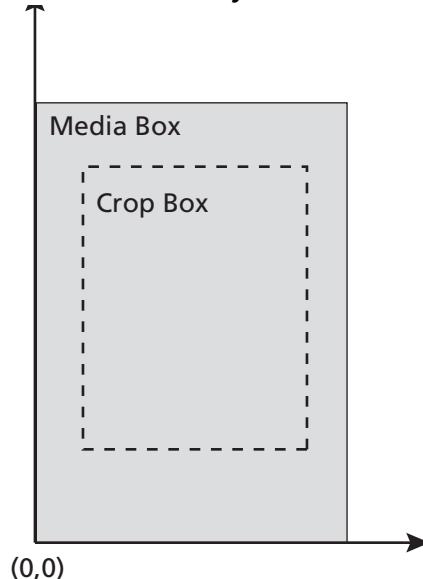
A Cos object in the core API (type `CosObj`) refers to its corresponding Cos object in the PDF document. Cos objects are represented as opaque 8-byte structures. They have subtypes of boolean, integer, real, name, string, array, dict, and stream.

Understanding Coordinate Systems

The core API defines two coordinate systems: *user space* and *device space*. In addition, some methods make use of a platform's native coordinate system, which is known as *machine port space*. This section describes each of these coordinate systems.

User Space

User space specifies coordinates for most objects accessed using PD layer methods. It is the coordinate system used within PDF files. [Figure 1.3](#) shows the user space coordinate system. In the figure, as in PDF, the *media box* is the rectangle that represents that page's size (for example, US letter, A4). The *crop box* is an optional rectangle that is present if the page has been cropped (for example, using the **Document -> Crop Pages...** menu item in Acrobat).

FIGURE 1.3 User Space Coordinate System

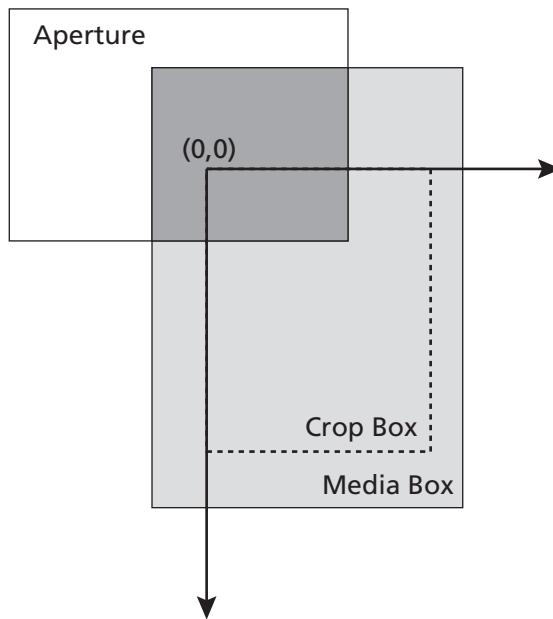
The default origin of the user space coordinate system is the lower left corner of a page's media box. The value of the x-coordinate increases to the right, and the value of the y-coordinate increases upward. Coordinates are represented as fixed point numbers, and rectangles are of type **ASFixedRect**.

Device Space

Device space specifies coordinates in screen pixels. See [Figure 1.4](#). Plug-ins use this coordinate system when calling AV layer methods to specify the screen coordinates of windows.

NOTE: Device space coordinates generally are not equal to points. One point is approximately 1/72 of an inch. Pixels and points are only nearly equivalent when the monitor has a resolution of 72 dpi and the zoom factor is 1.0.

FIGURE 1.4 Device Space Coordinate System



Device space defines an *aperture* as that portion of the Acrobat viewer's window in which the PDF file is drawn. The origin of the device space coordinate system is at the upper left corner of the visible page on the screen. The value of the x-coordinate increases to the right, and the value of the y-coordinate increases downward. Coordinates are represented as integers, and rectangles are of type **AVRect**.

NOTE: The upper left corner of the visible page is determined by the intersection of a page's PDF crop box and media box. As a result, the device space coordinate system changes when the cropping on a page changes.

Translating between User Space and Device Space

Sometimes a plug-in must translate user space coordinates to device space, and vice versa.

Say, for example, you want your plug-in to draw a rectangle on top of an annotation. You can get an annotation's bounding rectangle with **PDAnnotGetRect**. This rectangle is in *user space*, the coordinates of the PDF file, and is invariant of the view of the PDF file. You can draw a rectangle with the **AVPageViewDrawRect** method, but because this method is an AV layer method, it requires a rectangle in *device space* coordinates. The **AVPageViewRectToDevice** method translates a rectangle's coordinates from user space to device space, so the following code would draw the rectangle:

```
ASFixedRect userRect;
AVRect deviceRect;
```

```
PDAnotGetRect(anAnnot, &userRect);
AVPageViewRectToDevice(pageView, &userRect, &deviceRect);
AVPageViewDrawRect(pageView, &deviceRect);
```

If more than one page is displayed, as in the continuous display modes of Acrobat, *coordinates in user space may be ambiguous*. The problem is that user space coordinates are relative to a page, and more than one page is displayed. This raises the question of which page **AVPageViewRectToDevice** would use. To specify the page, call the **AVPageViewSetPageNum** method first. The code shown above now appears as:

```
ASFixedRect userRect;
AVRect deviceRect;

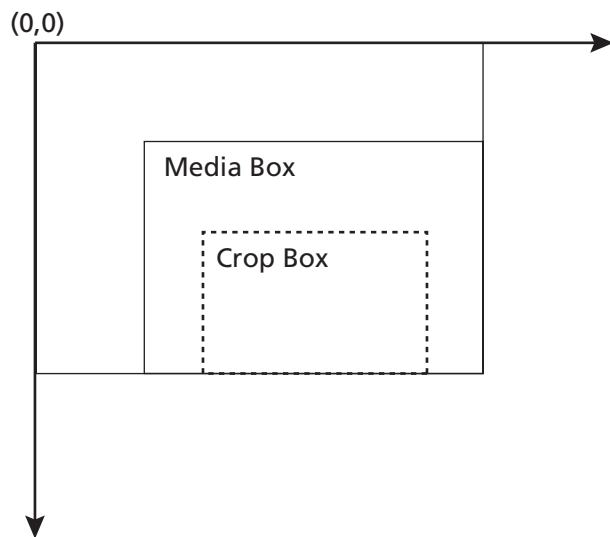
AVPageViewSetPageNum(pageView, annotPageNum);
PDAnotGetRect(anAnnot, &userRect);
AVPageViewRectToDevice(pageView, &userRect, &deviceRect);
AVPageViewDrawRect(pageView, &deviceRect);
```

Machine Port Space

You would use *Machine port space* if, for example, your plug-in needs to draw on the screen using QuickDraw (on the Macintosh platform) or GDI (on the Windows® platform). Machine port space is shown in [Figure 1.5](#).

If an object's coordinates are specified in user space and a plug-in needs to draw the object to the machine port, it must translate the points through the matrix obtained from **AVPageViewGetPageToDevMatrix**. On Macintosh only, it must also subtract the **left** field of the window's aperture from the x-coordinate and subtract the **top** field from the y-coordinate. The plug-in can then draw using GDI or QuickDraw.

If device space changes, any acquired machine ports are *not* updated to track device space; their coordinate systems are still set to the device space in effect when the ports were acquired.

FIGURE 1.5 Machine Port Space Coordinate System

Using Rectangles and Quadrilaterals

Rectangles (`rects`) and quadrilaterals (`quads`) are used in the core API. Both are geometric shapes with four rectilinear sides. A rectangle is specified by two corner points, and the rectangle's sides are always vertical and horizontal. A quad is specified by all four corner points, and the sides can have any orientation.

A plug-in should use rectangles as frequently as possible, because it can specify them with half as much data as a quad requires, and they are processed more efficiently. A plug-in should use quads when necessary, though; for example, to specify the box containing a rotated word.

Handling Exceptions

In general, methods do not provide return values but instead raise *exceptions* when errors occur. You can write *exception handlers* to catch and handle exceptions at different points in your plug-in. Acrobat viewers contain a default handler to deal with otherwise uncaught exceptions.

[Chapter 12, “Handling Errors,”](#) describes exception handling in detail.

Adding New Object Types

Plug-ins can extend the range of objects that Acrobat understands. For example, they can define new types of annotations and new tools. A plug-in generally adds new types by passing to Acrobat a structure (known as a *handler*) containing a number of function pointers. The number of function pointers and their purpose depend on what object a plug-in is adding, but functions usually include those for creating, destroying, and drawing the new object. [Chapter 10, “Handlers,”](#) describes each of the handlers a plug-in can add.

Storing Private Data in PDF Files

Plug-ins can store private data in PDF files, although private data must be stored in such a way that the file can still be drawn by a standard Acrobat viewer. Adobe maintains a registry of private PDF dictionary key names to reduce the possibility of a plug-in’s key names conflicting with names belonging to other plug-ins.

Private dictionary keys exist in three categories:

1. Specific keys that are proposed by third parties but are generally useful. Adobe maintains a registry of these names.
2. Keys registered by third parties as well as keys whose prefix is registered that are applicable only to a limited set of users. Adobe maintains a registry of these names and prefixes. For more information on registering and using plug-in prefixes, see Chapter 3 in [Acrobat Development Overview](#).
3. Keys that begin with a special prefix reserved by Adobe for private extensions. These keys are intended for use in files that are never seen by other third parties, since these keys may conflict with keys defined by others.

Contact the [Adobe Solutions Network Web site](#) to register private data types.

This chapter describes many of the details needed to understand how plug-ins work with the Acrobat core API. The chapter discusses a number of topics basic to plug-in development, including:

- Host Function Tables
- Replacing Built-In Methods
- Interaction Between Plug-ins and the Acrobat Viewer
- Callbacks
- Notifications
- Enumeration
- Handling Events
- Adding Message Handling
- Acrobat and Reader Differences
- Changing the Acrobat Viewer User Interface
- Page View Layers
- Reducing Conflicts Among Plug-ins

NOTE: Many of these topics, but not all, apply to development with the PDF Library.

Host Function Tables

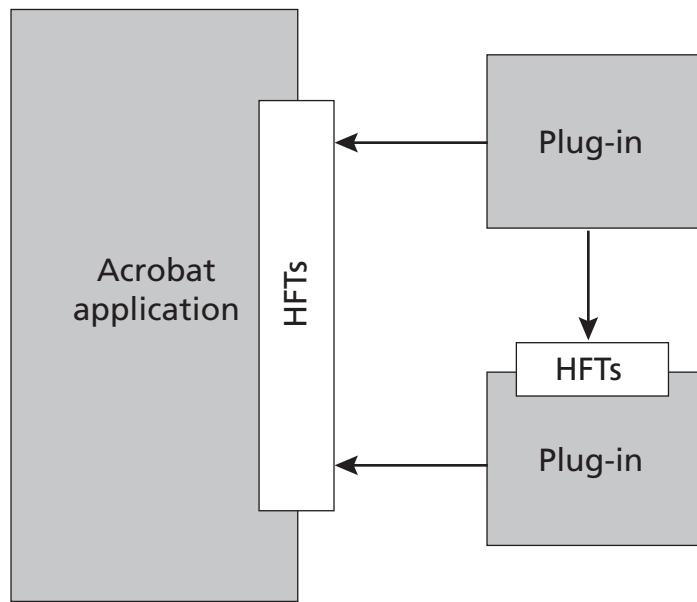
Host Function Tables (HFTs) are the mechanism through which plug-ins call methods in the Acrobat viewer or in other plug-ins. See [Figure 2.1](#).

An HFT is a table of function pointers. Each HFT has:

- A name
- A version number
- An array of one or more *entries*

Each entry represents a single method that plug-ins can call, and is set up as a linked list of function pointers. Acrobat uses linked lists because some HFT entries may be marked so that they can be replaced by a plug-in. Also, it is useful to keep a list of each implementation of a method that has been replaced (to allow methods to call the implementations they replaced).

FIGURE 2.1 Host Function Tables (HFTs)



Using HFTs

Plug-ins must use the [ASExtensionMgrGetHFT](#) method to import each HFT they intend to use. A plug-in requests an HFT by its name and version number. This importing takes place during plug-in initialization, which is described in “[Importing HFTs and Registering for Notifications](#)” on page 40.

When a plug-in calls a method in the Acrobat viewer or in another plug-in, the function pointer at the appropriate location in the appropriate HFT is dereferenced and executed. Macros in the Acrobat SDK’s header files hide this from you, so that plug-ins contain only what appear to be normal function calls.

HFT Servers

Each HFT is serviced by an HFT server. The HFT server is responsible for handling requests to obtain or destroy its HFT. As part of its responsibility to handle requests to obtain an HFT, the server can choose to support multiple versions of the HFT. These versions generally correspond to versions of the Acrobat viewer or of the plug-in that exposes the HFT. The ability to provide more than one version of an HFT improves backward compatibility by allowing existing plug-ins to continue to work when new versions of the Acrobat viewer (or other plug-ins whose HFTs they use) are produced. It is expected that HFT versions typically will differ only in the number—not the order—of methods they contain. In this case, supporting different HFT versions is straightforward, since all versions can use the same table but simply advertise it as having different lengths.

Creating a New HFT

Plug-ins can create their own HFTs, allowing other plug-ins to invoke one or more methods in them. For example, the Acrobat Search plug-in creates its own HFT to allow other plug-ins to programmatically perform cross-document searches. Plug-ins may allow one or more methods in their own HFTs to be replaced.

To create a new HFT, use the following procedure:

1. Invoke **HFTServerNew**, specifying a name for the HFT server, a procedure that returns an HFT specified by name and version number, and a procedure that handles requests to destroy the HFT server. Your plug-in also can specify private data.
2. Invoke **HFTNew** from within its HFT-providing procedure to create an empty HFT that can hold a specified number of methods.
3. Use **HFTReplaceEntry** to populate the entries in the HFT with pointers to the methods it is making available for other plug-ins to call.

For an example of how to create an HFT, see the *Acrobat Plug-In Tutorial*.

Replacing Built-In Methods

There are a small number methods in the Acrobat HFTs that can be replaced by plug-ins. For example, a plug-in could use this mechanism to change the appearance of all alert boxes displayed by the Acrobat viewer, or to override file opening behavior. For a list of all the replaceable Acrobat and Acrobat Reader methods, see “Replaceable Methods” in the section entitled “Lists” in the *Acrobat Core API Reference*.

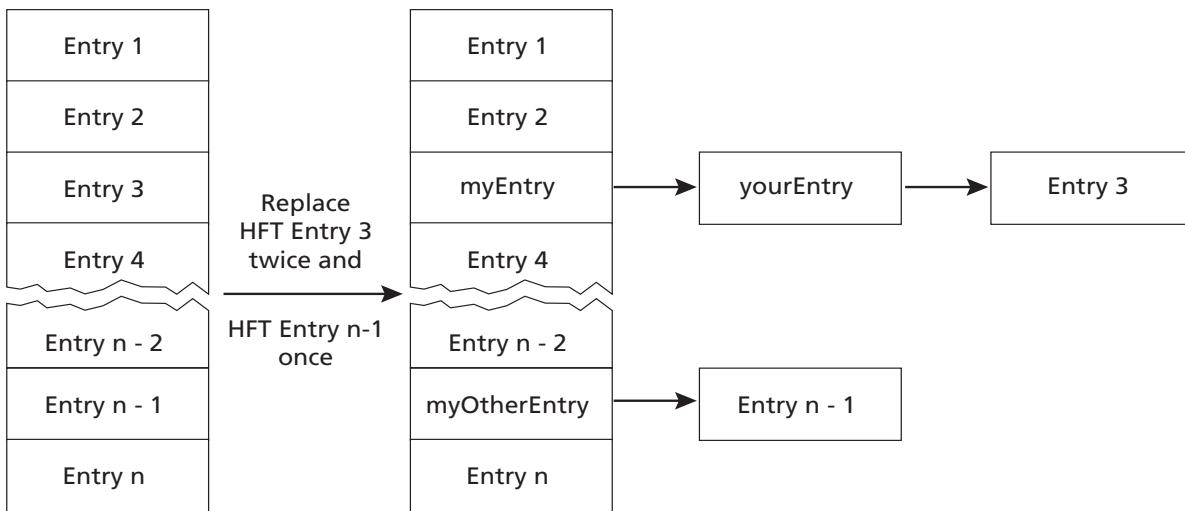
To replace one of these methods, a plug-in calls the **HFTReplaceEntry** method. In some cases, when the replacement method has finished executing, it should call the previous implementation of the method, using the **CALL_REPLACE_PROC** macro, to allow previously-registered implementations of the method (including the viewer’s built-in implementation) to execute. Previous implementations of the method are not called automatically; it is up to the replacement implementation to call them.

When a plug-in replaces a method in the Acrobat HFTs, it should allow its implementation of that method to be replaced. If, for example, your plug-in replaces the Acrobat viewer’s **AVAlert** method, it should not prevent other plug-ins from also replacing **AVAlert**.

All plug-ins, and the Acrobat viewer, share a single copy of each HFT. As a result, when a plug-in replaces a method’s implementation, all other plug-ins and the Acrobat viewer also use the new implementation of that method. In addition, once a method’s implementation has been replaced, there is no way to remove the new implementation without restarting the Acrobat viewer.

When an HFT entry is replaced, the entry’s linked list is updated so that the newly-added implementation is at the head of the linked list. Previous implementations, if any, follow (in order) as illustrated in Figure 2.2.

FIGURE 2.2 HFT Entry Replacement



Interaction Between Plug-ins and the Acrobat Viewer

A plug-in is a dynamic link library (DLL) on the Windows platform and a shared library on the Macintosh and UNIX platforms.

NOTE: On the Windows platform, plug-in names must end in `.API`, not `.DLL`. On UNIX, plug-in names must end in `.API` and the plug-in path must be specified correctly in the `.acrorc` file.

This section describes the sequence of operations the Acrobat viewers perform to initialize a plug-in and the operations a plug-in should perform in each step of the sequence.

There are several ways in which a plug-in can register one or more of its functions with the Acrobat viewer so that it can continue to interact with the Acrobat viewer after initialization. These include:

- Adding menu items or toolbar items that call the plug-in's routines.
- Registering a routine to be called when a certain event occurs (these routines are called *notifications*).
- Replacing an existing Acrobat viewer method, such as the method that opens files.
- Registering to receive Interapplication Communication (IAC) messages that other applications send to Acrobat viewer.

Locating Plug-ins

When it launches, Acrobat searches for plug-ins. It searches a directory named `Plug-Ins` in the same directory as the Acrobat viewer executable. In addition,

Acrobat searches any folders contained inside these folders when looking for plug-ins. This search only goes one level deep.

Windows plug-ins are identified by the .API suffix. Macintosh plug-ins must have a file type and creator of XTND and CARO, respectively.

The Acrobat viewer displays a progress message in the bottom line of the splash screen while it initializes. No plug-ins load if the **Shift** key is held down while the Acrobat viewer launches.

Handshaking and Initialization

The Acrobat viewer performs a handshake with each plug-in as it is opened and loaded. During handshaking the plug-in specifies its name, several initialization procedures, and an optional unload procedure.

The plug-in must implement the following handshaking function:

```
ACCB1 ASBool ACCB2 PIHandshake(ASUns32 handshakeVersion, void, *hsData)
```

During handshaking, the plug-in receives the **hsData** data structure shown in [Table 2.1](#) (see `PIVersn.h`). The Acrobat viewer converts all function pointers that are passed in this data structure into callbacks using `ASCallbackCreateProto`. See “[Callbacks](#)” on page 42 for more information.

TABLE 2.1 Handshake data structure

handshakeVersion (Passed to the plug-in)

The version of the handshaking data structure used. Currently `HANDSHAKE_V0200`.

extensionName (Required)

The `ASAtom` corresponding to the plug-in’s name. You can convert the plug-in’s name to an `ASAtom` using `ASAtomFromString`. The name should be less than 25 characters.

PluginExportHFTs (Optional)

Only plug-ins that provide methods that other plug-ins can call use this callback procedure. The Acrobat viewer calls `PluginExportHFTs` after it completes handshaking with all plug-ins. The only task this callback should perform is to export the plug-in’s own methods.

PluginImportReplaceAndRegister (Optional)

The Acrobat viewer calls this callback procedure after it has loaded all plug-ins and the plug-ins have exported their methods. Plug-ins should use this callback to import any methods they use from other plug-ins, replace functions in the Acrobat API, and register for notifications. If your plug-in replaces any API methods, it must do so in this procedure; methods must not be replaced at any other time.

NOTE: Your plug-in can register and unregister for notifications at any time; it does not have to do so in this procedure. On the other hand, if any method replacement is to be performed, it *must* be done in this procedure.

TABLE 2.1 **Handshake data structure** (Continued)**PluginInit** (Required)

The Acrobat viewer calls this callback procedure after it finishes calling each plug-in's **PluginImportReplaceAndRegister** callback procedure.

PluginInit finishes initializing the plug-in and adds the plug-in's menus, menu items, and so on.

PluginUnload (Optional)

Upon exit, the Acrobat viewer calls this callback procedure to release memory, undoes any changes the plug-in made to the Acrobat viewer user interface, and so on.

The plug-in must fill in its extension name and implement, at minimum, the initialization callback **PluginInit**.

The `DUCallbacks.h` header file declares all callbacks for your plug-in. Declarations are:

```
ACCB1 ASBool ACCB2 PluginExportHTFs(void);
ACCB1 ASBool ACCB2 PluginImportReplaceAndRegister(void);
ACCB1 ASBool ACCB2 PluginInit(void);
ACCB1 ASBool ACCB2 PluginUnload(void);
```

All callbacks return **true** if your plug-in's procedure completes successfully or if the callbacks are optional and are not implemented. If your plug-in's procedure fails, the callbacks return **false**.

NOTE: In addition to defining the callbacks listed above, `DUCallbacks.h` includes the the Adobe Dialog Manager header file `ADMUtilities.h`.

If either the Acrobat viewer or a plug-in aborts handshaking, the viewer displays an alert dialog showing a brief explanation. Then it continues loading other plug-ins.

Exporting HFTs

After the Acrobat viewer finishes handshaking with all the plug-ins, it calls each plug-in's **PluginExportHTFs** callback procedure. In this procedure, a plug-in may export any HFTs it intends to make available to other plug-ins. This callback should only export an HFT, not call any Cos, PD, or AV layer methods.

NOTE: This is the only time at which a plug-in can export an HFT.

Importing HFTs and Registering for Notifications

After the Acrobat viewer completes calling each plug-in's **PluginExportHTFs** callback procedure, it calls each plug-in's **PluginImportReplaceAndRegister** callback procedure. In this procedure, plug-ins may perform three tasks:

- Plug-ins may import any special HFTs they use (the standard Acrobat viewer HFTs are automatically imported). Plug-ins also may import HFTs any time after this while the plug-in is running.
- Plug-ins may replace any of the Acrobat viewer's replaceable methods (see “[Replacing Built-In Methods](#)” on page 37). Any method replacement must be performed at this time.
- Plug-ins may register for notifications (see [AVAppRegisterNotification](#)). Plug-ins also may register and unregister for notifications any time after this while the plug-in is running. A plug-in may receive a notification any time after it has registered for it, even if the plug-in's initialization callback has not yet been called. This can occur if another plug-in initializes first and performs an operation, such as creating a PDF document, which causes a notification to be sent. Plug-ins must be prepared to correctly handle notifications as soon as they register for them.

This callback should not call any Cos, PD, or AV layer methods.

NOTE: This is the only time a plug-in may import an HFT or replace a standard API method. Plug-ins may register for notifications at this time or any time afterward.

Initialization

After the Acrobat viewer completes calling each plug-in's [PluginImportReplaceAndRegister](#) callback procedure, it calls each plug-in's initialization procedure. Plug-ins should use their initialization procedures to hook into the Acrobat viewer's user interface by adding menu items, toolbar buttons, windows, and so on. It is also acceptable to modify the viewer's user interface later when the plug-in is running.

If your plug-in needs to carry out a task after *all* plug-ins have been initialized, it should register for the [AVAppDidInitialize](#) notification. This notification is broadcast when the Acrobat viewer has finished initializing and is about to enter its event loop.

Unloading

A plug-in's unload procedure should free any memory the plug-in allocated and remove any user interface changes it made. Acrobat calls this procedure when the viewer terminates or when any of the other handshaking callbacks return `false`. This function should:

- Remove and release all menu items and other user interface elements, HFTs, and HFTServers .
- Release any memory or any other resources allocated.

Currently, plug-ins unload only when the Acrobat viewer exits.

Callbacks

There are several situations in the core API where Acrobat will call routines that your plug-in has provided . These include:

- The `PIHandshake` routine
- When deciding to enable or mark items in menus or in the toolbar
- When menu items or toolbar buttons are clicked
- In enumeration methods
- Dialogs (filter procs, and so forth)
- Notifications

Whenever your plug-in passes a function pointer to the Acrobat viewer, it must first turn it into an `ASCallback` object. This allows the compiler to check that the correct prototypes are used for the functions for which callbacks are created.

Use `ASCallbackCreateProto`, `ASCallbackCreateReplacement`, and `ASCallbackCreateNotification` to convert functions into callbacks and to perform type checking on the function being converted.

NOTE: Type checking only occurs if the `DEBUG` macro is set to `1` at the time your plug-in is compiled. Be sure to set it appropriately in your development environment and remove it when you build the shipping version of your plug-in.

You can create a callback using this example as a template:

```
myProcType myCallback = ASCallbackCreateProto(myProcType, &myProc);
```

where `myProc` is the procedure to be converted into a callback, and `myProcType` is the procedure's type. The type checking performed is important, because it eliminates an extremely common source of bugs.

`ASCallbackCreateProto` returns a pointer to a function that may be directly called by any plug-in or the viewer application. Use `ASCallbackDestroy` to dispose of a callback that is no longer needed.

All callbacks *must* be declared with Pascal calling conventions. To make your code portable between platforms, declare all your callback functions using the **ACCB1** and **ACCB2** macros, for example:

```
static ACCB1 const char* ACCB2 getThingName(Thing* foo);
```

Declare any function pointer **typedefs** using the **ACCBPROTO1** and **ACCBPROTO2** macros, for example:

```
typedef ACCBPROTO1 const char* (ACCBPROTO2* fooProc) (Thing* foo);
```

You can use this example as a template to set up an **AVMenuItem** using callbacks:

```
/* All AVExecuteProcs must be declared with ACCB1/2 */

static ACCB1 void ACCB2 executeMyItem(void* clientData)
{
    AVAlertNote("There is a document open.");
}
static ACCB1 boolean ACCB2 isMyItemEnabled(void* clientData)
{
    return (AVAppGetNumDocs != 0);
}

static ACCB1 boolean ACCB2 isMyItemMarked(void* clientData)
{
    return false;
}
/* We do not need to use ACCB1/2 with this function, because it is not
being called from outside this plug-in. */

static void SetUpMyMenuItem(void)
{
    AVMenuItem menuItem = AVMenuItemNew("My Item", NULL, NULL, false,
        NO_SHORTCUT, NULL, gExtensionID);

    /* We must use ASCallbackCreateProto on our execute proc, because it
    will be called from outside the plug-in */

    AVMenuItemSetExecuteProc(menuItem,
        ASCallbackCreateProto(AVExecuteProc, &executeMyItem), NULL);

    /* Ditto for the compute-enabled proc and compute-marked proc */

    AVMenuItemSetComputeEnabledProc(menuItem,
        ASCallbackCreateProto(AVComputeEnabledProc, &isMyItemEnabled),
        NULL);
    AVMenuItemSetComputeMarkedProc(menuItem,
        ASCallbackCreateProto(AVComputeMarkedProc, &isMyItemMarked), NULL);
}
ACCB1 boolean ACCB2 initMyPlugin(void)
{
    SetUpMyMenuItem;
```

```
}
```

Notifications

The core API provides a *notification* mechanism so that plug-ins can synchronize their actions with Acrobat. Notifications allow a plug-in to indicate that it has an interest in a specified event (such as an annotation being modified) and provide a procedure that Acrobat calls each time that event occurs.

NOTE: See “Notifications” in the *Acrobat Core API Reference* for a complete list of notifications.

To receive notifications, follow these steps:

1. Use the **ASCallbackCreateNotification** macro to convert your procedure into a callback to be passed to the notification.

NOTE: **ASCallbackCreateNotification** only performs type checking if you have defined the **DEBUG** macro as **1** before compiling your plug-in. Remember to define **DEBUG** as **0** before compiling a shipping version of your plug-in.

2. Call the **AVAppRegisterNotification** method to have your function called back for a particular event.
3. Call **AVAppUnregisterNotification** if your plug-in is registered for a particular notification and no longer wants to receive notifications for it.

NOTE: Remember to pass the same **ASCallback** you created with your function. Passing a pointer to your function, rather than the **ASCallback** you created, into **AVAppUnregisterNotification** is a common mistake.

The order in which notifications occur varies among platforms. For instance, after opening an **AVDoc**, notifications may occur in this order on the Macintosh platform:

1. **AVPageViewDidChange**
2. **AVDocDidActivate**
3. **AVPageViewDidChange**
4. **AVDocDidOpen**

On the Windows platform, notifications may occur in this order:

1. **AVPageViewDidChange**
2. **AVDocDidOpen**
3. **AVDocDidActivate**
4. **AVPageViewDidChange**

NOTE: A plug-in may receive a notification *at any point* after registering for one, even if the plug-in's initialization procedure hasn't been called yet. Plug-ins need to allow for this possibility.

Enumeration

The core API provides several methods that enumerate all objects of a particular type. This can be useful, either because there is no way to access the objects (such as **PDPPaths**) directly, or for convenience (such as using **PDDocEnumFonts** to enumerate all fonts used in a document).

When these methods are called, the Acrobat viewer enumerates the specified objects, and calls a plug-in-specified callback procedure for each object of that type. For example, when **PDDocEnumFonts** is called, Acrobat enumerates all fonts used in the document, calling a procedure provided for each font it finds. Enumeration is complete after the enumeration method returns.

Your plug-in can call an enumeration method and create an array of found elements to be used later. Alternately, your plug-in can search for a particular item and, upon finding the item, stop the enumeration and immediately return. Using enumeration methods, your plug-in can find any toolbar or menu item, or any number of elements on a page.

Enumeration methods may take a *monitor* as a parameter. A monitor is a C structure that contains pointers to one or more plug-in-supplied callback procedures. The API calls one or more of the functions defined in the monitor for each object in a list. One method that uses a monitor is **PDPPathEnum**, which provides a set of callbacks for each path object in a page's *display list* (list of marking operations that represent the displayed portion of a page). This allows a plug-in to be aware of (but not to alter the rendering of) the path objects in a page.

Handling Events

Mouse Clicks

Mouse clicks are first passed to any procedures registered using **AVAppRegisterForPageViewClicks**. If all of those procedures return *false*, the click is passed to the active tool. If that returns *false*, the click is passed to any annotation at the current location.

Adjust Cursor

Adjust cursor events are first passed to any procedures registered using **AVAppRegisterForPageViewAdjustCursor**. If all of those procedures return

false, the event is passed to the active tool. If that returns *false*, the event is passed to any annotation at the current location.

Key Presses

Key presses are first passed to the currently active selection server. If the selection server's `AVDocSelectionKeyDownProc` callback returns `false`, the Acrobat viewer handles special keys (`Esc`, `Page Up`, `Page Down`, ...) or uses the key to select a tool from the toolbar.

Adding Message Handling

Plug-ins can add their own Apple events and DDE messages to those supported by the Acrobat viewers. On Windows, plug-ins can register to receive DDE messages directly.

NOTE: Plug-ins should use the DDEML library to handle DDE messages. Problems may arise if they do not.

On the Macintosh, plug-ins must hook into the Acrobat viewer's Apple event handling loop to handle Apple events. To do this, replace the API's

`AVAppHandleAppleEvent` method (see “Replacing Built-In Methods” on page 37).

If a plug-in receives an Apple event it does not want to handle, it should invoke the implementation of the method it replaced, allowing other plug-ins or the Acrobat viewer the opportunity to handle the Apple event.

The viewers on UNIX® do not currently provide built-in IAC support, but plug-ins can add IAC support via RPC or other mechanisms.

Plug-in Prefixes

You also should observe the conventions for naming and registering your plug-in and parts that it may contain such as HFTs, menus, menu items, and so forth. For details, see Chapter 2, “Registering and Using Plug-in Prefixes,” in the *Acrobat Development Overview*.

Acrobat and Reader Differences

Both Acrobat and Reader accept plug-ins. The difference between the two is that Reader can neither make changes to a file nor save a file. the Reader does not include API methods that change or save files.

NOTE: Reader only accepts Reader-enabled plug-ins. Contact Adobe's Developer Technologies group for information on the licensing terms for creating Reader-enabled plug-ins.

Plug-ins that cannot function fully under Reader must use the [ASGetConfiguration](#) method to check which of the Acrobat viewers is running. Failure to do so will, at best, expose the user to a variety of error alerts. If such a plug-in finds that it is running under Reader, it should usually notify the user that it cannot function fully, then proceed in one of several ways:

- Not load.
- Omit toolbar buttons and menu items that enable editing.
- Display disabled (grayed-out) toolbar buttons and menu items that enable editing.

Plug-ins that need to check whether or not they are running under Reader should do so as early in initialization as possible (see ["Interaction Between Plug-ins and the Acrobat Viewer" on page 38](#) for a discussion of initialization).

Plug-ins that create and manipulate custom annotations should allow their annotations to be displayed (they cannot be created, deleted, or edited) when running under Reader.

Changing the Acrobat Viewer User Interface

This section describes the kinds of things a plug-in can do to change the Acrobat viewer user interface.

Adding or Removing Menus and Menu Items

Plug-ins can add new menus and add items to any menu. They can also remove any menu or menu item.

Menu items added by plug-ins can have shortcuts (keyboard accelerators). The Acrobat viewer does not ensure that plug-ins add unique shortcuts, but it is possible for a plug-in to check which shortcuts are already in use before adding its own. Note that the only way to ensure there are no shortcut conflicts is for all plug-ins to check for conflicts before adding their own shortcuts.

You are encouraged to have your plug-in add its menu items to the **Tools** menu. When it is launched, the Acrobat viewer automatically adds this menu, as well as the **About Plug-ins** and **Plug-in Help** menus. After Acrobat loads all plug-ins, it checks these three menus and removes any that are empty.

Adobe keeps a registry of plug-in menu item names to help avoid conflicts between plug-ins. For more information on registering and using plug-in prefixes, see Chapter 3 in the [Acrobat Development Overview](#).

Modifying the Toolbar

Plug-ins can add items to the toolbar, although the size and resolution of the user's monitor can limit the number of tool buttons that can be added.

Plug-ins can remove buttons from the toolbar.

Plug-ins also can also create new toolbars, called *flyouts*, that can be attached to buttons on the main toolbar. The selection tools, for example, are all on a flyout. As of Acrobat 4.0, it is no longer true that all tool buttons are located on the main toolbar; some may be located on a flyout.

Controlling the “About” Box and Splash Screen

Plug-ins can set values in the preferences file (using the [AVAppSetPreference](#) method) to prevent the Acrobat viewer “About” box and/or splash screen from appearing before the viewer displays the first document. These changes take effect the next time the Acrobat viewer is launched.

About Acrobat Plug-Ins is a standard menu item in the **Help** menu. This menu item contains a submenu. You are encouraged to have your plug-in add a menu item to the submenu to bring up its own “About” box.

Placing Plug-in Help Files In a Standard Location

The **Help** directory that accompanies the Acrobat viewer application provides a standardized location for your plug-in help files. In Acrobat 4.0 and later, you can place a help file either in this **Help** directory or in a subdirectory of the **Help** directory. If, for example, your plug-in is localized for Japanese, you might want to place its Japanese help file in a **Help_JPN** subdirectory. To aid in opening locale-specific help files, the core API provides the [AVAppOpenHelpFile](#) method.

Page View Layers

The Acrobat viewer's drawing and mouse click processing relies on the concept of *page view layers*, which are numbers of type **ASFixed** that are associated with the document itself and each annotation type. The following table shows the predefined layers used by the Acrobat viewer.

Layer	Item
0	Page contents
LINK_LAYER (1)	Links
NOTE_LAYER (3)	Closed notes. Open notes are just above this.

These layers are used in the following situations:

- Drawing: the layers are drawn from lowest to highest. As indicated in the table, the page contents are drawn first, followed by links, closed text notes, and finally open text notes. As a result, open text notes draw over any closed text notes they overlap.
- Mouse click processing: occurs from highest layer to lowest layer. When a mouse click occurs, it is first passed to any open text note at the mouse click's location, then any closed text note, then any link, and finally to the page view itself. However, mouse clicks are passed to a lower layer only if a higher layer declines to handle the mouse click by returning `false` from its `DoClick` callback. (See the callbacks section of the [Acrobat Core API Reference](#) for a discussion of the various `DoClickProc` callbacks).

Annotation handlers provided by plug-ins can reside in any layer. (See “[Annotation Handlers](#)” on page 162 for more information.) For example, a plug-in could choose for its annotations to be between the page contents and links, such as in layer 0.5 (because layers are numbers of type `ASFixed`).

An annotation handler’s `AVAnnotHandlerGetLayerProc` callback is called during screen updates and mouse clicks to return its layer. Using a callback rather than a constant value allows an annotation’s layer to change, if desired. For example, the Acrobat viewer’s built-in text annotation changes its layer, allowing open text annotations to receive mouse clicks before closed annotations, if both are at the mouse click location. (On the other hand, the viewer’s built-in link annotation does not change its layer.)

NOTE: The Acrobat viewer does not poll `AVAnnotHandlerGetLayerProc` callbacks for changes in value, so be sure to invalidate the page rectangle of an annotation when its layer changes.

Reducing Conflicts Among Plug-ins

Most plug-ins can run without concern for what other plug-ins might be running at the same time. However, certain circumstances exist during which conflicts can arise. Specifically, if more than one plug-in overrides the same menu item or replaces the Acrobat viewer’s file access procedures, there is a possibility for conflict. To minimize this problem, you should code your plug-in such that it performs its special function. When it is done, it should then call the function it overrode. Coding your plug-in in this manner “reduces” the problem to the order in which conflicting plug-ins get to run, rather than to the plug-in that runs and locks out all others.

The Acrobat core API allows its clients to manipulate PDF file contents, to enhance and customize the Acrobat viewers to perform specialized functions, or to better integrate with existing environments. This chapter briefly describes some of the many possibilities and refers you to the corresponding sections of this document.

NOTE: Adobe may supply implementations of some of these applications with its products.

Controlling the Acrobat Viewers

Plug-ins can control the Acrobat viewer almost as if it were the user. All menu commands are available, plus additional functionality normally available from keyboard and mouse operations. Acrobat viewers can be instructed to run in the background or while they are minimized as icons.

For a discussion of menus, see

- “AVMenu” on page 76
- “AVMenubar” on page 77
- “AVMenuItem” on page 78

For a discussion of tool buttons, see

- “AVTool” on page 80
- “AVToolBar” on page 80
- “AVToolButton” on page 81

Drawing Into Another Window

Plug-ins can have the Acrobat viewer draw into an arbitrary window, allowing plug-ins to support PDF file viewing within their own user interface.

NOTE: It is also possible to draw into an arbitrary window using the interapplication communication (IAC) support present in the Acrobat viewers, as described in *Acrobat IAC Overview* and *Acrobat IAC Reference*. If you are interested in drawing into your own window, you should also read those documents to understand whether IAC or a plug-in is more appropriate for your needs.

When a plug-in redirects the Acrobat viewer's rendering into another window, the plug-in must handle mouse and keyboard events, make API calls for scrolling, change pages, zoom, or otherwise alter the view of the PDF file.

For more information, see the **PDPagedrawContentsToWindow** method in “**PDPagedrawContentsToWindow**” on page 99.

Indexed Searching

Indexed searching enables you to catalog, index, search, and highlight text in PDF files.

Regardless of document file format, simple sequential text searching is generally too time-consuming for long documents, and completely inadequate for searching a collection of documents.

Text retrieval systems overcome this problem by building a *search index* containing information on the location of all words in each document in the collection. A search system uses this index to determine which documents—and word locations within those documents—satisfy a given query. The search system then allows a user to browse the found documents, optionally displaying or highlighting the “hits,” or matching items.

Steps in the Acrobat Product’s Indexed Searching

The Acrobat core API enables plug-ins to extract the data necessary to build search indexes, open documents in an Acrobat viewer, and highlight words on pages.

Searching is not limited to page text. Text in annotations, bookmarks, and document-level attributes like Title and Subject can all be indexed and searched.

There are three steps to an indexed search of a PDF document. You can use one application for all three, or a separate application for each step.

1. Adding document-level information (optional)

In this step, you add to PDF files document-level information such as title, subject, author, and keywords. This *Document Info* allows users to locate specific documents easily, much like the card catalog in a paper-based library. Use of document-level fields to enhance searching is known as *fielded search* in many document retrieval systems.

Store document-level attributes in the Info dictionary in the PDF file format.

Plug-ins can use the **PDDocGetInfo** and **PDDocSetInfo** methods to read and modify document info fields.

Acrobat products, version 2.0 and higher, provide several ways to add document info fields to PDF files. The Distiller application and PDFWriter allow these fields to be set while generating a PDF file, and Acrobat allows users to edit **Document Info** fields.

2. Indexing

Indexing applications use the [PDWordFinder](#) object to extract text from PDF files and build a search index in a table or file. Through [PDWordFinder](#), plug-ins can obtain the character or word offset of each word in a PDF file, and length of each word. A plug-in can use the [PDDocGetInfo](#) method to obtain document-level attributes for it to index.

3. Searching and displaying search results

Through a user interface or some other means, a search application:

- Obtains a word or phrase to be found.
- Uses the search index and other API functions to open documents in an Acrobat viewer, display appropriate pages, and highlight targeted words.

Highlighting is limited to page text; text in an annotation or a bookmark cannot be highlighted. A plug-in can, however, select and show any annotation or bookmark in the document using the PDF file's **Document Info** fields as search criteria.

Extracting Text

The core API does not specify or constrain indexing applications. Your plug-in can create search indexes as desired. For example, some plug-ins add the search index filename to the PDF file's Info dictionary so that intra-document searching can be performed without the user having to specify the location of the search index.

Text is extracted in the same order as it occurs in the page's display list. This often is *not* the order in which a user would read the text. The application in which the original file was created determines the order in which text appears in the PDF file; different applications differ greatly.

NOTE: The order in which text appears in a file can affect phrase searches, proximity searches, and the operation of “next occurrence” functions.

In addition, individual words may be split into two or more pieces because PDF creators may emit kerned or differently-styled pieces of words at a different point in the page-generation sequence. The core API's word extraction algorithm attempts to reconstruct words by looking at the position and spacing of individual characters, and even handles the case in which words are hyphenated across lines.

In addition to the text of a word, plug-ins can also obtain the character and word offset from the beginning of the page and the number of characters in the word. This information is used to highlight the appropriate words in the Acrobat viewer during a search.

[“PDStyle” on page 101](#), [“PDWord” on page 105](#), and [“PDWordFinder” on page 106](#) describe the methods to obtain word, font, size, location and style information from a document.

Providing Document Security

PDF files may be encrypted, so that only authorized users can read them. In addition, the owner of a document can set permissions that prevent users from printing the file, copying text and graphics from it, or modifying it. Plug-ins can use the core API's built-in security, or they can write their own *security handlers* to perform user authorization in other ways (for example, by the presence of a specific hardware key or file, or by reading a magnetic card reader). See [Chapter 11, “Document Security,”](#) for more information.

Modifying File Access

Plug-ins can provide their own file access procedures that read and write data when requested by the Acrobat core API. Using this capability, a plug-in can enable PDF files to be read from on-line systems, e-mail, document management, or database programs.

In addition, plug-ins can allow other file formats to be imported to, or exported from, PDF using custom file access procedures. For importing, the random-access nature of PDF files makes it probable that the plug-in will have to write a complete PDF file to a local disk and use the core API to open that file.

For a discussion, see

- [“ASFileSys” on page 60](#)
- [“File Specification Handlers” on page 166](#)

Creating New Annotation Types

Plug-ins can create their own annotation types, including any data they need. For example, a custom annotation type might allow a user to draw (not just type) in an annotation, provide support for multiple fonts or text styles, or support annotations that can only be viewed by specific users.

For more information on annotations, see [“PDAnnot” on page 88.](#)

For information on adding a handler for a new annotation type, see [“Annotation Handlers” on page 162.](#)

Accessing the Info Dictionary

In addition to retrieving and setting values for the four standard fields in the PDF Info object, plug-ins can add, modify, query, and delete fields of their own.

For more information, see the **PDDocGetInfo** and **PDDocSetInfo** methods in “**PDDoc**” on page 91.

Adding Private Data To PDF Files

Plug-ins can add their own private data to PDF files, subject to the constraint that the data added must not interfere with the viewing of the PDF file by an Acrobat viewer that does not have the plug-in needed to interpret the private data.

For additional information, see [Chapter 9, “Cos Layer.”](#)

The Acrobat Support (AS) layer of the core API provides a variety of utility methods, including platform-independent memory allocation and fixed-point math utilities. In addition, it allows plug-ins to replace low-level file system routines used by Acrobat (including read, write, reopen, remove file, rename file, and other directory operations). This enables Acrobat to be used with other file systems, such as on-line systems.

Several AS methods return error codes rather than raising exceptions on errors. This is because these methods are called at a low level where exception handling would be inconvenient and expensive.

This chapter summarizes the AS objects and methods supported by the Acrobat core API. See the [Acrobat Core API Reference](#) for a detailed description of all methods.

This chapter also describes a set of platform-specific methods.

ASAtom

ASAtoms are hashed tokens that Acrobat uses in place of strings to optimize performance (it is much faster to compare **ASAtoms** than strings). **ASAtom** methods convert between strings and **ASAtoms**. Some of the **ASAtom** methods and the tasks they perform include:

ASAtomFromString	Converts a string to an ASAtom .
ASAtomGetString	Gets a string, given an ASAtom .

ASCab

ASCab objects (cabinets) can be used to store arbitrary key-value pairs. The keys are always null-terminated strings containing only low-ASCII alphanumeric characters. An **ASCab** owns all the non-scalar data inside it. That is, when a plug-in places a value inside a cabinet, the **ASCab** now manages the value and frees it when the key is destroyed. If, for example, a plug-in creates an **ASText** object and adds it as a value to an **ASCab**, that **ASText** object is no longer owned by the plug-in: it is owned by the **ASCab**. The **ASCab** destroys the **ASText** object when the object's associated key is removed or its value is overwritten.

ASCabs are used to stored data used by some of the new Acrobat 5.0 APIs including [AVCommand](#), [AVConversion](#), and batch security features (see “[New Security Features in Acrobat 5.0](#)” on page 176).

ASCab Method Naming

The **ASCab** naming conventions indicate how the **ASCab** methods work. The conventions are as follows:

- **Get** methods return a value. These objects are owned by the **ASCab** and should not be altered or destroyed by the caller of **Get**.
- **GetCopy** methods make a copy of the data; the **GetCopy** caller owns the resulting information and can do whatever it wants with it.
- **Detach** methods work the same way as **Get** methods, but the key is removed from the **ASCab** without destroying the associated value, which is passed back to the caller of **Detach**.

Any of the **ASCab** methods can take a compound name or string consisting of multiple keys, each of which is separated by a colon (:) character. This format is:

Grandparent:Parent:Child:Key

Acrobat burrows down through such a compound string until it reaches the most deeply nested cabinet.

Handling Pointers

Normally, pointers are treated like scalars (e.g. integers) in an **ASCab**: the **ASCab** passes the pointer value back and forth but does not own the data pointed to.

If, however, the pointer has an associated **destroyProc**, this is no longer the case. When the **destroyProc** is set, the **ASCab** reference counts pointers to track how many times the pointer is being referenced from any **ASCab**. For example, the reference count is incremented whenever the pointer is copied via [ASCabCopy](#). Detaching a pointer removes one reference to the pointer but does not destroy the associated data. The data is destroyed by a call to the **destroyProc** when the reference count is 0.

[ASCabDetachPointer](#) returns a separate value indicating whether the pointer can safely be destroyed or if it still is being referenced by other key/value pairs inside **ASCabs**.

ASCab Methods

ASCab methods include:

ASCabCopy	Copies all key-value pairs from one cabinet into another.
---------------------------	---

ASCabDestroy	Destroys a cabinet and all its key-value pairs.
ASCabDetachCab	Retrieves a cabinet stored as a key in another cabinet, and removes the key.
ASCabGetInt	Gets an integer value from a cabinet.
ASCabGetPointer	Gets a pointer value from a cabinet.
ASCabGetStringCopy	Gets a copy of a string in a cabinet.
ASCabNew	Creates a new, empty cabinet.
ASCabRemove	Removes a key from a cabinet, destroying the associated value.

ASCallback

Callbacks allow the Acrobat viewer to call functions in a plug-in. The core API provides macros to create and destroy callbacks. These include [ASCallbackCreateProto](#), [ASCallbackCreateReplacement](#), and [ASCallbackCreateNotification](#) (defined in `PICommon.h`) and [ASCallbackDestroy](#). These macros (which eventually call the macro [ASCallbackCreate](#)) allow compilers to perform type checking, eliminating an extremely common source of bugs.

NOTE: For these macros to perform type checking, you must `#define DEBUG 1` before compiling. Remember to `#define DEBUG 0` before compiling a shipping version of your plug-in.

It is sometimes necessary for a plug-in to call [ASCallbackCreate](#) directly; for example, when it is passing function pointers without [typedefs](#) to the Macintosh toolbox routines.

ASExtension

An [ASExtension](#) represents a specific plug-in. An unique [ASExtension](#) object is created for each plug-in when it is loaded. If the plug-in fails to initialize, the [ASExtension](#) will remain but is marked as dead. [ASFile](#) methods include:

ASEnumExtensions	Enumerates all ASExtensions .
----------------------------------	---

ASFile

The **ASFile** interface encapsulates Acrobat's access to file services. It provides a common interface for Acrobat viewers, applications, and plug-ins to access file system services on different platforms, and enables you to provide your own file storage implementations.

Three objects are defined by this interface:

- **ASFile** (described here)
- **ASFileSys**
- **ASPathName**

An **ASFile** is an opaque representation of an open file. It is similar to an **ASStream**, although it is a lower-level abstraction. An **ASFile** knows its **ASPathName** and **ASFileSys**. The Acrobat viewer creates one **ASFile** for each open file, and layers one or more **ASStream** objects on each **ASFile**. Logically, an **ASFile** is analogous to a file as used in the standard UNIX/POSIX low-level file I/O functions **open**, **close**, **read**, **write**, and so forth (located in `<fcntl.h>`), and an **ASStream** is analogous to a buffered file stream as used in standard C file I/O functions **fopen**, **fread**, **fwrite**, and so forth (located in `<stdio.h>`). **ASFile** methods include:

ASFileAcquirePathName	Acquires a file's path name
ASFileGetFileSys	Gets the file system on which a file resides
ASAtomGetString	Gets a string, given an ASAtom
ASFileRead	Reads data from a file

ASFileSys

An **ASFileSys** is a collection of functions that implement file system services, such as opening files, deleting files, reading data from a file, and writing data to a file. Each **ASFileSys** provides these services for one class of devices. The Acrobat viewer has a built-in **ASFileSys** that services the platform's native file system. The Windows Acrobat viewer includes an additional **ASFileSys** that services the OLE2 IStorage/IStream interfaces. Plug-ins may create additional **ASFileSys** objects to service other file systems. For example, a plug-in could implement an **ASFileSys** to access PDF files stored in a document database or to access PDF files across a serial link.

An **ASFileSys** is a structure containing pointers to callback procedures used by the **ASFileSys**, **ASFile**, and **ASPathName** methods. See “[File Systems](#)” on page 169 for more information on implementing an **ASFileSys**. The primary service of an **ASFileSys** is to provide clients with a readable and/or writable file object (**ASFile**)

corresponding to a particular named location (**ASPathName**) on a particular type of device. An **ASPathName** is specific to a given **ASFileSys**.

To allow Acrobat, another application, or a plug-in to open a file in your file system, your file system must implement the **pathFromDIPath** method, which is used by **ASFileSysDIPathFromPath**. This method converts a pathname specified in the Acrobat's device-independent pathname representation to a file system-specific **ASPathName**. The device-independent representation is a string that is equivalent to the URL path name, and is as defined in Section 7.4 in the *PDF Reference, second edition, version 1.3*. Example device-independent paths are:

```
/volumename/segment/segment/filename  
segment/filename  
.../.../segment/filename
```

To allow Acrobat, another application, or a plug-in to create new “open file” actions for your file system, your file system must implement the **diPathFromPath** method, which is used by **ASFileSysDIPathFromPath**. This method converts a pathname specified in your file system's **ASPathName** representation to Acrobat's device-independent path name representation.

NOTE: In some circumstances, the device-independent path name may be insufficient to uniquely identify a particular file in your file system. For example, on the Macintosh platform, it is possible to mount two different drives with the same name, and potentially have the same path name to two different files, each on one of the drives. If files cannot be uniquely identified by path name, you may also need to register a **PDFFileSpecHandler** for your file system. See “[PDFFileSpec](#)” on page 93 for discussion of **PDFFileSpecHandlers**.

To determine whether two **ASFileSys** instances are equal, your plug-in should compare their **ASFileSys** pointers. It cannot, however, compare two **ASPathNames** directly to determine whether or not they are equal; instead, it should convert them to device-independent pathnames using **ASFileSysDIPathFromPath**, and compare the resulting device-independent pathname strings. There are a few cases where comparing device-independent pathnames may result in incorrectly believing two path names specify the same file when they do not. For example, it is possible to have two identical path names that specify files on different Macintosh disks.

The **ASFileSys** methods include:

ASFileSysCreatePathName	Creates an ASPathName based on the input type and file specification.
ASFileSysDIPathFromPath	Converts a platform-independent pathname to a platform-dependent pathname.
ASGetDefaultFileSys	Gets a platform's default file system.

ASPathFromPlatformPath

Converts a platform-specific pathname to an **ASPathName**.

NOTE: Do not use this method on the Macintosh platform. Instead, call **ASFileSysCreatePathName**.

ASPathName

An **ASPathName** is a file system-specific named location for a particular type of device. It uses the **ASFileSys** structure pointers for callback. An **ASPathName** is specific to a given **ASFileSys**.

ASStm

An **ASStm** is a data stream that may be a buffer in memory, a file, or an arbitrary user-written procedure. You typically would use an **ASStm** to extract data from a PDF file. When writing or extracting data streams, the **ASStm** must be connected to a Cos stream (see “[CosStream](#)” on page 159).

ASStm methods include:

ASFileStmRdOpen

Creates an **ASStm** from which data can be read from a file.

ASMemStmRdOpen

Creates an **ASStm** from which data can be read from a memory buffer.

ASStmRead

Reads data into memory from an **ASStm**.

ASStmWrite

Writes data from memory to an **ASStm**. Can only be used to write to print stream when printing to a PostScript printer. It cannot be used for writing to files.

ASText

An **ASText** object holds encoded text. In Acrobat, encoded text can be specified in one of two ways:

- As a null-terminated string of multi-byte text coupled with a host encoding. The host encoding is platform-specific:
 - On the Macintosh, host encoding is specified as a script code.

- On Windows, host encoding is a **CHARSET** code.
- As an **ASUns16** string of Unicode text terminated by an **ASUns16 0** (two 0 bytes). The string can be in either BigEndian or HostEndian format.

Each of the formats described in [Table 4.1](#) can be mapped to one of the two cases outlined above.

TABLE 4.1 *Formats that can be mapped to encoded text*

Format	Description
Encoded	A multi-byte string terminated with a single null character and coupled with a specific host encoding indicator.
ScriptText	A multi-byte string terminated with a single null character and coupled with an ASScript code. (An ASScript is an enumeration of writing scripts.) This is merely another way of specifying the Encoded case; the ASScript code is converted to a host encoding using ASScriptToHostEncoding .
Unicode	A series of ASUns16 values containing Unicode values in big-endian format, terminated with a single ASUns16 0 . Unicode refers to straight Unicode without the 0xFE 0xFF prefix or language and country codes that can be encoded inside a PDF document.
PDText	A string of text pulled out of a PDF document. This is either a big-endian Unicode string pre-pended with the bytes 0xFE 0xFF or a string in PDF document encoding. In the Unicode case, this also may include language and country identifiers. ASText objects strip language and country information out of the PDText string and track them separately.

ASTexts also can be used to accomplish encoding conversions; your plug-in can request a string in any of the formats specified above.

In all cases the **ASText** code attempts to preserve all characters. For example, if your plug-in attempts to concatenate strings in separate host encoding, the implementation may convert both to Unicode and perform the concatenation in Unicode space.

When creating a new **ASText** object, or putting new data in an existing object, Acrobat will always copy the supplied data into the **ASText** object. The original data is yours to do with as you will (and release if necessary).

The size of **ASText** data always is specified in bytes, for example, the **len** argument to [ASTextFromSizedUnicode](#) specifies the number of bytes in the string rather than the number of Unicode characters.

Host encoding and Unicode strings are always terminated with a null character (which consists of one null byte for host-encoded strings and two null bytes for Unicode

strings). You cannot create a string with an embedded null character even using methods that take an explicit length parameter.

The **GetXXX** methods return pointers to data held by the **ASText** object. Your plug-in cannot free or manipulate this data directly. The **GetXXXCopy** methods return data that your plug-in can manipulate at will and is responsible for freeing.

ASText methods include:

ASTextCmp	Compares two ASText objects, ignoring language and country information.
ASTextFromEncoded	Creates a new text object from a null-terminated multi-byte string in the specified host encoding.
ASTextFromSizedEncoded	Creates a new text object from a multi-byte string in the specified host encoding and of the specified length.
ASTextFromSizedUnicode	Creates a new ASText from the specified Unicode string. This string is not expected to have 0xFE 0xFF prepended, or country/language identifiers.
ASTextGetCountry	Gets the country associated with a piece of text stored in an ASText object.
ASTextGetPDTTextCopy	Gets the text in a form suitable for storage in a PDF file.
ASTextNew	Creates a new ASText containing no text.
ASTextSetLanguage	Sets the language codes associated with an ASText .

Configuration

The **ASGetConfiguration** method allows a plug-in to determine the Acrobat viewer under which it is running. Because Acrobat Reader supports only a subset of the core API, it is vital that plug-ins use this method at start-up to ensure that the viewer currently running supports the API methods they need, including whether the viewer can save changes to files (Acrobat can always save changes).

Errors

Acrobat supports a mechanism for registering and using error codes. These error codes may be returned by methods, or (more commonly) used as exception codes

when raising exceptions. See [Chapter 12, “Handling Errors,”](#) for a description of exceptions and their handling.

Some of the representative error methods include:

ASGetExceptionErrorCode	Gets the code of the most recently raised exception. For convenience, your plug-in may use the ERRORCODE macro described in “Exception Handlers” on page 183.
ASRaise	Raises an exception.
ASRegisterErrorString	Registers a defined error and an associated descriptive string.

Fixed-point Math

These macros and methods support operations on fixed-point numbers. Acrobat uses 32-bit fixed numbers, with the least significant 16 bits of a fixed-point number representing the fractional part of the number. The operations supported include conversions between integers and fixed-point numbers, conversions between C strings and fixed-point numbers, math, rectangle utilities, and matrix operations. **ASFixed** is not a standard C type.

The core API includes some macros and some AS layer methods for making conversions.

Fixed-point Utility Macros

Some of the macros that convert between integer and fixed-point numbers, and that specify common fixed-point numbers are described here.

ASFixedRoundToInt32	Converts a fixed point number to an ASInt32 rounding the result.
ASFixedToFloat	Converts a fixed point number to a floating point number.
ASFixedTruncToInt32	Converts a fixed point number to an ASInt32 truncating the result.
ASFloatToFixed	Converts a floating point number to a fixed point number.
ASInt32ToFixed	Converts an ASInt32 to a fixed point number.

Fixed-point Mathematics Methods

The fixed-point mathematics methods are described here:

ASFixedDiv	Divides two fixed point numbers.
ASFixedMul	Multiplies two fixed point numbers.

Fixed-point Matrix Methods

The fixed-point matrix methods are described here.

ASFixedMatrixConcat	Multiplies two matrices.
ASFixedMatrixInvert	Inverts a matrix.
ASFixedMatrixTransformRect	Calculates the coordinates of a rectangle's corner points in another coordinate system.

In addition, the header file `ASExpT.h` contains a number of predefined constants for specific fixed-point numbers.

HFT Methods

HFTs are the mechanism through which plug-ins call methods in the Acrobat viewer or in other plug-ins. This capability enables plug-ins to override specific portions of Acrobat's functionality. For more information, see ["Host Function Tables" on page 35](#).

The AS group contains several methods for dealing with HFTs, including:

ASExtensionMgrGetHFT	Gets an HFT by name.
HFTNew	Creates an HFT.
HFTReplaceEntry	Replaces a replaceable method in an HFT.
HFTServerNew	Creates an HFT server.

Memory Allocation

The core API provides methods for allocating and managing memory. These should always be used instead of C functions such as `malloc` and `free`. Memory allocation methods include:

<code>ASmalloc</code>	Allocates a block of memory.
<code>ASfree</code>	Frees a block of memory.
<code>ASrealloc</code>	Reallocates (changes the size of) a block of memory.

Platform-specific Utilities

Macintosh

The core API includes Macintosh-specific methods for plug-ins. For details on all the Macintosh methods available, see “Macintosh-specific Methods” in the [Acrobat Core API Reference](#). Methods include:

<code>AVAppHandleAppleEvent</code>	Handles an Apple event.
<code>RectToAVRect</code>	Converts a Macintosh <code>Rect</code> to an <code>AVRect</code> .

UNIX

The core API also includes UNIX-specific utility methods, which are only available for plug-ins. These methods allow a plug-in to

- Find out about its environment
- Handle events
- Synchronize its operation with the window manager
- Read resources
- Write items into the preferences file
- Read the preferences file

UNIX methods include:

<code>UnixAppProcessEvent</code>	A wrapper function for <code>XtAppProcessEvent</code> .
<code>UnixSysGetCwd</code>	Gets the current working directory.

Windows

Windows-specific utility methods are only available for plug-ins. These methods allow a plug-in to:

- Manipulate modal and modeless dialogs
- Get the color palette
- Control the **AVAppIdle** timer

Windows methods include:

WinAppGetModalParent	Gets the appropriate parent for any modal dialogs created by a plug-in.
WinAppRegisterInterface	Registers a COM interface.
WinAppRegisterModelessDialog	Registers modeless dialogs with the viewer so that the dialog gets the correct messages.

The Acrobat Viewer (AV) layer of the core API (also known as AcroView) allows plug-ins to control the Acrobat viewer application and modify its user interface. Using the AV methods, a plug-in can add menus and menu items, add buttons to the toolbar, open and close files, display simple dialog boxes, and perform many other application-level tasks. Plug-ins must use AV layer methods to be accessible through the Acrobat viewer's user interface.

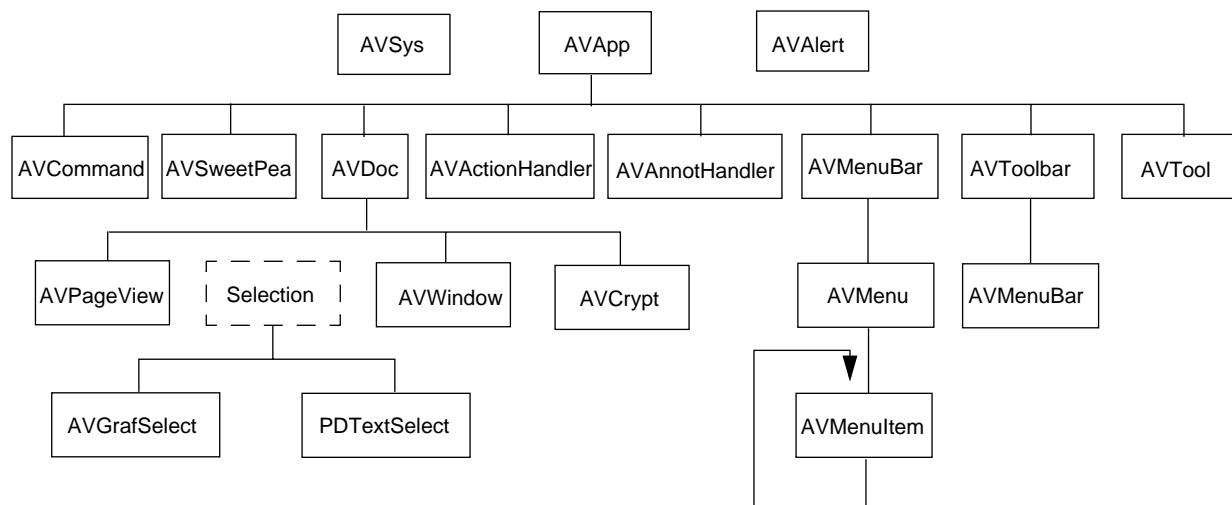
NOTE: AcroView methods are not available to the Adobe PDF Library.

The AV layer methods do not provide access to:

- Detailed internal structure of a PDF file (provided by the PD layer methods described in [Chapter 6, “Portable Document Layer”](#)).
- Editing page contents (provided by the PDFAEdit section of the core API, described in [Chapter 7, “PDFAEdit—Creating and Editing Page Content”](#)).
- Low-level disk structure of a file (provided by the Cos section of the core API, described in [Chapter 9, “Cos Layer”](#)).
- File I/O system (provided by the AS methods, described in [Chapter 4, “Acrobat Support”](#)).

The AV layer consists of the objects shown in [Figure 5.1, “Acrobat Viewer Objects”](#). The relationships in the figure are not strictly hierarchical, but are meant to indicate how objects are associated. For example, the **PDTTextSelect** object is included in the figure because of its close association with the AV selection methods.

FIGURE 5.1 Acrobat Viewer Objects



The following sections describe these objects and provide an overview of the methods each supports. See the [Acrobat Core API Reference](#) for a detailed description of each method.

General

General methods do not apply to a particular AV layer object. An example method is [AVDestInfoDestroy](#), which destroys a destination info object.

AVActionHandler

An **AVActionHandler** carries out an action. For instance, an action is what happens when a link or bookmark is clicked in the Acrobat viewer. See Section 7.5 in the [PDF Reference](#) for more information on actions.

When the Acrobat viewer executes an action, it looks for the action handler with a **type** matching that of the action it is trying to execute. The Acrobat viewer invokes the matching handler to perform the action. If no match is found, the Acrobat viewer ignores the user action.

Your plug-in can add new action handlers by using [AVAppRegisterActionHandler](#), expanding the range of action types to which the Acrobat viewer can respond. See “[Action Handlers](#)” on page 162 for further information on creating action handlers.

Your plug-in can use **AVActionHandlerGetProcs** to get a structure containing pointers to an action handler's methods. This method can be used to modify an existing action handler.

AVAlert

AVAlert provides platform-independent support for displaying simple dialog boxes. Methods include:

AVAlert	Displays a dialog box containing a string, icon, and up to three buttons.
AVAlertNote	Displays a dialog box containing a string and an OK button.

AVAnnotHandler

An **AVAnnotHandler** is responsible for creating, displaying, selecting, and deleting a particular type of annotation. There is one annotation handler for each annotation type. The Acrobat viewer contains two built-in annotation types (notes and links), and plug-ins can add new annotation handlers by using **AVAppRegisterAnnotHandler**. See “Annotation Handlers” on page 162 for details on creating new annotation types.

AVAnnotHandler has no methods of its own, but is instead accessed through **AVApp** methods.

AVApp

AVApp represents the Acrobat viewer. **AVApp** methods include:

AVAppGetActionHandlerByType	Gets the action handler whose type is specified.
AVAppGetActiveDoc	Gets the frontmost document view.
AVAppGetCancelProc	Gets the default procedure that is called to determine whether the user wants to cancel an operation.
AVAppGetDocProgressMonitor	Gets the default thermometer progress bar.

AVAppGetMenuBar	Gets the viewer application's menu bar.
AVAppRegisterCommandHandler	Registers an AVCommand handler to implement an AVCommand with the specified name. See “ AVCommand ” on page 72 and “ AVCommand Handlers” on page 163 for details.
AVAppRegisterForPageViewClicks	Registers a procedure to call each time a mouse click occurs.
AVAppRegisterFromPDFHandler	Registers an AVConversionFromPDFHandler to export from PDF to other file formats.
AVAppRegisterGlobalCommand	Registers an AVCommand in the global command list.
AVAppRegisterToolBarPosition	Sets the position of a toolbar. A toolbar can have separate positional attributes for internal and external views.
AVAppRegisterToPDFHandler	Registers an AVAppRegisterGlobalCommand to import other file formats.

AVCommand

An **AVCommand** represents an action that the user can perform on the current document or the current selection in the current document. Specifically, an **AVCommand** represents a command that can be added to a command sequence and executed either interactively or by means of batch processing. Commands can be executed with [AVCommandExecute](#).

Invoking AVCommands Programmatically

To invoke **AVCommands** programmatically using the **AVCommand** methods, a plug-in client must:

- Instantiate the command.
- Configure the command.
- Run the command.

Instantiating the AVCommand

To instantiate an **AVCommand**, the client must call the **AVCommandNew** method, providing the registered name of the command, for example

```
ASAtom cmdName;
AVCommand cmd;

cmdName = ASAtomFromString ( "MinimalCommand" );
cmd = AVCommandNew ( cmdName );
```

For details on registering **AVCommand** handlers, see “AVCommand Handlers” on page 163.

Configuring the Command

Prior to executing the **AVCommand**, the client can configure three categories of properties:

- Input parameters (required)
- Configuration parameters (optional - initialized to defaults)
- **AVCommand** parameters (optional - initialized to defaults)

Input Parameters. At minimum, the client must configure the **AVCommand**’s input parameters. The command must be provided with a **PDDoc** upon which to operate, as shown in this example.

```
PDDoc pdDoc; // Initialized elsewhere
// Create cab to hold input parameters and populate
ASCab inputs = ASCabNew();
ASCabPutPointer (inputs, kAVCommandKeyPDDoc, PDDoc, pdDoc, NULL);
// Set the input parameters and destroy the container ASCab
if (kAVCommandReady != AVCommandSetInputs (cmd, inputs)) {
    // Handle error
}
ASCabDestroy (inputs);
```

All other inputs are optional. See the description of **AVCommandSetInputs** in the [Acrobat Core API Reference](#), for details.

Configuration Parameters. Optionally the client can set configuration parameters. The default UI policy is for commands to be fully interactive. To invoke the command programmatically instead, the client can create an **ASCab** object and populate it with the appropriate parameters, for example,

```
// Create cab to hold config parameters and populate
ASCab config = ACabNew();
ASCabPutInt (config, "UIPolicy", kAVCommandUISilent);

if (kAVCommandReady != AVCommandSetConfig (cmd, config)) {
    // Handle error
}
ASCabDestroy (config);
```

AVCommand Parameters. An **AVCommand**'s parameter set is specific to each command. For example, the **Document Summary** command accepts values for five parameters: **Title**, **Subject**, **Author**, **Keywords**, **Binding**, and **LeaveAsIs**. As in the above example (see “[Configuration Parameters](#)” on page 73), a plug-in can create **ASCab**s to hold the appropriate parameters; then it can create empty **ASText** objects to hold the parameter values and put these values in the **ASCab**s. The following example uses this approach to set **Document Summary** title and subject values:

```
const char *docTitleValue = "Document Title";
const char *docSubjectValue = "Document Subject";

// Create cab to hold command parameters and populate

ASCab params = ASCabNew();
ASText text = ASTextNew();
ASTextSetEncoded (text, docTitleValue,
                  (ASHostEncoding)PDGetHostEncoding());
ASCabPutText (params, docTitleKey, text);
text = ASTextNew();
ASTextSetEncoded (text, docSubjectValue,
                  (ASHostEncoding)PDGetHostEncoding());
ASCabPutText (params, docSubjectKey, text);
...
ASCabDestroy(params);
```

Running the AVCommand

The client can use either of two methods to drive the command:

AVCommandExecute or **AVCommandWork**. **AVCommandExecute** is a wrapper method that repeatedly calls **AVCommandWork** until the command returns a status code other than **kAVCommandworking**.

AVCommand Methods

AVCommand methods include:

AVCommandCancel	Cancels the specified command.
AVCommandGetAVDoc	Gets the AVDoc from a command's inputs ASCab .
AVCommandGetInputs	Gets the input parameters of the specified command.
AVCommandGetParams	Gets the parameter set for the specified command.
AVCommandNew	Creates a new command of the specified type.
AVCommandSetParams	Sets the parameters for the specified command.
AVCommandShowDialog	Instructs the command to bring up its configuration dialog and gather parameter information from the user.

AVConversion

The **AVConversion** methods enable conversion from non-PDF file formats to PDF and vice versa. For information on using the **AVConversion** methods to create a file conversion handler, see “[File Format Conversion Handlers](#)” on page 166.

The **AVConversion** methods include:

AVConversionConvertToPDFWithHandler	Converts a PDF document to another file format using the specified handler.
AVConversionConvertToPDF	Converts the specified file to a PDF document using whatever converter is found.
AVConversionConvertToPDFWithHandler	Converts a file to a PDF document using the specified handler.

AVCrypt

AVCrypt methods implement the Acrobat viewer’s built-in dialogs for encryption control. They are present in the core API so that they can be used by other security handlers. The **AVCrypt** methods include:

AVAuthOpen	Determines if a user is authorized to open a document.
AVCryptDoStdSecurity	Displays a security dialog to the user, allowing the user to change a document’s print, edit, and copy permissions.
AVCryptGetPassword	Displays the standard dialog box that lets a user enter a password. The PDDocAuthorize or PDDocPermRequest method (see “ PDDoc ” on page 91) actually checks the password.

AVDoc

An **AVDoc** is a view of a PDF document in a window. There is one **AVDoc** per displayed document. Unlike a **PDDoc** (described in “[PDDoc](#)” on page 91), an **AVDoc** has a window associated with it. The **AVDoc** methods include:

AVDocClose	Closes a document.
AVDocDoSave	Replaceable method that allows plug-ins to implement their own save functionality.
AVDocFromPDDoc	Gets the AVDoc associated with a PDDoc .
AVDocPrintPages	Prints pages from a document without displaying a print dialog to the user.
AVDocSetViewMode	Shows bookmarks, thumbnails, neither, or uses full-screen mode.

AVGrafSelect

An **AVGrafSelect** is a graphics selection on a page. It is a rectangular region of a page that can be copied to the clipboard as a sampled image. After a plug-in creates an **AVGrafSelect**, it can use [AVDocSetSelection](#) to set the **AVGrafSelect** as the current selection and [AVDocShowSelection](#) to scroll it to a visible position in the window.

AVGrafSelect methods include:

AVGrafSelectCreate	Creates a graphics selection.
AVGrafSelectDestroy	Destroys a graphics selection.
AVGrafSelectGetBoundingRect	Gets a graphics selection’s bounding rectangle.

AVMenu

An **AVMenu** is a menu in the Acrobat viewer’s menubar. Plug-ins can create new menus, add menu items at any location in any menu, and remove menu items. Deleting an **AVMenu** removes it from the menubar (if it was attached) and deletes all the menu items it contains.

There is a special **AVMenu** with the title **Tools**. This menu (the **About Plug-ins** menu item and the **Plug-in Help** menu item) are always created when Acrobat is launched.

They are removed if and only if they are empty after every plug-in's initialization routines have been called.

Submenus (also called pullright menus) are **AVMenu** objects that are attached to an **AVMenuItem** instead of to the menubar.

Each menu has a title and a *language-independent* name. The title is the string that appears in the user interface, while the language-independent name is the same regardless of the language used in the user interface. Language-independent names allow a plug-in to locate the **File** menu without knowing, for example, that it is called **Fichier** in French and **Ablage** in German.

It is strongly encouraged that you begin your language-independent menu names with the plug-in name (separated by a colon) to avoid name collisions when more than one plug-in is present. For example, if my plug-in is named **myPlug**, it might add a menu whose name is **myPlug:Options**.

Your plug-in cannot directly remove a submenu. Instead, it must remove the **AVMenuItem** to which the submenu is attached.

The **AVMenu** methods include:

AVMenuAddMenuItem	Adds a menu item at a specified location in a menu.
AVMenuGetName	Gets the language-independent menu name.
AVMenuNew	Creates a new menu.
AVMenuRemove	Removes a menu from the menu bar.
AVMenuRelease	Releases a previously acquired menu.

AVMenubar

The **AVMenubar** is the viewer's menubar and contains a list of all menus. There is only one **AVMenubar**. Plug-ins can add new menus to, or remove any menu from, the menubar. The menubar can be hidden from the user's view. The **AVMenuBar** methods include:

AVMenubarAcquireMenuByName	Acquires the menu with the specified name.
AVMenubarAcquireMenuItemByName	Acquires the menu item with the specified name.
AVMenubarAddMenu	Adds a menu to the menubar.
AVMenubarHide	Hides the menubar.

AVMenuItem

An **AVMenuItem** is a menu item in a menu. It has attributes, including

- A name
- A keyboard shortcut
- A procedure to execute when the menu item is selected
- A procedure to compute whether the menu item is enabled
- A procedure to compute whether the menu item is check marked, and whether it has a submenu.

Menu items also may serve as separators between menu items. You are encouraged to position your plug-in menu items relative to a separator. This helps ensure that if a block of menu items is moved in a future version of Acrobat, your plug-in's menu items also are moved.

In Acrobat 4.0 and higher, plug-ins can be liberal in their use of separators. After initialization, Acrobat 4.0 and higher clean up by removing separators at the beginning or end of menus as well as removing duplicate separators.

A plug-in can simulate a user selecting a menu item by calling **AVMenuItemExecute**. If the menu item is disabled, **AVMenuItemExecute** returns without doing anything. **AVMenuItemExecute** works even when the menu item is not displayed (for example, if it has not been added to a menu, its menu is not displayed, or the menu bar is not visible). Plug-ins can set all attributes of menu items they create, but must not set the **Execute** procedure of the Acrobat viewer's built-in menu items.

Your plug-in can specify menu item names using either the names seen by a user, or language-independent names. The latter allows your plug-in to locate the **Print...** menu item without knowing, for example, that it is called **Imprimer...** in French and **Drucken...** in German.

You are strongly encouraged to begin your plug-in's language-independent menu item names with your plug-in's name (separated by a colon) to avoid name collisions when more than one plug-in is present. For example, if my plug-in is named **myPlug**, it might add menu items whose names are **myPlug:Open** and **myPlug:Find**.

The **AVMenuBar** methods include:

AVMenuItemExecute	Executes a menu item's ExecuteProc .
AVMenuItemGetName	Gets the language-independent name of the menu item.
AVMenuItemNew	Creates a new menu item.
AVMenuItemSetExecuteProc	Sets the procedure called when the menu item is selected by the user.

AVPageView

An **AVPageView** is the area of the Acrobat viewer's window that displays the contents of a document page. Every **AVDoc** has an **AVPageView** and vice versa. It contains references to the **PDDoc** and **PDFont** objects for the document being displayed. Plug-ins can control the size of the **AVPageView** through **AVWindowSetFrame** and **AVDocSetSplitterPosition**.

AVPageView has methods to display a page, select a zoom factor, scroll the page displayed inside, highlight one or more words, control screen redrawing, and traverse the history stack that records where users have been in a document.

In continuous page modes when more than one page may be displayed, **AVPageView** may not completely specify the view of the **AVDoc**. In these cases, your plug-in needs to call **AVPageViewSetPageNum** to set the page number that it wants. For instance, if your plug-in is getting an annotation's bounding rectangle with **AVPageViewGetAnnotRect**, it should call **AVPageViewSetPageNum** first, providing the annotation's page number. This ensures that your plug-in gets the **AVRect** on the page upon which the annotation appears.

Additional **AVPageView** methods include:

AVPageViewGetFirstVisiblePageNum	Returns the page number of the first page visible.
AVPageViewGetPage	Gets the PDPage for a page view.
AVPageViewPointToDevice	Transforms a points coordinates from user space to device space. See “ Understanding Coordinate Systems ” on page 28.
AVPageViewZoomTo	Sets the zoom factor.

AVSweetPea

The **AVSweetPea** methods are used to implement the Adobe Dialog Manager (ADM), a cross-platform API for creating and managing dialogs by Adobe applications. For details on how to use ADM for Acrobat dialogs, see [Using ADM in Acrobat](#). The **AVSweetPea** methods include:

AVSweetPeaGetBasicSuiteP	Accesses the basic ADM suite.
AVSweetPeaGetPluginRef	Gets a reference to the ADM plug-in itself (not the plug-in you are currently developing).
AVSweetPeaIsADMAvailable	Determines whether ADM is available.

AVSys

AVSys provides various system-wide utilities, including setting the cursor shape, getting the current cursor, and beeping. Methods include:

<code>AVSysAllocTimeStringFromTimeRec</code>	Gets a string representing the date and time.
<code>AVSysBeep</code>	Beeps.
<code>AVSysGetStandardCursor</code>	Gets the specified cursor.
<code>AVSysMouseIsStillDown</code>	Tests whether the mouse button is still being pressed.

AVTool

An **AVTool** is an object that can handle key presses and mouse clicks in the content region of an **AVPageView**. Tools do not handle mouse clicks in other parts of the viewer's window, such as in the bookmark pane. At any time, there is one *active tool*, which a plug-in can set using `AVAppSetActiveTool`.

Tools are often, but not always, set from toolbar buttons (see “[AVToolButton](#)” on [page 81](#)). Some buttons, such as **Zoom**, set an active tool; in this case, setting the active tool to one that drags out a rectangle, or adjusts the viewer's zoom level in response to user actions. Other buttons, such as the one that displays thumbnail images, do not change the active tool.

Use `AVAppRegisterTool` to add a new tool to the Acrobat viewer.

Additional **AVTool** methods include:

<code>AVToolGetType</code>	Gets a tool's type.
<code>AVToolIsPersistent</code>	Indicates whether a tool is persistent or is one shot.

AVToolBar

AVToolBar is the Acrobat viewer's toolbar (the palette of buttons). In Acrobat 4.0 and later, a plug-in can create *flyouts* that contain additional buttons and attach these flyouts to existing buttons.

Plug-ins can add buttons to and remove buttons from a toolbar, show or hide toolbars, and (Acrobat 5.0) create new toolbars. Because screen space is limited on many monitors, plug-ins should add as few buttons as possible to toolbars.

Buttons can be organized into groups of related buttons, with additional space between the groups. It is possible to implement a group in which only one button can be selected at a time. The logic of doing this is the plug-in's responsibility; the plug-in API does not provide any means to automatically relate one button's state to another button's state.

A plug-in adds buttons to a toolbar by specifying the relative position of the button (before or after) to an existing button.

Although there appear to be multiple toolbars in the Acrobat 4.0 and higher user interface, there is still only one **AVToolBar** containing all the buttons that are not on flyouts. A plug-in controls the toolbar upon which a button will appear by placing the button next to an existing one already in the appropriate location. The **AVToolBar** methods include:

AVToolBarAddButton	Adds a button to the toolbar.
AVToolBarGetButtonByName	Gets the button that has the specified name.
AVToolBarNewFlyout	Creates a new sub-toolbar for use as a tool button flyout.

AVToolBar

An **AVToolBar** is a button in the Acrobat viewer's toolbar. Like menu items, the procedure that executes when the button is clicked can be set by a plug-in. Although not required, there generally is a menu item corresponding to each button, allowing users to select a function using either the button or the menu item.

A plug-in can invoke a button as if a user clicked it. Buttons can be enabled (selectable) or disabled (grayed out), and can be marked (selected). Each button also has an icon that appears in the toolbar. **AVToolButtons** frequently, but not always, change the active tool (see “[AVTool](#) on page 80). For example, the button that selects the link tool changes the active tool; whereas, the button that goes to the last page of a document does not.

Normally, all tools are persistent and remain selected indefinitely. The **Option** key (Macintosh platform) or **Control** key (Windows) can be used to select a tool for one-shot use. Plug-ins should follow this convention to add buttons.

Separators between groups of buttons are themselves buttons, although they are neither selectable nor executable. Because they are buttons, however, they do have names, allowing other buttons to be positioned relative to them.

Plug-ins are encouraged to position their tool buttons relative to separators. Doing this increases the likelihood that tool buttons will be correctly placed if future versions of Acrobat move groups of toolbuttons around.

Acrobat 4.0 and higher cleans up separators. It ensures that separators don't appear back-to-back or at the beginning or end of the toolbar. Plug-ins can be liberal with separators in Acrobat 4.0 and higher versions.

You are strongly encouraged to begin your language-independent button names with the plug-in name (separated by a colon) to avoid name collisions when more than one plug-in is present. For example, if my plug-in is named `myPlug`, it might add a button whose name is `myPlug:LastFile`. For more information on plug-in naming, see Chapter 2, “Registering and Using Plug-in Prefixes,” in the *Acrobat Development Overview*.

The `AVToolButton` methods include:

<code>AVToolButtonExecute</code>	Invokes a button's execute procedure.
<code>AVToolButtonGetName</code>	Gets the name of a button.
<code>AVToolButtonNew</code>	Creates a new button.
<code>AVToolButtonRemove</code>	Removes (but does not destroy) a button from the toolbar.
<code>AVToolButtonSetFlyout</code>	Attaches a sub-toolbar to a tool button.

AVWindow

`AVWindow` provides methods for creating and managing windows. Plug-ins should use `AVWindow` methods for their own dialogs, floating palettes, and so forth, to ensure that those windows work well with the viewer; for example, that under Windows, they are hidden when the Acrobat viewer is iconified. Once the plug-in creates an `AVWindow`, it is free to use platform-dependent code to put whatever it wants in the window.

The Acrobat viewer uses the concept of a key window. The *key window* is the window that receives keyboard events. Only one window is the key window *at any time*. Windows can request to become the key window, or request that they no longer be the key window.

On the Macintosh platform, there is an essential distinction between a key window and an active window. A window is a key window if and only if it is the target of all keystrokes and menu selections. A window is the active window if mouse clicks in it are interpreted without requiring an initial activation click. This state is usually indicated (in modal and non-floating windows) with some highlighting in the title bar. Floating windows are inactive only if hidden.

NOTE: Plug-ins on the Macintosh platform should always use the core API methods to zoom, resize, or move windows. They should never use the toolbox routines (`ZoomWindow`, `SizeWindow`, `GrowWindow`, `MoveWindow`, and so forth) directly on `AVWindows`.

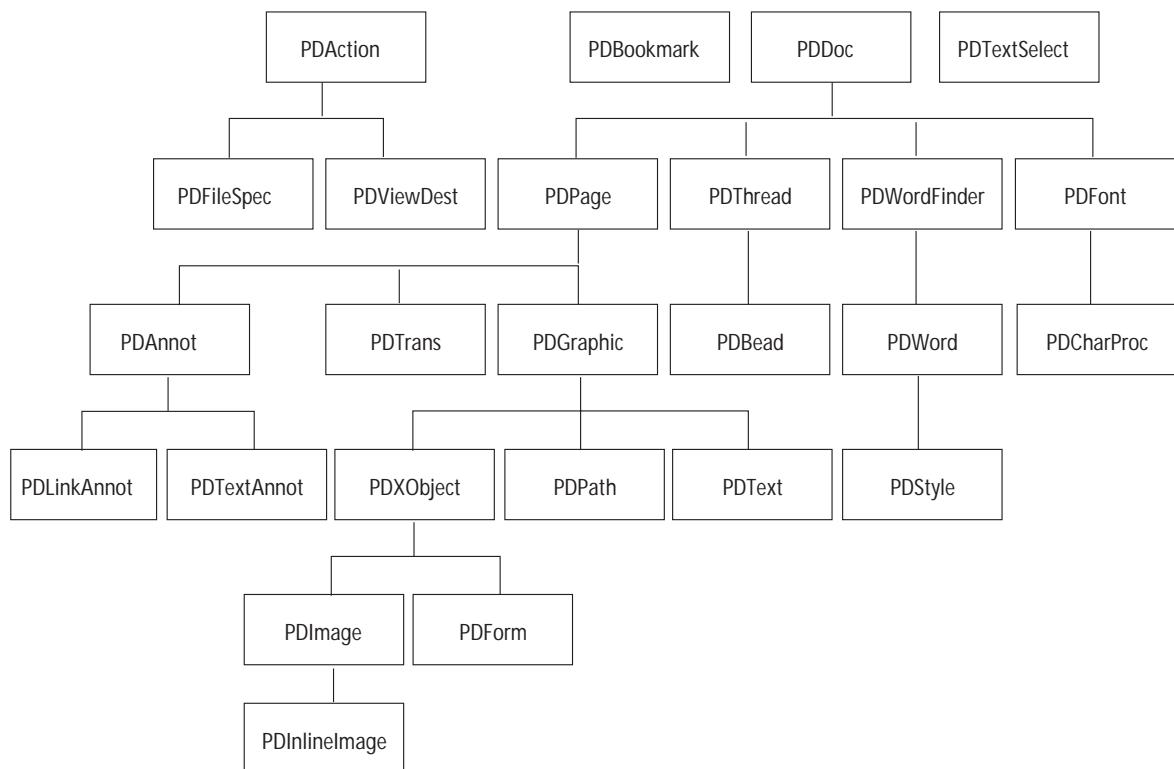
In addition, the Acrobat viewer reserves the **WRefCon** field in the **WindowRecord** structure for internal purposes. To attach client data to an **AVWindow**, a plug-in should use [AVWindowGetOwnerData](#) or [AVWindowSetOwnerData](#).

The **AVWindow** methods include:

AVWindowNew	Creates a new window.
AVWindowNewFromPlatformThing	Creates a new window from a platform-native window pointer or handle.
AVWindowHide	Hides a window.
AVWindowDrawNow	Updates a window.

The Portable Document (PD) layer of the core API (also called PDModel) is a collection of object methods enabling plug-ins to access and manipulate most data in a PDF file. [Figure 6.1](#) shows the objects in the PD layer.

FIGURE 6.1 *PD Layer Objects*



NOTE: Because of its close association to the **AVDoc** selection mechanism, the **PDTTextSelect** object is shown in [Figure 5.1, “Acrobat Viewer Objects.”](#)

Because many PD layer objects are based on PDF objects, it's important to understand PDF file structure. See the [PDF Reference](#) for details on PDF files.

PD layer methods perform the bookeeping that ensures any file written is a valid PDF file. In addition, they take care of navigating much of the PDF file structure, such as traversing the pages tree to get a specific page.

If you need lower-level access to the data in a PDF file, use Cos layer methods (see [Chapter 9, “Cos Layer”](#)). To control the Acrobat viewer application itself, use AV layer methods (see [Chapter 5, “Acrobat Viewer Layer”](#)). To modify page contents,

such as text, use PDFEdit methods, described in [Chapter 7, “PDFEdit—Creating and Editing Page Content.”](#)

The following sections describe each PD layer object and provide an overview of each object’s methods. See the [Acrobat Core API Reference](#) for detailed descriptions of the methods.

General PD Layer Methods

Some methods are general PD layer methods that are not associated with a specific object. They include:

<code>PDEnumDocs</code>	Enumerates the currently open PDDocs.
<code>PDGetHostEncoding</code>	Indicates whether a system is Roman or CJK-capable, i.e., capable of handling multibyte character sets, such as Chinese, Japanese, or Korean..
<code>PDGetPDFDocEncoding</code>	Gets a list describing the differences between host encoding and a predefined encoding <code>PDFDocEncoding</code> . See Appendix C of the PDF Reference for a description of <code>PDFDocEncoding</code> .
<code>PDXlateToHostEx</code>	Translates a string from <code>PDFDocEncoding</code> to host encoding, allowing for multibyte characters.
<code>PDXlateToPDFDocEncEx</code>	Translates a string from host encoding to <code>PDFDocEncoding</code> or Unicode.

Metadata

Metadata is information that describes document content or use. The PDF file format has always provided the `Info` dictionary, which contains metadata that applies to an entire document, with nine standard properties defined (including creation and modification date, title, and author).

New Metadata Features in PDF 1.4

PDF 1.4 introduces an enhanced way of representing metadata. It has the following features:

- In addition to document-level metadata, there can be metadata describing individual components of a PDF document, such as pages or images.

- Metadata properties and values are represented in the World Wide Web Consortium's Resource Definition Format (RDF), which is a standard metadata format based on XML.
- The set of standard metadata items is organized into *schemas*, each of which represents a set of properties from a particular application or industry standard. The schemas, as well as the physical representation, are defined by an Adobe standard (provisionally referred to as XAP).

See [PDF: Changes From Version 1.3 to 1.4](#) for details about the use of metadata in PDF 1.4.

Metadata APIs in Acrobat 5.0

Acrobat 5.0 provides a set of methods for accessing metadata. They are summarized here (note that there are PDFEdit and Cos layer calls in addition to PD layer methods):

PDDocGetXAPMetadata	Gets the metadata of a document and returns it as a newly allocated ASText object.
PDDocSetXAPMetadata	Sets the metadata of a document.
PDEContainerGetXAPMetadata	Gets the metadata associated with a document element
PDEContainerSetXAPMetadata	Sets the metadata for a document element
PDDocCalculateImplicitMetadata	Broadcasts a notification to request that plug-ins calculate and set implicit metadata items. <i>Implicit metadata</i> is metadata which depends on the state of a document and must be calculated, rather than being stored explicitly.
CosDictGetXAPMetadata	Gets the metadata associated with a dictionary or stream Cos object.
CosDictSetXAPMetadata	Sets the metadata for a dictionary or stream Cos object.

The Acrobat SDK contains several samples dealing with metadata. See the [Guide to SDK Samples](#) for information.

PDAction

Actions are tasks that the Acrobat viewer performs when a user clicks on a link or a bookmark. Acrobat viewers allow a document to execute an action automatically when a document is opened.

Action types include

- Going to another view within the same document
- Going to a specified view in another PDF file
- Launching an arbitrary file
- Playing a sound
- Resolving a URL

See Section 7.5 in the [PDF Reference](#) for more information on actions.

You can add custom action types to your plug-in by creating new action handlers (see “[Action Handlers](#)” on page 162) that are responsible for interpreting an action’s data and carrying out the action. **PDAction** methods include:

PDActionGetDest	Gets an action’s destination view.
PDActionGetSubtype	Gets an action’s subtype.
PDActionNew	Creates a new action.

PDAnnot

This is the abstract superclass for all annotations (see Section 7.4, “Annotations,” in the [PDF Reference](#)). Acrobat viewers have two built-in annotation classes:

PDTTextAnnot and **PDLINKAnnot**. Plug-ins add movie, Widget (form field), and other annotation types. You can define new annotation subtypes by creating new annotation handlers (see “[Annotation Handlers](#)” on page 162). There are no objects of type **PDAnnot**, but you can use **PDAnnot** methods on any subclass of **PDAnnot**.

The Acrobat SDK provides three useful macros to cast among **PDAnnot** and its text annotation and link annotation subclasses (see **PDExpT.h**). These are:

- [CastToPDAnnot](#)
- [CastToPDTTextAnnot](#)
- [CastToPDLINKAnnot](#)

The **PDAnnot** methods include:

PDAnnotGetRect	Gets an annotation’s size and location.
--------------------------------	---

PDAnotGetSubtype Gets an annotation's subtype.

PDAnotIsValid Tests whether an annotation is valid.

PDBead

A *bead* is a single rectangle in an article thread. An *article thread* represents a sequence of physically discontiguous but logically related items in a document (for example, a news story that starts on one page of a newsletter and runs onto one or more nonconsecutive pages). See Section 7.3.2, “Articles,” in the [PDF Reference](#) for more information on article threads and beads.

A bead remains valid as long as a thread is “current and active.” When traversing the beads in a thread using [PDBeadAcquirePage](#) or [PDBeadGetPrev](#), you can use [PDBeadEqual](#) to determine the wraparound point (end of the list).

Additional [PDBead](#) methods include:

PDBeadGetRect Gets a bead's bounding rectangle.

PDBeadGetThread Gets the thread containing the specified bead.

PDBookmark

A bookmark corresponds to an outline object in a PDF document (see Section 7.2.2, “Document Outline,” in the [PDF Reference](#)). A document outline allows the user to navigate interactively from one part of the document to another. An outline consists of a tree-structured hierarchy of bookmarks, which display the document’s structure to the user. Each bookmark has:

- A title that appears on screen
- An action that specifies what happens when the user clicks on the bookmark

Bookmarks can either be created interactively by the user through the Acrobat viewer’s user interface or can be generated programmatically. The typical action for a user-created bookmark is to move to another location in the current document, although any action (see “[PDAction](#)” on page 88) can be specified.

Each bookmark in the bookmark tree structure has zero or more *children* that appear indented on screen, and zero or more *siblings* that appear at the same indentation level. All bookmarks except the bookmark at the top level of the hierarchy have one *parent*, the bookmark under which it is indented. A bookmark is said to be *open* if its children are visible on screen, and *closed* if they are not.

A plug-in can get or set:

- The open attribute of a bookmark
- The text used for the bookmark's title
- The action that is invoked when the bookmark is selected

PDBookmark methods include the following:

PDBookmarkAddNewChild	Adds a new child to a bookmark.
PDBookmarkAddNewSibling	Adds a new sibling to a bookmark.
PDBookmarkDestroy	Destroys a bookmark and all of its children.
PDBookmarkFromCosObj	Converts an appropriate Cos object to a bookmark.
PDBookmarkGetAction	Gets a bookmark's action.
PDBookmarkGetCosObj	Gets the Cos object associated with a bookmark
PDBookmarkGetParent	Gets a bookmark's parent bookmark.
PDBookmarkSetOpen	Opens or closes a bookmark.
PDBookmarkSetTitle	Sets a bookmark's title.

PDCharProc

A **PDCharProc** is a character procedure, a stream of graphic operators (see “[PDGraphic](#)” on page 97) that draw a particular glyph of a Type 3 PostScript font.

A *glyph* is the visual representation of a character, part of a character, or even multiple characters. For example, a glyph can be a picture of the letter **A**, or it can be an accent mark, such as grave (`), or it can be a picture of multiple characters such as the ligature **fl**, which represents the letters **f** and **l**. Glyphs can also be used to represent arbitrary symbols, such as in the font ITC Zapf Dingbats®. Every glyph has a name in a Type 1, multiple master Type 1, or Type 3 font. In most TrueType fonts, glyphs are assigned names. In some TrueType fonts, the glyph names are implicit.

For information on Type 3 fonts, see Section 5.5.4 in the [PDF Reference](#).

To determine the sequence of graphics operations used to draw one or more glyphs in a Type 3 font, use **PDFontEnumCharProcs** to enumerate the glyphs in the font. Then use **PDCharProcEnum** to enumerate the graphic operators in each glyph of interest.

PDDoc

A **PDDoc** object represents a PDF document. There is a correspondence between a **PDDoc** and an **ASFfile**. Also, every **AVDoc** has an associated **PDDoc**, although a **PDDoc** may not be associated with an **AVDoc**.

NOTE: An **ASFfile** may have zero or more underlying files, so a PDF file does not always correspond to a single disk file. For example, an **ASFfile** may provide access to PDF data in a database.

A plug-in may create a new document or open a document using an **ASFfileSys** and an **ASPathName**. These frequently provide access to disk files, but could also provide access to PDF files by other methods, such as via a modem line. Because PD layer objects do not have a concept of an “active document”, or even of a user, getting the **PDDoc** of a document opened by the user requires calls to AV layer objects (see [Chapter 5, “Acrobat Viewer Layer”](#)).

Each PDF document contains, among other things:

- A tree of pages (**PDPage**)
- (Optionally) trees of bookmarks and articles
- (Optionally) information and security dictionaries

These objects correspond to **CosObj** objects in the catalog of a **CosDoc** (see [Chapter 9, “Cos Layer”](#)). However, they also have PD layer equivalents which are accessible directly through **PDDoc** methods. Other objects in the catalog of a PDF file may require Cos methods to access.

When you merge a PDF file containing form fields that have appearances, those appearances and forms data are merged along with all the other page contents. If you merge a file that has forms data into another file that has forms data, name conflicts are resolved (in the same way the Acrobat Forms plug-in resolves these conflicts).

NOTE: For PDF files with forms data, when inserting pages from another file using **PDDocInsertPages**, do not use the **PDIInsertAll** flag. Using this flag wipes out the previous forms data and replaces it with the information from the file being inserted.

Querying PDDoc Permissions

[Chapter 11, “Document Security,”](#) describes Acrobat’s security features.

With Acrobat 5.0 and higher, plug-ins can query the permissions on a **PDDoc** to a finer granularity than in previous Acrobat releases. Plug-ins can query specific **PDDoc** objects and for specific operations authorized to be performed on those objects. At the **PDDoc** level, for example, plug-ins can query whether printing the document is fully allowed, allowed only at a low resolution, or not allowed under any circumstances. A plug-in can request the applicable operations authorized for any the following objects:

- Document
- Page
- Link
- Bookmark
- Thumbnail
- Annotation
- Form
- Signature

To obtain the permissions, a plug-in can call the [PDDocPermRequest](#) method (which replaces [PDDocGetPermissions](#) used with earlier Acrobat versions). The plug in can request, for example, whether a particular operation can be performed on a particular object (from the list above) for a specified [PDDoc](#). The plug-in may, for example, request whether permissions allow a rotating operation on a page object in the [PDDoc](#).

For a list of all the operations ([PDPermReqOprs](#)) that each object ([PDPermReqObj](#)) supports (and a plug-in can request using [PDDocPermRequest](#)), see the [PDPermReqOpr](#) and [PDPermReqObj](#) enumerations in the [Acrobat Core API Reference](#).

New callbacks have been added to the security handler structure [PDCryptHandler](#) to support the finer granularity of permissions that plug-ins can query.

PDDoc Methods

PDDoc methods include:

PDDocAuthorize	Adds permissions to the specified document, if permitted.
PDDocClose	Closes an open document.
PDDocCreate	Creates a new document.
PDDocCreatePage	Creates a new page.
PDDocCreateStructTreeRoot	Creates a new StructTreeRoot element.
PDDocCreateWordFinderUCS	Creates a word finder (see “ PDWordFinder ” on page 106) for extracting text in UCS format from a PDF file.
PDDocGetInfo	Gets a value from a document’s Info dictionary.
PDDocInsertPages	Inserts pages from another document.

PDDocGetNumPages	Gets the number of pages in the document.
PDDocGetWordFinder	Gets the word finder associated with a document.
PDDocOpen	Opens a PDDoc from an ASFileSys and an ASPathName .
PDDocPermRequest	(Acrobat 5.0 and higher) Causes Acrobat to call the document's security handler via PDCryptAuthorizeExProc requesting whether the operation is allowed on the object. Replaces the PDDocAuthorize and PDDocGetPermission methods.
PDDocSave	Saves a document.
PDDocSetInfo	Sets a value in a document's Info dictionary.

PDFFileSpec

A **PDFFileSpec** corresponds to the PDF file specification object (see Section 3.10, “File Specifications,” in the *PDF Reference*). It is used to specify a file in an action (see “**PDACTION**” on page 88). A file specification in a PDF file can take two forms:

- A single platform-independent pathname
- A data structure containing one or more alternative ways to locate the file on different platforms

PDFFileSpecs can be created from **ASPathNames** or from **Cos** objects. Methods are also provided to get **ASPathNames** and device-independent pathnames from **PDFFileSpecs**. The **PDFFileSpec** methods include:

PDFFileSpecAcquireASPath	Acquires an ASPathName for the specified file specification and relative path.
PDFFileSpecGetDIPath	Gets the device-independent pathname from a file specification.
PDFFileSpecGetDoc	Gets the PDDoc that contains the file specification.
PDFFileSpecGetFileSysName	Gets the name of the file system to which the PDFFileSpec belongs.

PDFFont

A **PDFFont** is a font that is used to draw text on a page. It corresponds to a font resource in a PDF file (see Section 5, “Fonts,” in the [PDF Reference](#)).

Plug-ins can get a list of **PDFFonts** used on a **PDPage** or a range of **PDPages**. More than one **PDPage** may reference the same **PDFFont** object.

A **PDFFont** has a number of attributes whose values can be read or set, including an array of widths, the character encoding, and the font’s resource name.

In general, a **PDFFont** refers to a base font and an encoding. The base font is specified by the font name and the subtype (typically Type 0, Type 1, MMTyp1, Type 3, or TrueType). This combination of base font and encoding is commonly referred to as a *font instance*.

In single-byte character systems, an encoding specifies a mapping from an 8-bit index, often called a *codepoint*, to a glyph.

Type 0 fonts support single-byte or multibyte encodings and can refer to one or more *descendant fonts*. These fonts are analogous to the Type 0 or composite fonts supported by Level 2 PostScript interpreters. However, PDF Type 0 fonts only support character encodings defined by a *character map* (CMap). The CMap defines the encoding for a Type 0 font. It specifies the mappings between character codes and the glyphs in the descendant fonts. For more information on Type 0 fonts, see Section 5.6.5, “Type 0 Font Dictionaries,” in the [PDF Reference](#). See Section 5.6.4, “CMaps,” for information on CMAs.

Type 0 fonts may have a CIDFont as a descendant. A CIDFont is designed to contain a large number of glyph procedures and is used for languages such as Chinese, Japanese, or Korean. Instead of being accessed by a name, each glyph procedure is accessed by an integer known as a character identifier or CID. Instead of a font encoding, CIDFonts use a CMap to define the mapping from character codes to a font number and a character selector. For more information on CIDFonts, see the following sections in the [PDF Reference](#):

- Section 5.6.1, “CID-Keyed Fonts Overview”
- Section 5.6.2, “CIDSysInfo Dictionaries”
- Section 5.6.3, “CIDFonts”

To access documents on CIDFonts, see the [Adobe Solutions Network Web site](#).

For general information on CID Fonts, refer to these technical notes:

Technical Note #	Title
5092	<i>CID-Keyed Font Technology Overview</i>
5014	<i>Adobe CMap and CIDFont Files Specification</i>

For information on specific CID fonts, see these technical notes:

Technical Note #	Title
5078	<i>Adobe-Japan1-2 Character Collection for CID-Keyed Fonts</i>
5079	<i>Adobe-GB1-0 Character Collection for CID-Keyed Fonts</i>
5080	<i>Adobe-CNS1-0 Character Collection for CID-Keyed Fonts</i>
5093	<i>Adobe-Korea1-0 Character Collection for CID-Keyed Fonts</i>
5094	<i>Adobe CJK Character Collections and CMaps for CID-Keyed Fonts</i>
5097	<i>Adobe-Japan2-0 Character Collection for CID-Keyed Fonts</i>
5174	<i>CID-Keyed Font Installation for PostScript File Systems</i>

Each base font contains a fixed set of glyphs. There are some common sets of glyph names, and these sets are typically called *charsets*. Acrobat viewers take advantage of the most common charset to enable font substitution. This charset is called the Adobe Standard Roman Character Set (see Appendix E in the *PostScript Language Reference, third edition*). If the Acrobat viewer encounters a font with this charset, it knows that it can represent all of the glyphs in the font using font substitution. Other common charsets are the Adobe Expert and Expert Subset charsets, and the Symbol charset. Most decorative fonts, such as Carta™ and Wingdings, have custom charsets.

Given a base font and its charset, multiple encodings are possible. For example, one encoding for a font could specify that the glyph for the letter 'A' appears at codepoint 65. A different encoding could specify that 'A' appears at both codepoint 65 and at codepoint 97. If text were rendered using the second encoding using the text string "a is always A", it would appear as "A is always A" using a font such as Times™. Encodings allow glyphs to be reordered to the most convenient order for an application or operating system.

Every font has a default encoding, commonly called its *built-in encoding*. In PDF, shortcuts are taken when specifying an encoding in order to minimize document size. If a font instance uses the built-in encoding, no encoding information is written into the PDF document. If a font has a different encoding, only those codepoints for which the encoding differs from the built-in encoding are recorded in the PDF file. This information is called a *difference encoding*; it describes the difference between the built-in encoding and the current encoding.

For non-Roman systems, the font encoding may be a variety of encodings, which are defined by a CMap. See Section 5.6.4, "CMaps," in the [PDF Reference](#) for a list of predefined CMaps, such as SHIFT-JIS for Japanese.

A *host encoding* is a platform-dependent encoding for the host machine's base font. For non-UNIX Roman systems, it is MacRomanEncoding on the Macintosh platform and WinAnsiEncoding on Windows. For UNIX (except HP-UX) Roman systems, it is ISO8859-1 (ISO Latin-1); For HP-UX, it is HP-ROMAN8. See Appendix D, "Character Sets and Encoding," in the [PDF Reference](#) for descriptions of MacRomanEncoding and WinAnsiEncoding. These encodings specify a mapping from codepoint to glyph name for fonts that use the Adobe Standard Roman Character Set on the Macintosh and Windows platforms.

Across PDF documents—or even within a single PDF document—the same base font can be used with more than one encoding. This allows documents from different platforms to be combined without losing information. Therefore, it is not uncommon to see a document that contains two instances of Helvetica™, one using **MacRoman** encoding and another using **WinAnsi** encoding. See Appendix D, "Character Sets and Encodings," in the [PDF Reference](#) for descriptions of MacRomanEncoding and WinAnsiEncoding. For non-Roman systems, the host encoding may be a variety of encodings, which are defined by a CMap.

Type 3 fonts do not have the ability to provide a base font with more than one encoding. For each Type 3 font, there is only one encoding. This encoding is completely specified in the PDF file; there are no shortcuts as there are for other fonts.

See Section 5.7, "Font Descriptors," in the [PDF Reference](#) for a discussion of font descriptors. Methods are provided to create and destroy fonts, as well as to access the information in the font's descriptor.

The **PDFont** methods include the following:

PDFontGetName	Gets the PostScript name for a Type 1 or Type 3 font, and the "styled" name for a TrueType font.
PDFontGetSubtype	Gets the font's subtype.
PDFontGetWidths	Gets a font's character widths.

PDForm

A **PDForm** is a self-contained set of graphics operators that is used when a particular graphic is drawn more than once in a document. It corresponds to a form resource (see Section 4.9, "Form XObjects," in the [PDF Reference](#)). **PDForm** objects inherit from the **PDXObject** class; you can use any **PDXObject** methods on a **PDForm**.

NOTE: A **PDForm** does not correspond to Acrobat's interactive forms. See the Acrobat SDK document, [Acrobat Forms API Reference](#), for information on the Acrobat Forms plug-in API methods.

The **PDForm** methods include the following:

PDFormEnumPaintProc	Enumerates a form's drawing operations.
PDFormGetBBox	Gets a form's bounding box.
PDFormGetMatrix	Gets a form's transformation matrix.

PDGraphic

PDGraphic is the abstract superclass for all graphic objects that comprise page, charproc, and **PDForm** descriptions (see Chapter 4, “Graphics,” in the [PDF Reference](#)). There are no objects of type **PDGraphic**, but its methods can be used by any graphic object. There are three types of graphic objects: **PDPPath**, **PDTText**, and **PDIInlineImage**. In addition to these three objects, there are also operators in the content stream. These operators are: **Save**, **Restore**, references to **XObjects** (forms and image resources), and for Type 3 font descriptions only, **charwidth** and **cachedevice**. Access to these objects and operators is via **PDPageEnumContents**, **PDFormEnumPaintProc**, or **PDCharProcEnum**.

The **PDGraphic** methods include the following:

PDGraphicGetBBox	Gets a graphic's bounding box.
PDGraphicGetCurrentMatrix	Gets the current transformation matrix in effect for a graphic object.
PDGraphicGetState	Gets the graphics state associated with a graphic object.

Many of the methods provide access to parameters of the graphics state. For a discussion of the graphics state and its parameters, see Section 4.3, “Graphics State,” in the [PDF Reference](#).

PDIImage

A **PDIImage** is a sampled image or image mask, and corresponds to a PDF Image resource (see “Stencil Masking” in Section 4.8, “Images,” in the [PDF Reference](#)). You can use any **PDXObject** method on a **PDIImage**. The **PDIImage** methods include the following:

PDIImageGetAttrs	Gets the attributes of an image.
PDIImageSelectAlternate	Selects an alternate image to use.

PDIInlineImage

A **PDIInlineImage** is an image whose data is stored in the page description's contents stream instead of being stored as an image resource (see [PDIImage](#)). **PDIInlineImage** is a subclass of **PDGraphic** and corresponds to the PDF inline image operator (see Section 4.8.6, "In-Line Images," in the [PDF Reference](#)).

Inline images generally are used for images with small amounts of data (up to several kilobytes), while image resources are used for large images. The reason for this is that there is a tradeoff between the time needed to access an image resource and the time saved by not having to parse inline image data when **display large images** is disabled in the Acrobat viewer. For small images, the time needed to access an image resource is large compared to the time needed to parse the image data; the opposite is true for large images.

The **PDIInlineImage** methods include the following:

PDIInlineImageGetAttrs	Gets an inline image's attributes.
PDIInlineImageGetData	Gets the image data for an inline image.

PDLinkAnnot

A **PDLinkAnnot** corresponds to a link annotation (see Sections 7.4.5, "Annotation Types," in the [PDF Reference](#)). You can use any **PDAnnot** method on a **PDLinkAnnot**.

Plug-ins can get and set:

- The bounding rectangle and color, using [PDAnnot](#) methods
- The action that occurs when the link is activated, using [PDLinkAnnot](#) methods
- The link's border, using [PDLinkAnnot](#) methods

Plug-ins can create new link annotations and delete existing ones, using the [PDPPage](#) methods.

The following are useful macros for casting among **PDAnnot** and its text annotation and link annotation subclasses:

- [CastToPDAnnot](#)
- [CastToPDTTextAnnot](#)
- [CastToPDLinkAnnot](#)

The **PDLinkAnnot** methods include:

PDLinkAnnotGetAction	Gets a link annotation's action.
--------------------------------------	----------------------------------

PDLinkAnnotSetBorder

Sets a link annotation's border.

PDNameTree

A **PDNameTree** is used to map Cos strings to Cos objects, just as a Cos dictionary is used to map Cos names to Cos objects. However, a name tree can have many more entries than a Cos dictionary. You can create a **PDNameTree** and locate it where appropriate (perhaps under a page, but most often right under the catalog). A **PDNameTree** is used to store the named destination information.

Name trees use Cos-style strings, which may use Unicode encoding, rather than null-terminated C strings. Unicode encoding may contain bytes with zeroes in them (the high bytes of ASCII characters).

The **PDNameTree** methods include:

PDNameTreeGet Retrieves an object from the name tree.

PDNameTreeNew Creates a new name tree in the document.

PDNumTree

A **PDNumTree** is used to map integers to arbitrary Cos objects just as a Cos dictionary is used to map Cos names to Cos objects. However, a number tree can have many more entries than a Cos dictionary. The **PDNumTree** methods include the following:

PDNumTreeFromCosObj Creates a type cast of a **CosObj** to a number tree.

PDNumTreePut Puts a new entry in the number tree. If an entry with this number is already in the tree, it is replaced.

PDPage

A **PDPage** is a page in a document, corresponding to the PDF Page object (see “Page Objects” in Section 3.6.2, “Page Tree,” in the *PDF Reference*). Among other associated objects, a page contains:

- A series of objects representing the objects drawn on the page (**PDGraphic**)
- A list of resources used in drawing the page
- Annotations (which are subclasses of **PDAnot**)

- An optional thumbnail image of the page
- Beads used in any articles that occur on the page

PDPage methods include:

PDPageAddAnnot	Adds an annotation to a page.
PDPageDrawContentsToWindow	Draws the contents of a page into a user-supplied window.
PDPageGetAnnot	Gets an annotation from a page.
PDPageGetBBox	Gets the bounding box for a page.
PDPageGetDoc	Gets the document containing a page.
PDPageGetNumber	Gets a page's number
PDPageGetNumAnnots	Gets the number of annotations on a page.
PDPageHasTransparency	Checks whether a page has any transparency features.

PDPageLabel

A **PDPageLabel** represents a page label. These labels allow non-sequential page numbering or the addition of arbitrary labels for a page (such as the inclusion of Roman numerals at the beginning of a book). A **PDPageLabel** specifies:

- The numbering style to use (for example, uppercase or lowercase Roman, decimal, and so forth)
- The starting number for the first page
- An arbitrary prefix to be added to each number (for example, "A-" to generate "A-1", "A-2", "A-3", and so forth)

PDPageLabel methods include:

PDPageLabelEqual	Compares two page labels to see if they are equivalent.
PDPageLabelGetStart	Gets the starting page number for the page label specified.

PDPPath

A **PDPPath** is a graphic object (a subclass of **PDGraphic**) representing a path in a page description. Paths are arbitrary shapes made of straight lines, rectangles, and cubic curves. Path objects may be filled or stroked, and they can serve as a clipping path. For details, see the following sections in the [PDF Reference](#):

- Section 4.1, “Graphic Objects”
- Section 4.4, “Path Construction and Painting”

PDPPath methods include:

PDPPathEnum	Enumerates a path’s operators, calling one of several user-supplied callbacks for each.
PDPPathGetPaintOp	Determines which paint/close/clip operators are used for the path.

PDStyle

A **PDStyle** object provides access to information on the fonts, font sizes, and colors used in a **PDWord**. **PDStyle** methods include:

PDStyleGetColor	Gets a style’s color.
PDStyleGetFontSize	Gets a style’s font size.

PDTText

A **PDTText** is a graphic object (a subclass of **PDGraphic**) representing one or more character strings on a page. For details, see the following sections in the [PDF Reference](#):

- Section 4.1, “Graphics Objects”
- Section 5.3, “Text Objects”

Like paths, text can be stroked or filled, and can serve as a clipping path.

There are **PDTText** methods to access the text-specific parameters in the graphics state. See Section 4.3, “Graphics State,” in the [PDF Reference](#) for a discussion of graphics state. **PDTText** methods include:

PDTTextEnum	Enumerates the strings of a text object.
--------------------	--

PDTextGetState

Gets the text state for a text object.

PDTextAnnot

A **PDTextAnnot** corresponds to a PDF text annotation. For details, see "Text Annotations" in Section 7.4.5, "Annotation Types," in the [PDF Reference](#). You can use any **PDAnnot** method on a **PDTextAnnot**.

Plug-ins can use **PDTextAnnot** methods to:

- Get and set attributes including the rectangle, textual contents, and whether or not the annotation is open.
- Create new text annotations and delete or move existing ones using **PDAnnot** methods.
- Manipulate the behavior of text annotations by modifying the Text Annotation Handler.

The Acrobat SDK provides three useful macros to cast among **PDAnnot** and its text annotation and link annotation subclasses. These are (see `PDExpT.h`):

- [CastToPDAnnot](#)
- [CastToPDTextAnnot](#)
- [CastToPDLINKAnnot](#)

PDTextAnnot methods include:

PDTextAnnotGetContents

Gets the text of a text annotation.

PDTextAnnotSetOpen

Opens or closes a text annotation.

PDTextSelect

PDTextSelect objects represent a selection of text on a single page, and may contain more than one disjointed group of words. A text selection contains one or more *ranges* of text, with each range containing the word numbers (in PDF order, as returned by [PDWordFinderEnumWords](#) or [PDWordFinderAcquireWordList](#)) of the selected words. Each range has a start word (the first word in the series) and an end word (the first word *not* in the series).

PDTextSelect methods are useful for:

- Processing a text selection created by a user via an Acrobat viewer's user interface
- Programmatically creating a region of text.

You can manipulate text selections using the [PDTTextSelectRangeRec](#) data structure. This structure contains two start/end pairs

- The first pair indicates the word offsets of the start and end words of the selection.
- The second pair indicates the character offsets within the start and end words of the beginning and end of the selection.

NOTE: Plug-ins should set both character offset fields to zero, because the current Acrobat viewer highlights only whole words, not substrings within words.

To create a selection, plug-ins can:

- Supply a list of [PDTTextSelectRangeRec](#) structures to [PDTTextSelectCreateRanges](#)
- Supply a list of word highlights to [PDTTextSelectCreateWordHilite](#)

While character offsets are well-defined quantities in a PDF file, word numbers are calculated by the [PDWordFinder](#) algorithm and, therefore, may change as the word finder algorithm is improved in future versions. Because of this, long-term storage of selection information (in custom annotations, for example) is safer if done with page-relative character offsets and [PDTTextSelectCreatePageHilite](#).

Once a plug-in creates a text selection, it can make it the current selection using [AVDocSetSelection](#).

[PDTTextAnnot](#) methods include the following:

PDTTextSelectCreatePageHilite	Creates a text selection containing one or more words specified by their character offsets from the start of the page.
PDTTextSelectCreateRanges	Creates a text selection from a list of start/stop word offset pairs.
PDTTextSelectCreateWordHilite	Creates a text selection containing one or more words specified by their word offsets from the start of the page.
PDTTextSelectEnumText	Enumerates the strings of the specified text select object, calling a procedure for each string.

NOTE: The first three methods above have new versions, with “**Ex**” appended, which let you specify the version of the word finder algorithm to use (see “[PDWordFinder](#)” on page 106).

PDThread

A *thread* corresponds to an article in the Acrobat viewer's user interface, and contains an ordered sequence of rectangles that bound the article. Each rectangle is called a *bead*. See Section 7.3.2, “Articles,” in the [PDF Reference](#) for more information on articles and beads in PDF.

Threads can be created interactively by the user or programmatically. They are internally represented by a circular linked list of [PDBeads](#).

PDThread methods include the following:

PDThreadNew	Creates a thread.
PDThreadGetFirstBead	Gets an article thread's first bead.
PDThreadIsValid	Tests whether a thread is valid.

PDThumb

A **PDThumb** is a thumbnail preview image of a page.

PDTrans

A **PDTrans** represents a transition to a page. The **Trans** key in a page dictionary specifies a transition dictionary, which describes the effect to use when going to a page and the amount of time the transition should take. See Section 7.3.3, “Presentations,” in the [PDF Reference](#) for more information on transitions. **PDTrans** methods include the following:

PDTransGetDuration	Gets the duration of a transition.
PDTransNew	Creates a new transition

PDViewDestination

A **PDViewDestination** represents a particular view of a page in a document. It contains a reference to a page, a rectangle on that page, and information specifying how to adjust the view to fit the window's size and shape. It corresponds to a PDF **Dest** array (see “Named Destinations” in Section 7.2, “Document-Level Navigation,” in the [PDF Reference](#)) and can be considered a special form of a [PDACTION](#).

PDViewDestination provides a number of methods to get or set the attributes describing the location and size of the view, including the page, rectangle, and fit style.

PDViewDestination methods include:

PDViewDestCreate	Creates a new view destination object.
PDViewDestGetAttr	Gets a view destination's fit type, destination rectangle, and zoom factor.

PDWord

A **PDWord** object represents a word in a PDF file. Each word contains a sequence of characters in one or more styles (see “[PDStyle](#)” on page 101).

All characters in a word are not necessarily physically adjacent. For example, words can be hyphenated across line breaks on a page.

Each character in a word has a *character type*. Character types include: control code, lowercase letter, uppercase letter, digit, punctuation mark, hyphen, soft hyphen, ligature, white space, comma, period, unmapped glyph, end-of-phrase glyph, wildcard, word break, and glyphs that can't be represented in the destination font encoding. See [Character Type Codes](#) in the *Core API Reference* for details.

The [PDWordGetCharacterTypes](#) method can get the character type for each character in a word. The [PDWordGetAttr](#) method returns a mask containing information on the types of characters in a word. The mask is the logical **OR** of several flags, including the following:

- One or more characters in the word cannot be represented in the output encoding.
- One or more characters in the word are punctuation marks.
- The first character in the word is a punctuation mark (this bit is on in addition to the punctuation bit).
- The last character in the word is a punctuation mark (this bit is on in addition to the punctuation bit).
- The word contains a *ligature* (a special typographic symbol consisting of two or more characters such as the English **fi** ligature used to replace the two-character sequence, **f** followed by **i**). Ligatures are used to improve the appearance of a word.
- One or more characters in the word are digits.
- There is a hyphen in the word.
- There is a soft hyphen in the word.

A word's location is specified by the offset of its first character from the beginning of the page (known as the *character offset*). The characters are enumerated in the order

in which they appear in page's content stream in the PDF file (which is not necessarily the order in which the characters are read when displayed or printed).

A word also has a *character delta*, which is the difference between the number of "characters" representing the word in the PDF file and the number of characters in the word. The character delta is non-zero, for example, when a word contains a ligature.

PDWord methods include the following:

PDWordGetAttr	Gets a bit field containing information on the types of characters in a word.
PDWordGetCharacterTypes	Gets the character type for each character in a word.
PDWordGetCharOffset	Returns the offset of a word from the beginning of the page.
PDWordGetString	Converts a word to a null-terminated string and converts ligatures to their constituent characters.

PDWordFinder

A **PDWordFinder** extracts words from a PDF file, and enumerates the words on a single page or on all pages in a document. The core API provides methods to extract words from a document, obtain information on the word finder, and to release a list of words after a plug-in is done using it.

To create a word finder, use **PDDocCreateWordFinder** or **PDDocCreateWordFinderUCS**.

There are two primary methods of using word finders:

- Calling the method **PDWordFinderEnumWords**, which calls a user-provided procedure each time a word is recognized on a page.
- Using **PDWordFinderAcquireWordList**, which builds a word list for an entire page before it returns. This method can return the recognized words in two possible orders:
 - The order in which the words are encountered in the PDF file.
 - According to word location on the page. For a page containing a single column of text, this generally is the same as reading order. For a page containing multiple columns of text, this is *not* true.

The **PDWordFinder** methods include:

PDWordFinderAcquireWordList	Finds all words on a page, and returns one or more tables containing the words.
------------------------------------	---

PDWordFinderGetNthWord

Gets the nth word in the word list obtained using **PDWordFinderAcquireWordList**.

PDXObject

This object corresponds to a PDF XObject (see Section 4.9.4, “Form XObjects,” in the [PDF Reference](#)). **PDXObject** objects currently used by Acrobat viewers are of one of the two X Object subclasses: **PDIImage** and **PDFForm**. You can use any **PDXObject** method on these objects.

The **PDXObject** methods include:

PDXObjectGetData

Passes the data from an XObject to a user-supplied procedure.

PDXObjectGetSubtype

Gets the subtype of an XObject (**PDIImage** or **PDFForm**).

Introduction

The PDFEdit API provides easy access to PDF page contents. With PDFEdit, your plug-in can treat a page's contents as a list of objects rather than manipulating the content stream's marking operators.

Page content is a major component of a PDF file. It represents the visible marks on a page that are drawn by a set of PDF marking operators. The set of marking operators for a page also is referred to as a *display list*, since it is a list of marking operations that represent the displayed portion of a page. See Section 3.7.1, “Content Streams,” in the [PDF Reference](#) for an overview of page content streams and references to other chapters that describe the marking operators in detail.

PDFEdit provides easy access to PDF page contents. PDFEdit is meant to be used in conjunction with the Acrobat PD layer and Cos layer methods for manipulating PDF documents. To use PDFEdit effectively, you should be familiar with PDF page marking operators and the PD layer of the core API, described in [Chapter 6](#).

PDFEdit works with the page contents associated with the **Contents** key in the page dictionary. See Table 3.17 in the [PDF Reference](#), for the entries in a page dictionary. It can also handle Form XObject appearances represented by the **AP** (appearances) key in an annotation dictionary. See Table 7.9 in the [PDF Reference](#), for the entries in an annotation dictionary.

Overview of PDFEdit

Why PDFEdit?

Acrobat Distiller and PDFWriter create documents from PostScript or as output from a printer driver. Non-PDFEdit methods in the core API allow displaying and printing documents, and provide the ability to rearrange pages and to add annotations. However, most of these manipulations are creation-centered, or only deal with objects at the page level and above. PDFEdit methods, on the other hand, allow your plug-in to deal with objects at the level of a page's contents.

The content of a page either resides in a stream object or an array of stream objects. Inside the stream, the elements of a page are not described as objects; they are described as marking operators. There are graphic and clip states at any point in the page description. This state is modified by other operators (such as **SC**, **w**, and so on).

These streams are difficult to manipulate using non-PDFEdit core API methods for these reasons:

- It is awkward to parse or enumerate a stream. Existing methods such as [PDPageStmGetToken](#) get data from the contents stream, but the tokens returned are uninterpreted page marking operators. The method does not readily allow your plug-in to modify the content.
- Resource handling is difficult. Non-PDFEdit methods in the core API treat resource and contents as unrelated entities. Text is not readily associated with its font resource, for instance.
- Attribute handling requires reverse scanning. Given some piece of a page's contents, it is difficult to determine its attributes. For instance, to determine the font used in a text string, a plug-in must find the immediately preceding **Tf** (text state, font size) operator in the stream.
- The stream must be decoded. The page contents stream is typically encoded to compress it, so it cannot be readily accessed by external programs.

What is PDFEdit?

PDFEdit provides an API to treat page contents as a list of objects whose values and attributes can be modified. PDFEdit allows plug-ins to read, write, edit, and create page contents and page resources, which may contain fonts, images, extended graphics states, and so on. For details, see Section 3.7.1, “Content Streams and Resources” in the [PDF Reference](#). PDFEdit also provides mapping between document fonts and system fonts and allows creating new page content objects.

PDFEdit offers these advantages:

- PDFEdit objects are independent of each other. Each object encapsulates all the relevant information about itself. A text object contains font attributes, for instance.
- Your plug-in can use PDFEdit methods to modify the appearance of a page. It can convert a page's content to a **PDEContent** (see “[PDFEdit Paradigm](#)” on [page 110](#)), change the **PDEContent**, and then write it back to the page. Your plug-in also can create pages from scratch.

PDFEdit Paradigm

PDFEdit converts page contents, XObjects, and charprocs to a **PDEContent** object for the page. A **PDEContent** object contains a linear list of objects, which your plug-in can manipulate or create from scratch. It can convert a **PDEContent** back to page contents and resources, thus modifying the page. PDFEdit works with *one page at a time*.

The only effect of the ordering of objects in the display list is *layering*. Objects that occur later in the display list can obscure earlier objects (or partially obscure them, with the introduction in PDF 1.4 of transparency). There is no other meaning that the

order provides. For example, the fact that some text appears later in the list implies nothing about its placement on the page.

When reading and modifying page display lists with PDFEdit, the resulting page stream may be very different from the original. For example, there are many ways to represent the text drawn on a page. PDFEdit is not constrained by the representation used in the original page stream. The resulting stream will, of course, have exactly the same visual representation if a plug-in simply reads a page's contents and then writes the contents back using PDFEdit methods.

PDFEdit works mainly with the page contents associated with the **Contents** key in the page dictionary of a PDF file.

PDFEdit Classes

PDFEdit defines a set of classes to represent the contents and resources of a page. [Figure 7.1](#) shows the PDFEdit class inheritance hierarchy.

Like the other core API classes, these classes are implemented as C structures rather than C++ classes.

Basic Classes

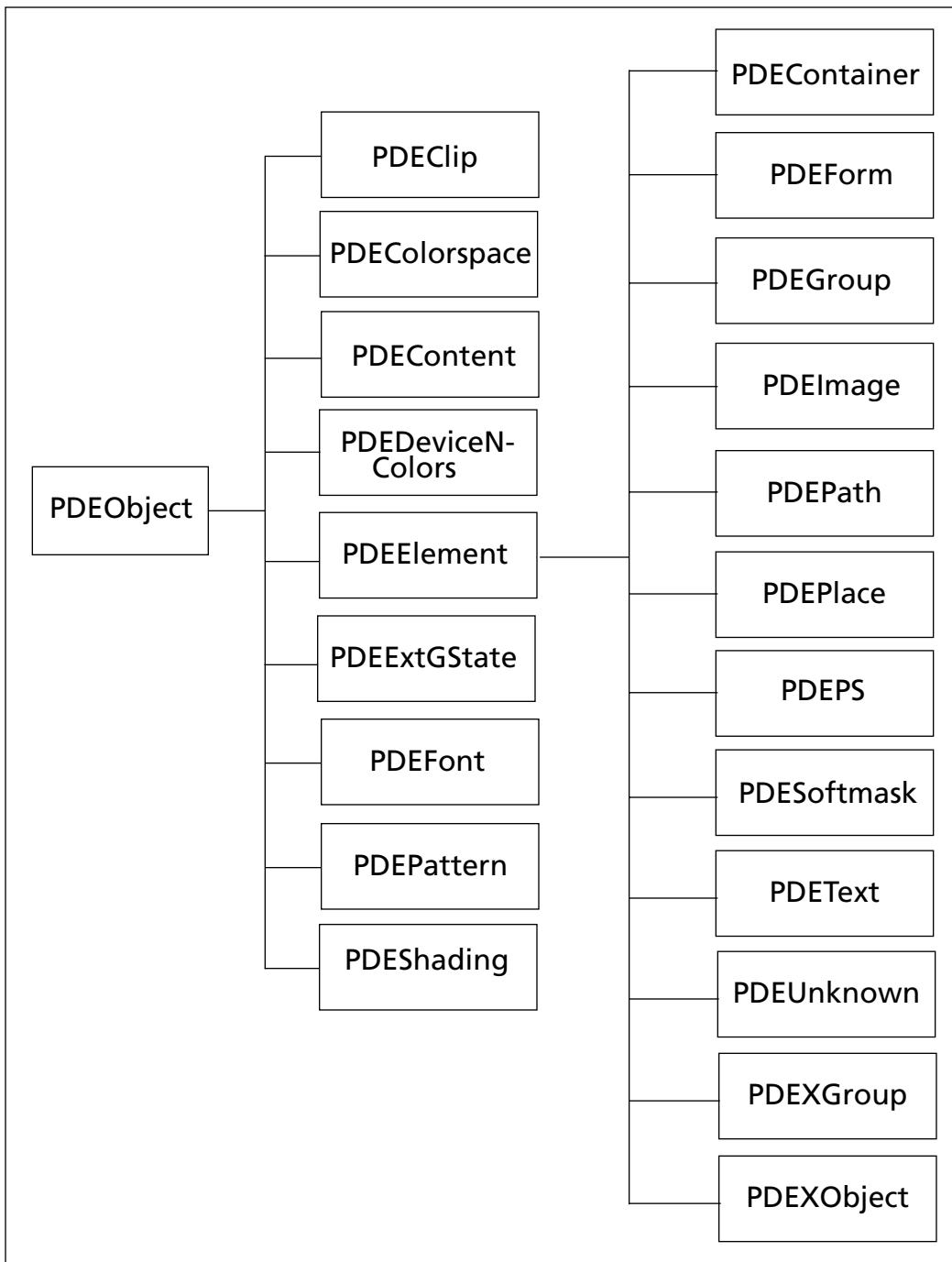
PDEObject is the base class of all PDFEdit objects.

A page display list is represented as a **PDEContent** object that contains **PDEElement** objects. Each **PDEElement** object is a path, text, image, form, or a marked content place or container of **PDEElements**. Objects in the **PDEContent** are listed in the page's drawing order. Your plug-in can add or remove objects inside a **PDEContent**. It can also change attributes of objects in a **PDEContent**, such as a bounding box, a text font, or a clipping path.

Each **PDEElement** contains its state: colors, matrix, fonts, and clip. Each **PDEElement** is independent of the others. Therefore, the display list does not need to be traversed to determine an object's clip or matrix. An element can be moved or copied from one display list to another without relying on or altering the neighboring elements.

A plug-in can attach information to **PDEElements**. This information is identified by a client ID and a client-provided key or tag. Any **PDEElement** can be queried for its client ID and tag.

FIGURE 7.1 PDFEdit Classes



Several PDFEdit classes are container classes. A **PDEContent** is a list of the objects on a page. A **PDEContainer** contains a set of **PDEElements** drawn on a page between marked content operators. A **PDEClip** contains a set of path and text objects defining a clipping path.

NOTE: Since all PDFEdit classes derive from the **PDEObject** class, your plug-in can cast any PDFEdit object as a **PDEObject** and use it with **PDEObject** methods.

PDEElement Classes

PDEElement is the base class of page elements. The following classes represent these elements:

PDEContainer	A container of PDEElements collected between marked content BMC/EMC or BDC/EMC pairs.
PDEForm	An XObject form. Forms are listed in page XObject resources. See Section 3.7.2, “Resource Dictionaries,” in the PDF Reference for details on the entries in a resource dictionary.
PDEGroup	A container of PDEElements .
PDEImage	An inline or XObject image. XObject images are listed in page XObject resources.
PDEPath	A path.
PDEPlace	A place in the display list, marked by an MP or DP operator.
PDEPS	A pass-through PostScript object.
PDESoftMask	A reference to a Softmask object.
PDEText	Text.
PDEUnknown	An unknown element.
PDEXGroup	A reference to an XGroup resource.
PDEXObject	An arbitrary type of XObject

NOTE: Since these classes all derive from the **PDEElement** class, your plug-in can cast any object in these classes as a **PDEElement** and use it with **PDEElement** methods.

PDEElement Attribute Classes

PDEElements can have attributes, represented by these classes:

PDEClip	Container of PDEPath and PDEText objects describing paths and charpaths.
----------------	--

PDEColorSpace	Color space attribute of a PDEElement . Color spaces are listed in page ColorSpace resources.
PDEExtGState	Extended Graphics State attribute of a PDEElement . ExtGStates are listed in page ExtGState resources.
PDEFont	A font. Part of a text's font information is listed in page Font resources.

Example

The following sample outlines how to add text and a path to a page using *PDFEdit* methods.

```

/* Get contents object for a page */
PDDoc pdDoc = PDDocCreate();
PDPage pdPage = PDDocAcquirePage(pdDoc, 0);
PDEContent pdeContent = PDPageAcquirePDEContent(pdPage, clientID);

/* Set up some objects needed... */
/* Set up m as a transformation matrix */
ASFixedMatrix m;
...
/* Set up a graphics state */
PDEGraphicStateP gstateP;
...
/* Get a system font */
PDSysFont f;
...
/* Create text and font objects */
PDEText pdeText = PDETextCreate();
PDEFont pdeFont = PDEFontCreateFromSysFont(f, 0);

/* Add text to the end of the contents */
/* Create an array to hold the text to add */
ASU8 hello[] = "Hello!";

PDETextAdd(pdeText, kpDETextRun, 0, hello, 6, pdeFont, &gstateP,
           sizeof(gstateP), NULL, 0, &m, NULL);
PDEContentAddElem(pdeContent, kpPDEAfterLast, (PDEElement)pdeText);
PDERelease(pdeText);

/* Create a path and add it to contents */
Fixed pathData[6]; //array for path elements
pathData[0] = kpDEMoveTo; //moveto operator
pathData[1] = Int32ToFixed(10); // x
pathData[2] = Int32ToFixed(100); // y
pathData[3] = kpDELineeTo; //lineto operator

```

```
pathData[4] = Int32ToFixed(400); // x
pathData[5] = Int32ToFixed(100); // y
PDEPath pdePath = PDEPathCreate();
PDEPathSetPaintOp(pdePath, kPDEStroke);
PDEPathSetData(pdePath, pathData, 4*6);
PDEContentAddElem(pdeContent, kPDEAfterLast, (PDEElement)pdePath);
PDERelease(pdePath);

/* Put content back in page */
PDPageSetPDEContent(pdPage, clientID);
PDPageReleasePDEContent(pdPage, clientID);
PDPageRelease(pdPage);
```

Comparing PDFEdit to Other Core API Methods

Classes

PDFEdit has its own classes, distinct from the classes of objects used in a **PDPage**. For instance, a **PDEFont** object is *not* a **PDFFont** object.

Mapping Between PDF Operators and PDFEdit

In general, a sequence of PDF page marking operators creates the visible content of a page. These operators may not be closely associated inside the stream.

For instance, each **Tf** (text state, font size) operator sets the font size and font used for the subsequent **Tj** (text showing, show a text string) operator, which places a text string in a page's contents until the next **Tf** operator. Thus, for any text string, there's a set of associated attributes set by the **Tf** operator and other operators that affect text appearance.

The **PDETText** object corresponding to this text also has a set of attributes. However, these attributes are *get* and *set* with **PDETText** methods rather than by manipulating a stream.

The PDF page marking operators that directly create marks on the page, such as **Tj**, correspond to the PDFEdit methods that create objects, such as **PDETTextCreate**. The operators that determine *how* marks are placed on a page, such as **Tf**, correspond to the PDFEdit methods that change an object's attributes, such as **PDETTextRunSetFont**.

Page Contents Stream and PDFEdit Object List Correspondence

Every page in a PDF file has a contents member, which is either a stream or an array of streams. Without PDFEdit, your plug-in would access these streams through the

ASStream object methods in the core API; but as noted in “[Why PDFEdit?](#)” on page 109, the streams are not easy to manipulate.

ASStream objects are not objects in the PDF file. They are abstractions that the core API uses for reading, seeking, and so on.

PDFEdit converts these content streams to a **PDEContent** object, containing various **PDEElement** objects in a list that represents the page content. The list objects can be rearranged, removed, or added to. The objects in the list have attributes that may be changed as well.

PDFEdit does not provide the ability to *directly* modify an existing page contents stream. When a display list is converted back to a stream, an entirely new stream is written. It is not possible to insert a stream of new bytes into, or remove a stream of bytes from, a stream while leaving the rest of the stream unchanged.

PDFEdit *does* let your plug-in edit an existing page’s contents stream. It makes a copy of the page for editing. Then it replaces the existing page with the modified copy. The Acrobat viewer does not know the difference. As far as it is concerned, the page’s contents stream has been updated.

Enumerating Page Objects

PDFEdit provides methods your plug-in can use to determine how many objects there are on a page. The example code below shows how a plug-in can get the number of objects and loop through them to obtain individual **PDEElements**. For each object, it can then determine the object type.

See the [Acrobat Plug-In Tutorial](#) for details on the complete plug-in example that this code is part of.

```
numElems = PDEContentGetNumElems(pdeContent);
/* loop through elements to find text */
for (j = 0; j < numElems; j++)
{
    ASFixedRect bbox;
    AVRect rect;
    pdeElement = PDEContentGetElem(pdeContent, j);
    /* determine if the object is of type text */
    if (PDEObjectGetType((PDEObject) pdeElement) == kPDETText)
        ...
    ...
}
```

NOTE: Two core PD layer methods were available in pre-4.0 Acrobat to describe the objects in a page description, namely: **PDPagEnumContents** and **PDPagEnumResources**. It is not recommended that you use either of these methods in new plug-ins, as they cannot fully parse PDF files that are version 1.2 or later.

Using PDFEdit versus PDWordFinder

A **PDWordFinder** provides access to the **PDWords** of a **PDPPage**. Given a **PDWord**, your plug-in can

- Find its position and character offset on a page
- Find various attributes such as whether it has ligatures
- Get a text representation of the word

However, your plug-in cannot alter **PDWords** or their placement on the page.

PDFEdit does not recognize **PDWords**. Instead, it deals with the text on a page represented as **PDETText** objects. Your plug-in not only can get information about the text, but it can alter text and its attributes, such as the font and position on the page.

When a **PDPPage**'s contents are first converted to a **PDEContent**, a plug-in can map the **PDWords** in a **PDWord** list to the characters in the **PDEContent**. After changing the **PDEContent**, this direct mapping is lost. A plug-in can obtain a new **PDWord** list by converting a **PDEContent** into the contents of a **PDPPage**, and then calling **PDWordFinderAcquireWordList**.

The design of PDFEdit assumes that characters and show strings—not words—are the basic elements of a **PDETText** object. When a plug-in edits and creates text blocks, it must be able to specify the location of each character to perform kerning and other character placement operations. Having page display operations use words as a basic element is not general enough in this situation.

Using PDFEdit Versus PDPPageAddCosContents

PDPPageAddCosContents completely replaces the contents of a specified page with new contents. The new contents must be a Cos object, which your plug-in could get from the **Contents** key of the page contents provided by the **PDPPageGetCosObj** method. However, the Cos object contains streams that are difficult to manipulate, as discussed in “[Why PDFEdit?](#)” on page 109.

PDFEdit allows you to manipulate or add to the objects on a page, treating them as text, path, image, form, XObject, and container objects.

Hit Testing

PDFEdit provides the ability to locate objects at a point. The **PDEElementIsAtPoint** and **PDEElementIsAtRect** methods allow your plug-in to determine whether a point or rectangle is on an element. **PDETTextIsAtPoint** and **PDETTextIsAtRect** provide the same capability for text objects.

PDFEdit also allows your plug-in to specify exact placement on a page of text, graphics, and path objects. If your plug-in knows where a user clicked on screen, it can convert the device space coordinates to user space coordinates and determine which objects are beneath that point.

Using PDFEdit Methods

Reference Counting

All **PDEObjects** contain a *reference count*. The reference count is initialized to 1 when an object is created. The count is also incremented when an object is added to another object, such as a **PDEContent**, and is decremented when it is removed from an object. When the reference count becomes 0, the object is freed and the reference to the object is no longer valid.

A plug-in may explicitly increment or decrement the reference count of an object using the **PDEAcquire** and **PDERelease** methods, respectively. When a plug-in no longer needs an object, it should call **PDERelease**.

Objects should only be disposed of with **PDERelease** if the method by which they were obtained incremented the reference count for the object. In general, methods that “get” an object do not increment the reference count. As you recall from “[Core API Methods](#)” on page 24, methods that increment the reference count typically contain the word “acquire” or “create” in the method name and specifically state that your plug-in must release the object in the method description.

All of the **Create** methods, including the **PDEContentCreateFromCosObj** method, set the reference count to 1 on the newly created object.

When a plug-in adds an object to a container, that container’s reference count is incremented. When a container’s reference count becomes 0, it releases all of its contained objects. Typically, if an object is created and added to a container, a plug-in should call **PDERelease** immediately after the add operation. The object then has a reference count of 1 and will be destroyed when its container is destroyed.

Your plug-in should take care when removing an object from one container and adding it to another. It should acquire the object *before* removing it from the container. Otherwise, the container’s reference count could reach 0 when your plug-in removes it. In this situation, adding the object to another container would be illegal since the object would be invalid.

The **Get** methods in the API do not change the reference count. However, a plug-in must be careful not to hold an object it did not acquire for too long, since a subsequent remove operation on its container could destroy the object.

The **Set** methods increment the reference count of the object whose attribute is set. For example, the **PDETextRunSetFont** method increments the reference count of the font. Furthermore, the reference count of the previous attribute object (the older font in this example) is decremented.

The **PDEGraphicsState** attribute contains references to up to five objects:

- Fill and stroke color spaces
- Fill and stroke color objects (if the color space is Pattern)
- The ExtGState

(Some of these objects may be **NULL**.) The **Get** and **Set** rules apply to the component references in this case.

A **PDEObject** may be contained or referenced by more than one **PDEObject**. This is obvious in the case of resources such as fonts and color spaces. Other objects also may be referenced multiple times. For example, a **PDEElement** may be contained in two **PDEContents**.

The best approach a plug-in can take to reference counting is to:

- Create its objects
- Add the objects to a content object
- Add the content object to a page
- Release all the objects it created

Matrix Operations

Several **PDEElements** have matrixes associated with them.

PDFEdit flattens matrixes on input and output. When parsing a page content stream, it sets the matrix of an element to the value of the current transformation matrix. When parsing a path, the matrix is applied to the path segments. Thus, after parsing a page, paths have the identity matrix, images contain the final image matrix, and text runs contain a single matrix composed of the graphics state matrix, the text state matrix, and the text placement and scaling operators.

The **PDEElementSetMatrix** method applies the matrix immediately to images; **PDETTextRunSetMatrix** applies the matrix immediately to text runs. The path matrix is not applied to path segments until the page is emitted to a page content stream. After emission, the path matrix is reset to the identity matrix. This operation is deferred because it can be time consuming to apply the matrix to each path segment in a large path. Because the matrix operation is deferred, a plug-in must always examine the path matrix when processing path data.

Clip Objects and Sharing

A **PDEClip** object may be shared among multiple **PDEElements**. Therefore, care must be taken when a plug-in changes the clip of an element, since modifying the clip of one element can have the side effect of modifying the clip for multiple elements. If the clip for only one element is to be changed, a plug-in should copy the clip object and apply the modifications to the copy.

Marked Content

PDFEdit supports the marked content operators. These operators provide a mechanism for attaching additional meaning to locations and to objects in a page content stream. For details, see Section 8.4.2, “Marked Content,” in the [PDF](#)

Reference. The **PDEPlace** object marks a location in a sequence of **PDEElements**. The **PDEContainer** object marks a group of 0 or more **PDEElements**.

The **PDEContainer** contains a **PDEContent**, which itself contains the marked **PDEElements** that constitute the markings on the page. Nested marked content operators result in nested **PDEContainers** and **PDEPlace** objects.

When enumerating a **PDEContent** that contains marked content operators, it is necessary for a plug-in to examine the **PDEContainers** to find all renderable **PDEElements**.

Cos Objects and Documents

Many of the **PDEObjects** contain a reference to a Cos object. Examples include **PDEFONT**, **PDEFORM**, and some **PDECOLORSPACE** objects. In the current Cos implementation, composite and indirect objects belong to a particular document. Cos objects *cannot* be intermingled between documents. For example, a dictionary in one document cannot contain a reference to an array in another document.

When a plug-in calls **PDEContentToCosObj**, it specifies a destination Cos document. If the **PDEContent** contains references to Cos objects in a different document, PDFEdit makes copies of the Cos objects in the destination document and refers to the copies.

Some of the PDFEdit **Create** methods can create a Cos object, such as **PDEFONTCreateFromCosObj** and **PDEFORMCreateFromCosObj**. PDFEdit creates these Cos objects in a scratch document that it maintains. This can result in unexpected behavior from the **GetCosObj** methods. A method may return the original object from which the **PDEObject** was instantiated, or it may return a copy of the Cos object in a different document.

XObjects and PDEObjects

A Cos XObject resource (an image or a form) may occur more than once on a page. A distinct **PDEElement** is created for each occurrence, but the **PDEElements** share the underlying Cos object. Each **PDEElement** typically has a distinct matrix and graphic state. When a **PDEContent** is written to a stream, multiple references to the Cos objects are recognized such that all instances of an XObject are referred to by the same resource name.

Resources

In PDF, page content streams do not directly refer to Cos resources. Instead, they refer to them by a name that is used to look up the Cos resource in the **Resources** dictionary. When reading and writing streams in an existing document, PDFEdit tries to preserve the original page resource names. When this is not possible, PDFEdit generates new names as required.

PDFEdit maintains a list of names and Cos objects for each open document to which any PDFEdit operation has been performed. Every time a stream is generated, PDFEdit consults the database. If no name exists for a Cos object, it generates a new name and adds the name to the database. If a name exists but is already in use by another Cos object, it generates an alternative name. Thus, a single Cos object may be referenced by different names in different page content streams.

For example, consider a document with two pages. Page 1 contains text set in Helvetica, and the font's page resource name is F1. Page 2 contains text set in Times-Roman, and the font's page resource name also is F1. Now, if a plug-in adds Helvetica text to page 2, it cannot use the resource name F1, because F1 already refers to Times-Roman on this page. Therefore, PDFEdit generates a new page resource name for the Helvetica added to page 2.

Each form (and Type 3 font) should contain a resources dictionary. It is the plug-in's responsibility to put the resources dictionary in the form's attribute dictionary upon return from the [PDEContentToCosObj](#) method when the [PDEContent](#) is a form.

Client Identifiers

Some methods such as [PDEAddTag](#) or [PDPageAcquirePDEContent](#) require a client identifier (ID).

A plug-in should use its [gExtensionID](#) for the client ID.

When several plug-ins want to operate on the same page at the same time, they should use a separate ID for each thread.

Guide to Page Creation

In general, a plug-in should use [PDPageAcquirePDEContent](#) to obtain a page's [PDEContent](#). The plug-in can modify the [PDEContent](#) with PDFEdit methods as desired. Then it should put the [PDEContent](#) back on the page with [PDPageSetPDEContent](#) and call [PDPageReleasePDEContent](#) after it is completely done with the page.

Common Code Sequence

A plug-in can frequently use a sequence like this to get a page's contents, modify it, and put it back:

```
PDEContent pdecontent;
PDPage pdpage;
ASBool result;
ASInt32 count;

/* Get page's contents */
pdecontent = PDPageAcquirePDEContent(pdpage, clientID);
```

```

/* Modify the contents */
...
/* Put the contents back in the page */
result = PDPageSetPDEContent(pdpage, clientID);
/* Clean up */
count = PDPageReleasePDEContent(pdpage, clientID);
/* Release the page */
PDPageRelease(pdpage);

```

Ways To Modify a Page's Content

This section shows how to modify a page's content in various ways.

Sequence for Adding Text to a Page

```

/* Enumerate the system fonts. Look for font=FONTPNAME */
/* The fontEnumProc callback looks for a font */

PDEFont gFont;
char buf[255];
PDEText pdeText = NULL;
PDETextState tState;
FixedMatrix matrix;
PDEFontAttrs textFontAttr;
PDEGraphicState gstate;

PDEnumSysFonts(fontEnumProc, (void *)FONTPNAME);

/* If font found, proceed */
if (gFont) {
    strcpy(buf, "Added text...");

    /* Create a new PDEText object */
    pdeText = PDETextCreate();
    memset(&tState, 0, sizeof(PDETextState));

    /* Set the text matrix to 24 pt type, v=300 pts, h=72 pts */
    memset(&matrix, 0, sizeof(FixedMatrix));
    matrix.a = Int16ToFixed(24);
    matrix.d = Int16ToFixed(24);
    matrix.v = Int16ToFixed(300);
    matrix.h = Int16ToFixed(72);

    /* Set the text attributes, including font name and type of font */
    /* Type of font could be Type1, MMType1, or TrueType */

    memset(&textFontAttr, 0, sizeof(textFontAttr));
    textFontAttr.name = ASAtomFromString(FONTPNAME);
    textFontAttr.type = ASAtomFromString("Type1");

    /* Set up the default Graphics state */

```

```

memset(&gstate, 0, sizeof(PDEGraphicState));
gstate.strokeColorSpec.space =
gstate.fillColorSpec.space = pdeColorSpace;
gstate.miterLimit = fixedTen; /* constants */
gstate.flatness = fixedOne;
gstate.lineWidth = fixedOne;

/* Create PDETText object as a text run, */
/* and add to PDEContent */
PDETTextAdd(pdeText, kPDETTextRun, 0, (unsigned char *)buf,
strlen(buf), gFont, &gstate, sizeof(gstate), &tState,
sizeof(PDETTextState), &matrix, NULL);

/* Add to end of contents */
PDEContentAddElem(pdecontent, kpDEAfterLast, (PDEElement)pdeText);

/* Release the PDETText object after it has been added */
PDERelease((PDEObject)pdeText);
}

```

Sequence for Adding a Path to a Page

```

/* Draw a filled rectangle */
PDEColorSpace pdeColorSpace;
PDEPath pdePath;
Fixed pathSeg[5];
PDEGraphicState gstate;

pdeColorSpace =
PDEColorSpaceCreateFromName(ASAtomFromString("DeviceGray"));

/* Draw a filled in rectangle */
pdePath = PDEPathCreate();
pathSeg[0] = kPDERect; /* rectangle */
pathSeg[1] = Int32ToFixed(72); /* x */
pathSeg[2] = Int32ToFixed(2*72); /* y */
pathSeg[3] = Int32ToFixed(3*72); /* width */
pathSeg[4] = Int32ToFixed(72); /* height */

PDEPathSetPaintOp(pdePath, kPDEFill);

/* Set up the default Graphics state */
memset(&gstate, 0, sizeof(PDEGraphicState));
gstate.strokeColorSpec.space = gstate.fillColorSpec.space =
pdeColorSpace;
gstate.miterLimit = fixedTen; /* constants */
gstate.flatness = fixedOne;
gstate.lineWidth = fixedOne;

PDEElementSetGState((PDEElement) pdePath, &gstate, sizeof(gstate));
PDEPathSetData(pdePath, pathSeg, sizeof(pathSeg));

```

```
/* Add rectangle to end of PDEContent */
PDEContentAddElem(pdecontent, kPDEAfterLast, (PDEElement)pdePath);
/* Release the PDEPath object */
PDERelease((PDEObject)pdePath);
```

Sequence for Adding an Image to a Page

```
/* Create a simple bitmap graphic */
char* data;
PDEImageAttrs imageAttrs;
FixedMatrix matrix;
PDEColorValue pdeColorValue;
PDEColorSpace pdeColorSpace;
PDEImage pdeImage = NULL;

pdeColorSpace =
    PDEColorSpaceCreateFromName(ASAtomFromString( "DeviceGray" )) ;

data = (char*)ASmalloc(sizeof(char)*10000);
if (data)
{
    memset(&imageAttrs, 0, sizeof(PDEImageAttrs));
    imageAttrs.flags = kPDEImageExternal;
    imageAttrs.width = 100L;
    imageAttrs.height = 100L;
    imageAttrs.bitsPerComponent = 8L;

    /* Fill in matrix for image. Offset it to v=500 pts, h=72 pts */
    matrix.a = Int16ToFixed(100);
    matrix.b = matrix.c = fixedZero;
    matrix.d = Int16ToFixed(-100);
    matrix.v = Int16ToFixed(500);
    matrix.h = Int16ToFixed(72);

    memset(&pdeColorValue, 0, sizeof(PDEColorValue));
    pdeColorValue.color[0] = fixedZero;

    /* Create stream for image data (you can also use a */
    /* file stream or a proc stream) */
    memset(data, 0, 10000);
    for (i = 0; i < 5000; i++)
        data[i]=1;
    stm = ASMemStmRdOpen(data, 10000L);

    pdeImage = PDEImageCreate(&imageAttrs, (Uns32)sizeof(imageAttrs),
        &matrix, 0, pdeColorSpace, &pdeColorValue, NULL, stm,
        NULL, 0);

    /* Add the image to end of PDEContent */
    if (pdeImage)
    {
        PDEContentAddElem(pdecontent, kPDEAfterLast,
```

```
    (PDEElement)pdeImage);
    PDERelease((PDEObject)pdeImage);
}

/* Close the stream and free the image data */
if (stm)
    ASStmClose(stm);
if (data)
    ASfree(data);
}
```

Generating Efficient Pages

Create a **PDPage** before creating the **PDEContent** for it. Page notifications work better this way.

Font Embedding and Subsetting

To author a document with an embedded system font, a plug-in must set the **kPDEFontCreateEmbedded** flag when calling **PDEFontCreateFromSysFont**. This causes the font to be embedded when first used in a document; no further work is required.

To author a document with an embedded and subsetted system font, a plug-in must set both the **kPDEFontCreateEmbedded** and **kPDEFontWillSubset** flags when calling **PDEFontCreateFromSysFont**. The font is embedded and given a subset name, but it is not subsetted until later. The plug-in can then use characters from the font at will; PDFEdit tracks which characters were actually emitted into each **CosDoc** via **PDEContentToCosObj**. PDFEdit tracks character usage separately for each **CosDoc** in which a font is used.

A plug-in eventually must call **PDEFontSubsetNow** on a font to subset it. The font is subsetted to contain only the characters used in the **PDEText** objects that reference that font. The plug-in should call **PDEFontSubsetNow** after it has set the content for the page. Calling **PDEFontSubsetNow** on a font that was not created using the **kPDEFontWillSubset** flag does nothing.

A font that is created using the **kPDEFontCreateEmbedded** flag is always embedded. A plug-in does not need to call **PDEFontSubsetNow** if it sets the **kPDEFontWillSubset** flag. In addition, it is possible for a plug-in to call **PDEFontSubsetNow** multiple times. The subsetted data is rewritten with glyphs for any additional text used with the font.

A plug-in also can call **PDEmbedSysFontForPDEFont** to embed a system font, including one for which **kPDEFontWillSubset** was specified. Therefore, the plug-in can create a font with the **kPDEFontWillSubset** flag, and, at a later time, decide to leave the font unembedded (by doing nothing), create the subset by calling **PDEFontSubsetNow**, or embed the entire font by calling **PDEmbedSysFontForPDEFont**.

Proper Use of Marked Content

Marked content allows a plug-in to identify, characterize, and organize a PDF file. For instance, it may mark a set of paragraphs with style information. Similarly, a place in the document may have information for looking up entries in a database. The structure that marked content provides to a document can facilitate extracting data from a PDF file or converting it to another file format. Structural information could be used by other programs to implement PDF and Acrobat enhancements.

PDF marked content operators are used in page descriptions to indicate a part of the stream that may be significant to an application other than a strict PDF consumer. These operators attach a tag and, optionally, a property list, to part of the stream.

There are two kinds of marks, those that bracket a sequence of objects, and those that mark a place in the stream. These operators may appear only *between* objects.

The **BMC/EMC** or **BDC/EMC** operators bracket an object sequence and correspond to the **PDEContainer**, which contains a set of objects. This is useful for grouping objects that “belong” together in some sense, defined by the document creator. Bracketed sequences may be nested within each other, and thus a **PDEContainer** may contain other **PDEContainer** objects.

Places are marked with either **MP** or **DP** operators. These correspond to the **PDEPlace** object, which marks a place in the object list. Theoretically, your plug-in could bracket a sequence of operators with a pair of related **PDEPlace** objects, but this is not recommended. Use the **PDEContainer** for enclosing a set of objects.

The **BDC** and **DP** operators take a property list dictionary argument; otherwise they function identically to **BMC** and **MP** operators. Similarly, the **PDEContainerCreate** and **PDEPlaceCreate** methods take an optional dictionary parameter, in addition to a tag for the object. Use this dictionary to provide additional information about the **PDEContainer** or **PDEPlace**, if needed.

For details on the marked content operators, see Section 8.4.2, “Marked Content,” in the [PDF Reference](#).

Debugging Tools and Techniques

Dump methods allow dumping objects and their attributes.

Object Dump

The **PDEObjectDump** enumeration method gets a text description of a given object. Since objects can be nested—**PDEContent** objects can contain other objects—your plug-in specifies the nesting level for the children and attributes it wants to see. The **PDEObjectDumpProc** callback specified in **PDEObjectDump** returns a buffer with text describing each object.

The following example illustrates dumping a content object, **pdecontent**, to a file:

```

/* Dump the content object */
PDEContent pdecontent;
...
PDEObjectDump((PDEObject)pdecontent, 10,
    ASCCallbackCreateProto(PDEObjectDumpProc, myPDEObjectDumpProc),
    NULL);
...
/* Dump callback function */
ACCB1 void ACCB2 myPDEObjectDumpProc(PDEObject pdeobject,
    char* dumpInfo, void* clientData)
{
/* Output the data to a file */
FILE *f;

f=fopen( ":PDEObjectDump.txt", "a" );
if (f)
{
    fprintf(f, "%s\n", dumpInfo);
    fclose(f);
}
}

```

Here is a dump of a **PDEContent** object, showing its flags and number of elements, among other information. It also includes the objects inside the **PDEContent**:

- A path
- An image
- A text object

The number after the pound sign (#) is each object's reference count.

```

Content (0) #1 3a9d264
Content flags: none Num elems: 3

Path (2) #1 3a9d31c
Op: fill Size: 20 bbox 71 143 288 216
{
    ColorSpace (9) #6 3a9d2fc
    {DeviceGray }
    fill: DeviceGray 0 0 0 0
    Rect: 72 144 216 72
}

Image (3) #1 3a9dbec
CSpace: DeviceGray bbox 72 400 172 500

Text (1) #1 3a9dbac
Num elems: 1
{
    font: 3a9e2cc
    matrix: 24 0 0 24 72 300 bbox 67 294 256 323
}

```

```

    fill: DeviceGray 0 0 0 0
    This is a test.
}

```

The objects inside the **PDEContent** could also be dumped individually.

Dump Log

The **PDELogDump** method's **PDEObjectDumpProc** callback returns a buffer with text describing each object that has been created. The text is in the object dump format noted above.

Attribute Enumeration

The **PDEAttrEnumTable** method enumerates the shared resource objects, providing their reference counts.

Reference Counts

Your plug-in can determine if any object has a non-zero reference count, that is, it has not released the object.

After your plug-in has released a page, it should call **PDEAttrEnumTable** with an enumeration function. The function is called back with every attribute object that has a reference count greater than 0. The enumeration function provides an opaque pointer to each object, which your plug-in can compare to the objects it created to check if it did not release an object.

NOTE: Some objects, such as fonts, are allocated at initialization time and deallocated at termination time, so some **PDEObjects** are left over after your plug-in completes its processing of the PDF file and releases objects it created.

PDFEdit Methods

Dump Methods

Dump methods allow enumerating objects and their attributes. An object's information can be dumped in human readable form. An object dump includes its reference count, which is useful in debugging reference count problems. The **Dump** methods include:

PDELogDump	Enumerates PDEObjects .
PDEObjectDump	Dumps an ASCII version of an object, its children, and their attributes.
PDEAttrEnumTable	Enumerates the table of attributes.

General Methods

These utility methods simplify tasks, such as setting up graphics information structures and merging resources for a page. The **General** methods include the following:

PDEDefaultGState	Fills out a structure with the default graphic state.
PDEEnumElements	Enumerates all PDEElements in a given stream.
PDEMergeResourcesDict	Merges two resources dictionaries.

PDEClip

A **PDEClip** is a list of **PDEElements** containing a list of **PDEPaths** and **PDETexts** that describe a clipping state. **PDEClips** can be created and built up with **PDEClip** methods. Any **PDEElement** object can have **PDEClip** associated with it by using the **PDEElementSetClip** method. The **PDEClip** methods include the following:

PDEClipAddElem	Adds an element to a clipping path.
PDEClipCreate	Creates an empty clip object.
PDEClipGetElem	Gets an element from a clip object.
PDEClipGetNumElems	Gets the number of path and charpath elements in a clip object.
PDEClipRemoveElems	Removes one or more elements from a clip object.

PDEColorSpace

A **PDEColorSpace** object is a reference to a color space used on a page. The color space is part of the graphics state attributes of a **PDEElement**. See Sections 4.5, “Color Spaces,” in the [PDF Reference](#), for details on color spaces and color operators. The **PDEColorSpace** methods include the following:

PDEColorSpaceCreate	Creates a new color space object of the specified type.
PDEColorSpaceCreateFromCosObj	Creates a color space object from a Cos object.

<code>PDEColorSpaceGetBase</code>	Obtains the name of the base color space.
<code>PDEColorSpaceGetBaseNumComps</code>	Gets the number of components in the base color space of an indexed color space.
<code>PDEColorSpaceGetCosObj</code>	Gets a Cos object for a color space.
<code>PDEColorSpaceGetCTable</code>	Obtains component information for an indexed color space.

PDEContainer

A **PDEContainer** contains a group of **PDEElements** on a page. In the PDF file, containers are delimited by the Marked Content operator pairs **BMC/EMC** or **BDC/EMC**. Every **PDEContainer** has a Marked Content tag associated with it. In addition to grouping a set of elements, a **BDC/EMC** pair specifies a property list to be associated with the grouping. Thus a **PDEContainer** corresponding to a **BDC/EMC** operator pair also has a property list dictionary associated with it.

For example, the following PDF marking operators would correspond to a **PDEContainer** that contains several paths and has the tag **PathABC**:

```
\PathABC BMC
1 g
84.96 745.2 449.28 -9.596 ref
1 g
427.957 153.071 m
435.4 153.071 441.436 159.107 441.436 166.549
c
441.436 173.993 435.4 180.028 427.957 180.028
c
420.514 180.028 414.479 173.993 414.479
166.549 c
414.479 159.107 420.514 153.071 427.957
153.071 c
b
EMC
```

A **PDEContainer** is itself a **PDEElement**, so a **PDEContainer** can contain other **PDEContainer** objects, which would reflect nested marked content operator pairs.

Marked content is useful for adding structure information to a PDF file. For instance, a text processing program may have font and style information associated with a paragraph. You may want to retain this information in the PDF file, and marked content provides a means to do so.

See Sections 8.4.2, “Marked Content,” in the *PDF Reference*, for information on marked content and property lists.

A **PDEPlace** object allows marking a single point in a PDF file with information rather than marking a group of objects.

The **PDEContainer** methods include:

PDEContainerCreate	Creates a container object.
PDEContainerGetContent	Gets the PDEContent for a PDEContainer .
PDEContainerGetDict	Gets the marked content dictionary for a container.
PDEContainerGetMCTag	Obtains the marked content tag for a container.
PDEContainerSetContent	Sets the content for a container.

PDEContent

The **PDEContent** object is the workhorse of the PDFEdit API, since it contains the modifiable contents of a **PDPage**.

A **PDEContent** may be obtained from an existing page or from a form XObject or from a Type 3 charproc. You can create an empty **PDEContent**. A **PDEContent** contains **PDEElements**. In addition, a **PDEContent** may have attributes such as Form matrix and setcachedevice parameters.

The simplest way to obtain the **PDEContent** for a page is with the **PDPageAcquirePDEContent** method. After your plug-in modifies the content, it can put it back in the page with **PDPageSetPDEContent**, using the same filters with which the page was originally encoded.

Once your plug-in has the page's **PDEContent**, it can get, add, or remove elements with **PDEContent** methods. It can modify individual page elements with the methods for **PDEElements**, such as **PDETText** or **PDEPath**.

The **PDEContent** methods include:

PDEContentAddElem	Inserts an element into a PDEContent .
PDEContentCreate	Creates an empty content object.
PDEContentGetElem	Gets the requested element from a content.
PDEContentGetNumElems	Gets the number of elements in a PDEContent .
PDEContentGetResources	Obtains the number of resources of a specified type and, optionally, pointers to the resource objects.
PDEContentRemoveElem	Removes an element from a PDEContent .

PDEContentToCosObj	Converts a PDEContent into PDF contents and resources.
---------------------------	---

PDEDeviceNColors

A color space with a variable number of device-dependent components. Usually used to store multiple spot colors in a single color space. The **PDEDeviceNColors** methods include:

PDEDeviceNColorsCreate	Creates an object that can be used to store n color components when in a PDEDeviceNColors color space.
PDEDeviceNColorsGetColorValue	Gets the value of a color component of a PDEDeviceNColors color space.

PDEElement

PDEElement is the base class for elements of a page display list (**PDEContent**) and for clip objects. The general **PDEElement** methods allow you to get and set general element properties.

PDEElement is an abstract superclass from which the **PDEContainer**, **PDEForm**, **PDEImage**, **PDEPath**, **PDEPlace**, **PDEText**, and **PDEXObject** classes are derived. Your plug-in can find the type of an element with the **PDEObjectGetType** method. It can then cast and apply the methods in that class to the object. In addition, it can cast any **PDEElement** subclass object to a **PDEElement** and use it anywhere a **PDEElement** is called for, such as in **PDEElement** methods. The **PDEElement** methods include the following:

PDEElementCopy	Makes a copy of an element.
PDEElementGetBBox	Obtains the bounding box for an element.
PDEElementGetClip	Gets the current clip for an element.
PDEElementGetGState	Obtains the graphics state information for an element.
PDEElementGetMatrix	Obtains the transformation matrix for an element.

PDEExtGState

A **PDEExtGState** object is a reference to an **ExtGState** resource used on a page. It specifies a **PDEElement**'s extended graphics state, which is part of its graphics state, as specified in a **PDEGraphicState** structure. See Section 7.15 and Section 8.2 in the *PDF Reference*, for information on extended graphics states.

```
// The graphics state controls the various style properties of the text
// including color, weight, and so forth.

memset (&gState, 0, sizeof(PDEGraphicState));
gState.strokeColorSpec.space = gState.fillColorSpec.space =
pdeColorSpace;
gState.miterLimit = fixedTen;
gState.flatness = fixedOne;
gState.lineWidth = fixedOne;
gState.extGState = pdeExtGState;
gState.wasSetFlags = kPDEMiterLimitWasSet | kPDEFattnessWasSet |
kPDELineWidthWasSet | kPDEExtGStateWasSet;
```

You can get or set the graphics state associated with a **PDEElement** or **PDEText** object with the **PDEElementGetGState** or **PDEElementSetGState** methods.

Setting the Opacity of an Object

With Acrobat 5.0 and PDF 1.4 and higher, every object has an opacity property (default is opaque) in the extended graphics state. The code snippet below shows how to add the opacity property to either existing or new elements:

```
DURING
    pdeExtGState = PDEExtGStateCreateNew ( PDDocGetCosDoc(pdDoc) );
    PDEExtGStateSetOpacityFill (pdeExtGState, FloatToFixed(0.5));
    PDEExtGStateSetOpacityStroke (pdeExtGState, FloatToFixed(0.5));
HANDLER
    if (pdeExtGState) {
        PDERelease ((PDEObject) pdeExtGState);
        pdeExtGState = NULL;
    }
END_HANDLER
```

For information on transparency, see the section entitled “Transparency in PDF” in *PDF: Changes From Version 1.3 to 1.4*.

PDEExtGState Methods

The **PDEExtGState** methods include the following:

PDEExtGStateCreate

Creates a new **PDEExtGState** from a Cos object.

<code>PDEExtGStateGetCosObj</code>	Obtains a Cos object for a PDEExtGState .
<code>PDEExtGStateGetOpacityFill</code>	Gets the opacity value for the fill of the PDEElement .
<code>PDEExtGStateGetOpacityStroke</code>	Gets the opacity value for the stroke of the PDEElement .
<code>PDEExtGStateGetOPFill</code>	Determines whether overprint is turned on for the fill of the PDEElement .
<code>PDEExtGStateGetOPM</code>	Gets the overprint mode.
<code>PDEExtGStateSetOpacityFill</code>	Sets the opacity value for fill operations.
<code>PDEExtGStateSetOpacityStroke</code>	Sets the opacity value for stroke operations.
<code>PDEExtGStateSetOPStroke</code>	Sets the overprint value for stroke operations.

PDEFont

A **PDEFont** object is a reference to a font used on a page. It may be equated with a font in the system. A **PDEFont** is not the same as a **PDFFont**; a **PDEFont** is associated with a particular document.

See Sections 7.7 to 7.9 in the [PDF Reference](#), for information on fonts.

A **PDSysFont** object represents a system font and is a distinct object from a **PDEFont**. You can create a **PDEFont** from a system font with the `PDEFontCreateFromSysFont` method.

Your plug-in can set the font of a text run with the `PDETextRunSetFont` method. The `PDEFont` methods include the following:

<code>PDEFontCreate</code>	Creates a new <code>PDEFont</code> from specified parameters.
<code>PDEFontCreateFromCosObj</code>	Creates a <code>PDEFont</code> corresponding to a Cos object.
<code>PDEFontCreateWithParams</code>	Creates a new <code>PDEFont</code> from parameters.
<code>PDEFontGetAttrs</code>	Obtains the attributes for a font object.
<code>PDEFontGetCosObj</code>	Obtains a Cos object for a <code>PDEFont</code> .
<code>PDEFontGetWidths</code>	Gets the widths for a font object.
<code>PDEFontSubsetNow</code>	Subsets a <code>PDEFont</code> in a <code>CosDoc</code> .

When creating a new `PDEFont` with `PDEFontCreateWithParams`, the `PDEFont` may be represented as an embedded font (a `FontFile` value in PDF). To create a `PDEFont` that will be stored as an embedded font, the `FontFile` stream may be passed as `fontStm`, and the `len1`, `len2`, and `len3` parameters contain the `Length1`, `Length2`, and `Length3` values of the `FontFile`. The caller must close the `fontStm` after calling this `PDEFontCreateWithParams`. This method extends `PDEFontCreate` to support multibyte fonts.

PDEForm

A `PDEForm` is a `PDEElement` that contains a form XObject. Form XObjects are described in Section 4.9, “Form XObjects,” in the *PDF Reference, second edition, version 1.3*. A `PDEContent` may be obtained from a `PDEForm` to edit the form’s display list. The `PDEForm` methods include:

<code>PDEFormCreateFromCosObj</code>	Creates a new form from a Cos object.
<code>PDEFormGetContent</code>	Obtains a <code>PDEContent</code> object for a form.
<code>PDEFormGetCosObj</code>	Obtains a Cos object from a form.

PDEGroup

A **PDEElement** that specifies the beginning and ending of marked content in a PDF file. Any objects added to a **PDEGroup** object will be surrounded by the **BMC/EMC** marked content tags. The **PDEGroup** methods include:

PDEGroupCreate	Creates a PDEGroup object.
PDEGroupSetContent	Sets the PDEContent for a PDEGroup . The existing PDEContent is released by this method.

PDEImage

A **PDEImage** is a **PDEElement** that contains an image XObject or inline image. Image XObjects and inline images are described in the following sections in the [PDF Reference](#):

- Section 4.1, “Graphic Objects”
- Section 4.7, “External Objects”
- Section 4.8.6, “Inline Images”

You can associate data or a stream with an image via **PDEImageSetData** and **PDEImageSetDataStream** methods. **PDEImage** methods allow your plug-in to get and set properties of images, such as the color space and filters. Additional **PDEImage** methods include:

PDEImageCreate	Creates an image object from a stream or buffer of image data.
PDEImageCreateFromCosObj	Creates an image object from a Cos object.
PDEImageDataIsEncoded	Determines if image data is encoded.
PDEImageGetAttrs	Obtains attributes for an image.
PDEImageGetData	Gets an image’s data.
PDEImageGetFilterArray	Obtains the filter array for an image.
PDEImageIsCosObj	Determines if an image is represented by a Cos object.

PDEObject

PDEObject is the abstract superclass of PDFEdit classes. You can find the type of any object with the **PDEObjectGetType** method. You can then cast and apply that class' methods to the object. In addition, you can cast any of the PDFEdit objects to a **PDEObject** and use it anywhere a **PDEObject** is called for, such as in the **PDEObject** methods. **PDEAcquire** and **PDERelease** increment and decrement the reference counts of a **PDEObject**.

PDEObject methods include:

PDEAcquire	Increments the reference count for an object.
PDEAddTag	Adds an identifier—value pair to an object.
PDEGetTag	Obtains an object's value for a given client ID.
PDERemoveTag	Removes an object's value for a given client ID.

PDEPath

A **PDEPath** is a **PDEElement** that contains a path. It can have fill and stroke attributes. It also has graphics state attributes. The shape of a **PDEPath** can be used to represent a clipping path.

The **PDEPath** methods allow constructing a path from segments and setting its fill and stroke attributes. **PDEPath** methods include:

PDEPathAddSegment	Adds a segment to a path.
PDEPathCreate	Creates an empty path element.
PDEPathGetData	Obtains size of path data and, optionally, path data.
PDEPathSetPaintOp	Sets fill and stroke attributes of a path.

PDEPattern

A **PDEPattern** is a reference to a pattern resource used on a page. See Section 4.6 in the [PDF Reference](#), for information on patterns. **PDEPath** methods include:

PDEPatternCreate	Creates a pattern object.
-------------------------	---------------------------

PDEPatternGetCosObj	Obtains a Cos object corresponding to a pattern object.
----------------------------	---

PDEPlace

A **PDEPlace** is a **PDEElement** that marks a place on a page. In a PDF file, a place is represented by the **MP** or **DP** marked content operators.

Marked content is useful for adding structure information to a PDF file. For instance, a drawing program may want to mark a point with information, such as the start of a path of a certain type. Marked content provides a way to retain this information in the PDF file. A **DP** operator functions the same as the **MP** operator and, in addition, allows a property list dictionary to be associated with a place.

See Section 8.4.2, “Marked Content,” in the *PDF Reference*, for information on marked content and property lists.

A **PDEPlace** object allows marking a particular group of objects in a PDF file, rather than a place, with information.

PDEPlace methods include:

PDEPlaceCreate	Creates a place object.
PDEPlaceGetDict	Obtains the marked content dictionary for a PDEPlace .
PDEPlaceSetMCTag	Sets the marked content tag for a PDEPlace .

PDEPS

A **PDEPS** is a pass-through PostScript object. **PDEPS** methods include:

PDEPSCreate	Creates a PDEPS object.
PDEPSCreateFromCosObj	Creates a PDEPS object from a CosObj object.
PDEPSGetAttrs	Returns a PDEPS object's attributes.
PDEPSSetData	Sets the data for a PDEPS object.

PDEShading

A **PDEShading** is a **PDEElement** that represents smooth shading. **PDEShading** methods include:

PDEShadingCreateFromCosObj	Creates a smooth shading object.
PDEShadingGetCosObj	Gets the CosObj for a PDEShading .

PDESoftMask

A **PDESoftMask** is a reference to a SoftMask resource used to support transparency. **PDESoftMask** methods include:

PDESoftMaskAcquireForm	Acquires the XObject form of the soft mask.
PDESoftMaskGetBackdropColor	Gets the array of color values of the backdrop color.

PDEText

A **PDEText** object is a **PDEElement** that represents text. It is a container for text as show strings or as individual characters. Each subelement may have different graphics state properties. However, the same clipping path applies to all sub-elements of a **PDEText**. Also, the charpath of a **PDEText** object can be used to represent a clipping path.

Text consists of text runs, which are runs of characters in a PDF file with the same attributes. For instance, the text in the string before a **Tj** operator would constitute a text run or part of a text run. PDFEdit combines text from multiple **Tj** operators into a single text run, when possible.

NOTE: All text is in text runs. It's possible for a text run to be a single character.

Many **PDEText** methods take an index parameter to indicate a text position. These methods also take a **PDETextFlags** parameter to indicate whether a plug-in is accessing the text by characters or by text runs. If the plug-in uses the **kPDETextChar** flag, the index is the character offset from the beginning of the text element. This lets a plug-in ignore the fact that the **PDEText** consists of text runs. If a plug-in uses the **kPDETextRun** flag, the index is the index of the text run in the text element. Accessing text by text run is faster than accessing text a character at a time.

A plug-in can get and set attributes (such as the font or text matrix) of a **PDEText** object with **PDEText** methods. **PDEText** methods include:

PDETextAdd	Adds a character or text run to a <i>PDEText</i> object.
PDETextCreate	Creates an empty text object.
PDETextGetBBox	Obtains the bounding box of a character or text run.
PDETextGetGState	Obtains the graphics state of a character or text run.
PDETextGetNumChars	Obtains the number of characters in a text object.
PDETextGetTextState	Obtains the text state of a character or text run.
PDETextRunGetCharOffset	Obtains the character offset of the first character of a text run.
PDETextRunSetFont	Sets the font of a text run.
PDETextRunSetTextState	Sets the text state of a text run.
PDETextSplitRunAt	Splits a text run into two text runs.

PDEUnknown

A **PDEUnknown** is a **PDEElement** representing an unknown element. The **PDEUnknownGetOpName** method gets the operator name of an unknown operator.

PDEXGroup

A **PDEXGroup** is a reference to an XGroup resource used to support transparency. **PDEXGroup** methods include:

PDEXGroupAcquireColorSpace	Acquires the color space of the transparency group.
PDEXGroupCreate	Create a new XGroup of the given type (must be kPDEXGroupTypeTransparency).
PDEXGroupGetKnockout	Gets the knockout boolean value of the transparency group.

PDEXObject

A **PDEXObject** object is a **PDEElement** representing an arbitrary XObject. See Section 4.7, “External XObjects,” in the [PDF Reference](#), for information on XObjects. **PDEXObject** methods include:

PDEXObjectCreate	Creates a new PDEXObject from a Cos object.
PDEXObjectGetCosObj	Gets a Cos object corresponding to a PDEXObject .

NOTE: Use the appropriate methods for **PDEForm** and **PDEImage** objects. Do not use **PDEXObject** methods.

PDSysEncoding

A **PDSysEncoding** is a subclass of **PDEElement** that provides system encoding for a PDF file. **PDSysEncoding** methods include:

PDSysEncodingCreateFromBaseName	Create an encoding object from base name.
PDSysEncodingGetWMode	Returns the writing mode (0 for horizontal writing; 1 for vertical).

PDSysFont

A **PDSysFont** is a reference to a font installed on the host system. **PDSysFont** methods allow your plug-in to list the fonts available on the host system and to find a font on the system that matches a **PDEFont**, if it is present.

The **PDSysFont** and **PDEFont** classes are distinct. Your plug-in can create a **PDEFont** from a system font with the **PDEFontCreateFromSysFont** method. It can determine what system fonts are available using the **PDEnumSysFonts** and **PDFindSysFont** methods.

PDSysFont methods include:

PDEnumSysFonts	Enumerates all the system fonts.
PDFindSysFont	Finds the system font that matches given attributes.
PDSysFontGetAttrs	Obtains the attributes of a system font.
PDSysFontGetEncoding	Obtains encoding of single byte encoded system font.

PDSysFontGetName	Obtains the PostScript or TrueType styled name for a system font.
-------------------------	---

Introduction

PDF files are well known for representing the physical layout of a document; that is, the page markings that comprise the page contents. In addition, PDF versions 1.3 and beyond provide a mechanism for describing *logical structure* in PDF files. This includes information such as the organization of the document into chapters and sections, as well as figures, tables, and footnotes.

Further, PDF 1.4 and Acrobat 5 introduced *tagged PDF*, which is a particular use of structured PDF that allows page content to be extracted and used for various purposes such as reflow of text and graphics, conversion to file formats such as HTML and XML, and accessibility to the visually impaired.

This chapter describes how to create and access structure information in a PDF document. The PDSEdit methods in the core API provide access to this capability.

To use PDSEdit effectively in the plug-ins you write, you should understand how logical structure is represented in a PDF file. For details, see Section 8.4.3, “Logical Structure in PDF,” in the [PDF Reference](#).

Why Have Logical Structure?

Text on a page might clearly represent a paragraph or section to a reader, but prior to PDF 1.3 nothing in a PDF file represented such elements. When the original application generated the PDF file, information on the structure of a document’s content was lost. A PDF file could not distinguish paragraphs nor readily store paragraph style information.

Similarly, the core API prior to Acrobat 4.0 provided no way to extract paragraphs or other such text structures from a PDF file. The only text objects obtainable at the time were:

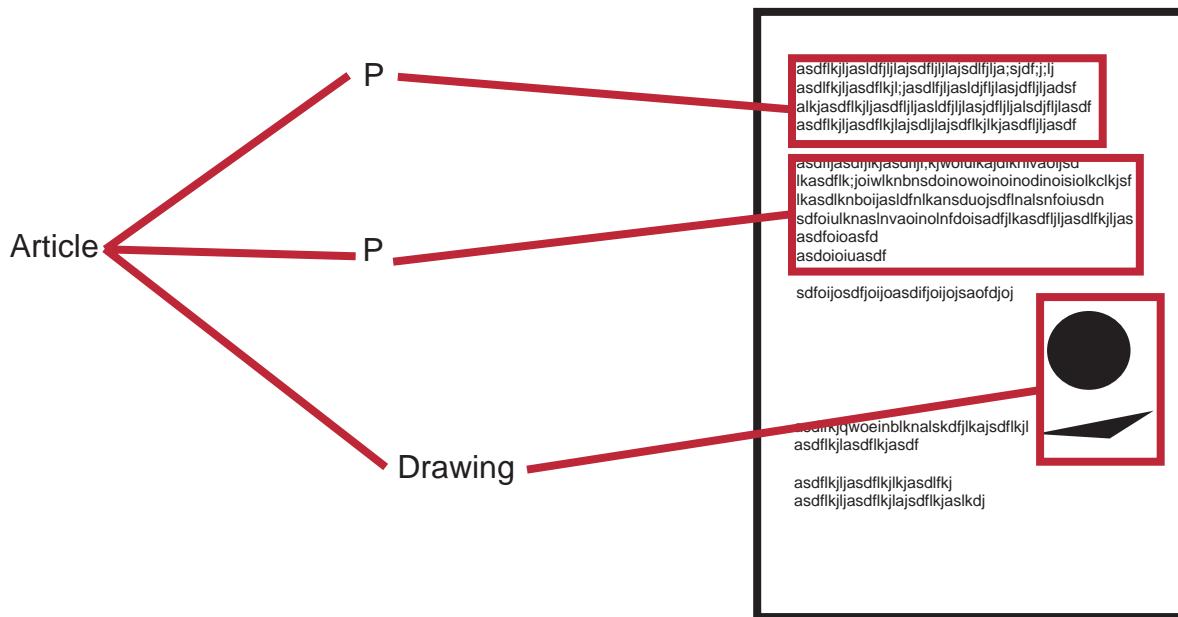
- Words and text selections, using **PDWord** and **PDTTextSelect** object methods provided in the PD layer of the core API.
- Text on a page, using **PDETText** object methods in the PDFFedit API (described in [Chapter 7](#)).

It became increasingly important for users be able to access the meta-information in the PDF document itself, without going to the original document and application--key reasons to have a portable document format in the first place.

[Figure 8.1](#) illustrates the relationship of some structural elements on a document page. It shows a structure hierarchy, namely, an article containing two paragraphs (P)

and a drawing. The PDSEdit methods provide the ability to represent this logical structure.

FIGURE 8.1 Structure in a Document



Other motivations behind PDF structure are:

- Accessibility for the sight-impaired to PDF documents. Readability of a document is considerably enhanced if cues are provided indicating the beginning and end of paragraphs or other natural groupings that the sighted take for granted. Structure information can indicate what is part of the content and what can be ignored. It can describe in words what a diagram shows.
- Metadata (data about the data), which is useful for purposes such as:
 - Tracking history to indicate document revisions
 - Tracking intellectual property
- Associating private data embedded in a PDF document with its content.
- Reflow of text and graphics.

The following sections explain how logical structure is provided.

Logical Structure in a PDF Document

PDF logical structure is layered on top of a document's page contents using a special markup language. HTML and XML use a similar layout for logical structure: text enclosed in a hierarchy of tags. In HTML, each component is wrapped with a set of

tags that define its structure. For example, the text of a top-level header begins with a `<h1>` tag and ends with a `</h1>` tag. PDF provides similar constructs with its *marked content* operators.

In fact, HTML logical structure can be preserved in a PDF document. The Web Capture feature introduced in Acrobat 4.0 allows converting HTML to PDF. Such PDF may optionally contain structure information from the HTML data. Acrobat 4.0 can generate bookmarks from this structure data.

The Structure Tree

Logical structure is independent of, though related to, the page content (that is, the actual marks on the page made by the marking operators).

In a PDF document, logical structure is represented by a tree of elements called a *structure tree*. There are pointers from the logical structure to the page contents, and vice versa. The structure tree provides additional capability to navigate, search, and extract data from PDF documents. By accessing a PDF document via its structure tree, for instance, you can obtain logically ordered content independently of the drawing order of the page contents.

Navigating a PDF Document

PDSEdit methods allow navigation of a document according to its structure. Bookmarks made from structure can go to an individual paragraph or a whole section, rather than just to a point on a page. PDSEdit also allows searching within structure elements, for example searching for a word within elements of a certain type, such as headings. It can be used to move around a document, to analyze its content, and to traverse its hierarchical structure.

Extracting Data From a PDF Document

The PDSEdit API allows you to extract portions of pages according to their context, such as all of the headings or tables. The extracted data can be used in different ways, such as summarizing document information, importing the data into another document, or creating a new PDF document.

Adding Structure Data To a PDF Document

Authoring applications create documents that can be converted to PDF. When the document is converted to PDF and viewed, Acrobat does not automatically add structure to the document.

You can add structural information to any PDF file with the PDSEdit API. Once a file has logical structure, PDSEdit allows you to use it.

Using pdfmark to Add Structure Data to PDF

Authoring applications may add structure **pdfmarks** to the PostScript language code generated when a document is printed. When the Acrobat Distiller application creates a PDF file from such PostScript code, it generates structure information in the PDF file from the **pdfmarks**. This approach requires the authoring application to add structure **pdfmarks** to the PostScript code it generates, or for some other application to generate the **pdfmarks**. See the [pdfmark Reference](#) for more information.

PDSEdit Classes

PDSEdit is organized around a set of classes representing structure components.

PDSTreeRoot

All logical structure information is in the *structure tree*, and the **PDSTreeRoot** is its root. There is at most one **PDSTreeRoot** in each document. **PDSTreeRoot** methods include:

PDSTreeRootCreateClassMap	Creates a PDSClassMap in a structure tree root. .
PDSTreeRootCreateRoleMap	Creates and sets a PDSRoleMap of a tree root.
PDSTreeRootGetKid	Gets the child at an array index in a structure tree root.

PDSElement

PDSElement is the basic building block of the structure tree. It represents PDF structural elements, which are nodes in a tree, defining a PDF document's logical structure. **PDSElement** methods include:

PDSElementAddAttrObj	Associates an attribute object with an element at the element's current revision value.
PDSElementGetClass	Gets the class name of an element.

PDSAttrObj

A **PDSAttrObj** represents a structure attribute object, which is a Cos dictionary or stream describing attributes associated with a **PDSElement**. The attribute's data may be application-specific, suiting the application that adds or extracts logical structure information. An attribute object can have a *revision number* to indicate whether other

applications have modified either the associated element or the element's contents since the application created or modified the element. **PDSAttrObj** methods include:

PDSAttrObjCreate	Creates a new attribute object.
PDSAttrObjCreateFromStream	Creates an attribute object from a Cos stream.

PDSMC

Portions of a page's contents may be wrapped with marked content operators. A **PDSMC** object represents this marked content. A tag and an optional property list may be associated with a **PDSMC**. **PDSMC** is identical to the PDFEdit class **PDEContainer**. **PDSMCs** may be nested. The **PDSMC** object has one method:

PDSMCGetParent	Gets the parent element of the specified marked content.
-----------------------	--

PDSOBJR

An object reference (**OBJR**) is a reference to a PDF object. A **PDSOBJR** object references an entire Cos dictionary or stream. The **PDSOBJR** object has one method:

PDSOBJGetParent	Gets the parent element of the specified marked content.
------------------------	--

PDSClassMap

The **PDSClassMap** (or *class map*) associates *class names* with a set of attribute objects. A structural element may have a list of names identifying the classes to which it belongs. Associated attributes are shared by all structural elements belonging to a given class. There is only one class map per document, associated with the **PDSTreeRoot**. **PDSClassMap** methods include:

PDSClassMapAddAttrObj	Adds an attribute object to a PDSClassMap .
PDSClassMapGetAttrObj	Gets the attribute object associated with a class name.
PDSClassMapRemoveClass	Removes a class from a PDSClassMap .

PDSRoleMap

Each structure element must have a structure type. The definition of such types is application-specific. In addition, PDF 1.3 defines a standard set of structure types for logical structure in PDF documents. The *role map* (**PDSRoleMap**) maps

application-specific element types to the standard element types that have a similar function. There is only one **PDSRoleMap** per document, associated with the **PDSTreeRoot**.

Relationship of PDSEdit and PDFEdit

A **PDSMC** object represents marked content in a page's contents stream. Specifically, it represents the content that is bracketed by **BMC/EMC** or **BDC/EMC** marked content operators.

In the PDFEdit API, **PDEContainer** objects are also defined as the content delimited by **BMC/EMC** or **BDC/EMC** operators.

This means that **PDSMC** and **PDEContainer** objects are *identical* and may be freely cast back and forth. You can use the PDFEdit API to create **PDEContainer** objects, then cast and use them as **PDSMC** objects.

Using the PDSEdit API: Examining Structure

Structure Tree Root

The starting point for access to PDF structure is the **PDSTreeRoot**, the *structure tree root*.

You can obtain a **PDDoc**'s **PDSTreeRoot** by calling [PDDocGetStructTreeRoot](#). The return value indicates whether the document has any structure at all. A document has structure if and only if it has a structure tree root and, hence, a structure tree.

The structure tree root may contain a role map, which can help you identify elements that serve common uses in the structure. You should call [PDSTreeRootGetRoleMap](#) to get the tree root's role map.

The structure tree root may also hold a class map, which associates sets of attributes with elements in the structure tree. You can get the class map with the [PDSTreeRootGetClassMap](#) method.

Structure Elements

The actual structure elements or **PDSElements** of a document are grouped into subtrees that are attached to the structure tree root. Each subtree's root is itself a **PDSElement** to which other **PDSElements** may be attached.

Call [PDSTreeRootGetNumKids](#) to get the number of elements attached to the tree root. To obtain each of these elements, use the [PDSTreeRootGetKid](#) method.

This example:

- Gets the structure tree root
- Checks if the tree root has children
- Gets the last child

```
PDSTreeRoot treeRoot;
if (!PDDocGetStructTreeRoot(pdDoc, &treeRoot))
    return; /* no structure tree */
ASInt32 numKids;
if ((numKids = PDSTreeRootGetNumKids(treeRoot)) == 0)
    return; /* no kids */
PDSElement listElement;
/* get last kid */
PDSTreeRootGetKid (treeRoot, numKids - 1, &listElement);
```

Traversing Elements in a Subtree

A **PDSElement** may have other **PDSElements** attached to it to form a subtree.

PDSElementGetKid is perhaps the most important access method in PDSEdit. A structural element—unlike the structure tree root—can have *several different kinds of children*:

- Another element (**PDSElement**)
- Marked content (**PDSMC**)
- A reference to an entire PDF object (**PDSOBJR**)

To allow for this, **PDSElementGetKid** returns a parameter that indicates the child's type.

If the returned child is a **StructElem** or **PDSOBJR**, the child is stored in the parameter **cosObjKid**; if the return value is **PDSMC**, the child is stored in the parameter **pointerKid**. This method optionally provides the page on which an object or marked content child is located.

Suppose you want to traverse the entire structure tree, looking for an element or a set of elements that satisfy some search criteria. The **PDSTreeRootGetNumKids** and **PDSTreeRootGetKid** methods allow it to get the elements in the root of the structure tree. You can then use **PDSElementGetNumKids** and **PDSElementGetKid** to traverse the children of each element it encounters. Since the structure is a tree, it lends itself to recursive handling.

If a child is a **PDSElement**, it may have children of its own, which you can examine as indicated above. Given a **PDSElement**, you can use **PDSElement** class methods to determine the type (also called its *tag name*), title, and attributes.

If the child is a **PDSOBJR**, it can be a reference to a Cos dictionary or Cos stream object on a page. For instance, the object referenced may be an XObject representing an image. Handling of object references in the structure tree typically is application-specific.

If the child is a marked content element, you can use PDSEdit methods to examine it. For example, it can see what text the marked content element contains or copy the content to another document.

NOTE: A marked content object is referred to as type **PDSMC** in the PDSEdit API, and this is actually a synonym for the PDSEdit class **PDEContainer**. Be sure to cast objects appropriately and observe the conventions for acquiring and releasing PDSEdit objects.

For more information, see [Chapter 7, “PDSEdit—Creating and Editing Page Content.”](#)

The example below:

- Gets the children of an element
- Looks for a marked content element
- Gets the marked content

In this example, the element contains a list of other elements as children, and marked content may be attached to these children.

```
/* Get the number of kids*/
ASInt32 listLength = PDSElementGetNumKids(listElement);
if (!listLength)
    return; /* no kids */
/* Extract information from each kid */
ASInt32 i; for (i = 0; i < listLength; i++) {
    CosObj cosObjKid, cosObjKid1;
    PDSMC mcKid;
    ASAtom kidType = PDSElementGetKid(listElement, i, &cosObjKid,
        (void**)&mcKid, NULL);
    if (kidType != ASAtomFromString("StructElem"))
        continue; /* Not a structure element */
    /* Look at first kid of structure element */
    kidType = PDSElementGetKid((PDSElement)cosObjKid, 0, &cosObjKid1,
        (void**)&mcKid, NULL);
    if (kidType != ASAtomFromString("MC"))
        continue; /* Not an MC */
    /* Got the MC. Get its content. */
    PDEContainer pdeContainer = (PDEContainer)mcKid;
    PDEContent markedContent = PDEContainerGetContent(pdeContainer);
    /* Process the marked content */
    ...
}
```

Object Attributes

An element may have attributes representing application-specific information.

Attributes are Cos dictionaries or Cos streams. You can use

[PDSElementGetNumAttrObjs](#) and [PDSElementGetAttrObj](#) to iterate through the attribute objects attached to an element. You can filter these attribute objects according to their revision number (as mentioned in [“PDSEdit Classes” on page 146](#))

by comparing the returned revision number from `PDSElementGetAttrObj` with the revision number of the element returned by `PDSElementGetRevision`.

Once you have an attribute object, you may examine the object using standard Cos-level methods. Note that each attribute object may contain zero or more attributes. The attributes of an element are the union of the attributes given by all the attribute objects.

Other Object Characteristics

In addition to attributes, an object can have other characteristics associated with it, such as a title or ID. Use `PDSElement` accessor functions such as `PDSElementGetTitle` to get this information.

Element Types and the Role Map

A structure element is represented by a Cos dictionary. In this dictionary, the `Type` key always has the value `StructElem`. There also is a required `Subtype` key, and this key's value indicates what kind of structure element it has. `PDSElementGetType` returns an element's type.

Although a plug-in is free to define whatever element types it wishes, PDF defines a standard list of element types. The role map associates a user-defined element type with one of the standard types. There is only one role map in a PDF document, as previously noted in “[PDSRoleMap](#)” on page 147.

Given an element's type, you can consult the structure tree root's role map via `PDSRoleMapDoesMap` and `PDSRoleMapGetDirectMap` to find any standard roles assigned to an element of this type. For example, you can find out whether an element is some sort of section or table element in a document. A plug-in can use a well-made role map to help make sense of a document.

Classes and the Class Map

A set of attributes may be associated with a class, and an element may belong to one or more classes. If an element belongs to a class, it has all the attributes associated with the class. The class map contains this information. There is only one class map in a PDF document, as previously noted in “[PDSClassMap](#)” on page 147.

Use `PDSElementGetNumClasses` to get the number of classes an element belongs to, and call `PDSElementGetClass` to obtain each class to which the element belongs. Then you can call `PDSClassMapGetNumAttrObjs` to get the number of attributes associated with the class and `PDSClassMapGetAttrObj` to obtain each attribute in the class.

Using the PDSEdit API: Creating Structure

Structure Tree Root

Before adding structure to a PDF document, you must first create a structure tree root if the **PDDoc** doesn't already have one. To do this, call **PDDocGetStructTreeRoot** to determine if the **PDDoc** has a structure tree root. Call **PDDocCreateStructTreeRoot** to create one.

To add structure elements to the tree root, use **PDSTreeRootInsertKid**.

Structure Elements

Creating structure using PDSEdit is mainly a process of creating elements with **PDSElementCreate**, connecting them using **PDSElementInsertKid**, and attaching the resulting subtrees to the structure tree root using **PDSTreeRootInsertKid**. You can also construct the tree by adding **PDSElements** to the tree root, then adding children to these **PDSElements**. Or you can do a combination of these.

Create a structural element by calling **PDSElementCreate**. You must set its type with **PDSElementSetType** before doing anything else with it. You may optionally set an element's ID, title, and alternate text representation with the respective methods, **PDSElementSetID**, **PDSElementSetTitle**, and **PDSElementSetAlt**.

The **PDSClassMapAddAttrObj** method adds an attribute object to an element. **PDSElementAddClass** adds a class to an element.

The example below:

- Creates a **PDDoc**'s structure tree root if one doesn't exist
- Adds a structure element to it

```
PDSTreeRoot treeRoot;
if (!PDDocGetStructTreeRoot(pdDoc, &treeRoot))
    PDDocCreateStructTreeRoot(pdDoc, &treeRoot);
PDSElement listElement;
PDSElementCreate(pdDoc, &listElement);
PDSElementSetType(listElement, ASAtomFromString("L"));
/* list element */
#define TEXT_LIST "Text element list"

PDSElementSetTitle(listElement, (const ASUms8*)TEXT_LIST,
    strlen(TEXT_LIST));
PDSTreeRootInsertKid(treeRoot, listElement,
    PDSTreeRootGetNumKids(treeRoot));
```

Adding Marked Content to an Element

Use PDFEdit methods to create or obtain marked content to add to a structural element. You can then cast a **PDEContainer** object to **PDSMC** to use it with the PDSEdit API. It can add marked content to a structure element with **PDSElementInsertMCAsKid**.

You can add a reference to marked content to only *one* structure element. A **PDSMC** can have only one parent, because of the implementation: marked content points to its parent. If you need to refer to marked content in more than one place, it can refer to the structure element that has the **PDSMC** as a child rather than referring directly to the **PDSMC**.

This example:

- Creates a marked content container
- Adds a text element
- Casts the container as a **PDSMC**
- Adds the container to a structure element

The example sets the container tag to "**LI**" so that it is the same as the subtype of the element containing the marked content. This allows using the **New Bookmarks from Structure...** feature introduced with Acrobat 4.0.

```

PDPage pdPage;
pdPage = PDDocAcquirePage(pdDoc, thisPage);
CosObj pageCos = PDPageGetCosObj(pdPage);
PDEContainer pdeContainer = PDEContainerCreate(ASAtomFromString("LI"),
    NULL, false);
PDEContent textContent = PDEContentCreate();
PDEContentAddElem(textContent, k, PDEBeforeFirst, pdeElement);
/* pdeElement is some text element obtained earlier */
PDEContainerSetContent(pdeContainer, textContent);
/* Create structure element; put container in that element as an MC. */

PDSElement listItemElement;
PDSElementCreate(pdDoc, &listItemElement);
#define TEXT_ELEMENT "A text element"
PDSElementSetType(listItemElement, ASAtomFromString("LI"));
/* list item */
PDSElementSetTitle(listItemElement, (const ASUns8*)TEXT_ELEMENT,
    strlen(TEXT_ELEMENT));
/* Put marked content into element */
PDSElementInsertMCAsKid(listItemElement, pageCos, (PDSMC)pdeContainer, 0);
PDPageRelease(pdPage);

```

Adding an Object Reference to an Element

You can add a PDF object reference to an element with **PDSElementInsertOBJAsKid**.

The object reference can be added to only one structure element. A **PDSOBJR** can have only one parent, because of the implementation: an object points to its parent. If you need to refer to an object in more than one place, you should refer to the structure element that has the **PDSOBJR** as a child, rather than referring directly to the **PDSOBJR**.

This example adds a **PDSOBJR** to a structure element.

```
PDSElement XObjectElement;
PDSElementCreate(pdDoc, &XObjectElement);
PDSElementSetType(XObjectElement, ASAtomFromString("LI"));
/* Insert the reference to the CosObj obj we obtained somewhere else */
PDPage pdPage;
pdPage = PDDocAcquirePage(pdDoc, thisPage);
CosObj pageCos = PDPageGetCosObj(pdPage);
/* Add the object reference */
PDSElementInsertOBJAsKid(XObjectElement, pageCos, obj, 0);
PDPageRelease(pdPage);
```

Class Map

You can create a class map in the structure tree root with **PDSTreeRootCreateClassMap**, which provides the class map created. You can get an existing class map in a structure tree with **PDSTreeRootGetClassMap**. There is only one class map in a PDF document.

To add an attribute for a class to the class map, use **PDSClassMapAddAttrObj**. If the class does not already exist in the class map, it is created and the attribute added to it. **PDSClassMapRemoveClass** removes a given class from the class map. **PDSClassMapRemoveAttrObj** removes an attribute from a given class in the class map.

Role Map

PDSTreeRootCreateRoleMap creates a role map in a structure tree and provides the newly-created role map. **PDSTreeRootGetRoleMap** obtains an existing role map. A PDF document has only one role map.

To specify that a user-defined element type has the role of a standard element type, call **PDSRoleMapCopy**. For more information, see the section entitled “Marked PDF” in the document, *PDF: Changes From Version 1.3 to 1.4*.

The Cos layer provides access to the low-level object types and file structure used in PDF files. PDF documents are trees of *Cos objects*. Cos objects represent document components such as bookmarks, pages, fonts, and annotations, as described in Section 3.6, “Document Structure,” in the [PDF Reference](#).

Unlike using the AV and PD layer methods, using Cos layer methods improperly could result in an invalid PDF file. Therefore, you should not use Cos methods unless necessary, for example to add private data to portions of a PDF file that cannot be accessed in other ways.

This chapter describes the Cos object types, data structures, and methods. See the [Acrobat Core API Reference](#) for detailed information on each method. See Section 3.4, “File Structure,” and Section 3.6, “Document Structure,” in the [PDF Reference](#), for details on file structure and Cos objects. The [Acrobat Plug-In Tutorial](#) also includes a chapter on using Cos object methods.

Cos Objects: Direct and Indirect

PDF files contain several types of Cos objects: booleans, numbers, strings, names, arrays, dictionaries, and streams, plus a special `null` object.

Your plug-in can create objects of any of these types either as *direct* objects or *indirect* objects; the choice is specified as a parameter to the method that creates the object. For details on direct and indirect objects, see Section 3.2.9, “Indirect Objects,” in the [PDF Reference](#).

When a direct object is created, the object itself is returned. As a result, a direct object can only be attached to *one* other Cos object at a time; it cannot, for example, be shared by two different dictionaries.

When an indirect object is created, something equivalent to a pointer to the object is returned. As a result, an indirect object can be attached to multiple places in a PDF file simultaneously; it can, for example, be shared by two different dictionaries.

Attaching a Cos object to another is referred to as putting it into a *container* object. Core API methods that put objects into container objects raise an exception if the object to be put is a direct object that already is contained in another object.

Direct booleans, integers, fixed numbers, and names need not be destroyed when they are no longer needed. Other object types (and indirect objects of these types) should be destroyed when they are no longer needed.

File structure

A PDF file consists of four sections:

- A one-line header specifying the PDF version.
- A body, which is a sequence of objects representing a PDF document.
- A cross-reference table containing information allowing access to indirect objects in the file.
- A trailer containing information on certain special objects in the file.

There is one entry in the cross-reference table for each indirect object in a file; the entry specifies the byte offset of the object from the beginning of the file. When a file is opened, if Acrobat determines that the offsets are incorrect (indicating that the file has been damaged in some manner), it attempts to rebuild the cross-reference table as described in Appendix C.1 in the *PDF Reference, second edition, version 1.3*.

Multiple files may be open simultaneously. Each open file is represented by a document pointer, and all indirect objects must be associated with a document. However, objects belonging to one document *cannot be stored in objects in another document*. The Cos layer uses [ASStm](#) objects to access a file's contents.

Cos Objects in the Core API

In the Acrobat core API, there are two defined objects:

- [CosDoc](#), which represents an entire PDF file.
- [CosObj](#), which represents all the individual object types. There are various methods to create the different types of Cos objects, as well as getting and setting their values.

Each [CosObj](#) can be specified as being one of the other supported types:

[CosArray](#), [CosBoolean](#), [CosDict](#), [CosFixed](#), [CosInteger](#), [CosName](#), [CosNull](#), [CosStream](#), and [CosString](#).

The Cos layer provides methods to create and modify objects of each of the supported types, as well as methods to read and write objects to and from a file. Additional utility methods include those to get the root node of the tree of objects representing a PDF document, and the info dictionary for a PDF document.

CosDoc

A **CosDoc** object is the Cos layer representation of an entire PDF file. See [Appendix B](#) for an overview of PDF document structure. See Section 3.6.1, "Document Catalog," in the [PDF Reference](#), for a description of the catalog dictionary.

CosDocClose	Closes a Cos document.
CosDocCreate	Creates an empty Cos document.
CosDocGetRoot	Gets a document's root Cos object, the catalog dictionary
CosDocOpenWithParams	Opens a Cos document.
CosDocSaveToFile	Sets a single element in an array.

CosObj

A **CosObj** is a general object in a PDF file, which may be of any Cos object type.

The Cos layer provides several methods that are not specific to any particular object. Several methods are available to manipulate a Cos object and include:

CosObjCopy	Copies a CosObj from one document to another (or the same document).
CosObjGetDoc	Gets the CosDoc containing the object (indirect or non-scalar objects only).
CosObjGetType	Gets an object's type.
CosObjIsIndirect	Tests whether an object is indirect. See Section 3.2.9 in the PDF Reference for details on indirect objects.

CosArray

Cos arrays are one-dimensional collections of objects accessed by a numeric index. Array indexes are zero based. An array's elements may be any combination of the Cos data types.

The **CosArray** methods include:

CosArrayGet	Gets a single element from an array.
-----------------------------	--------------------------------------

CosArrayInsert Inserts an element into an array.

CosArrayPut Sets a single element in an array.

CosNewArray Creates an array.

CosBoolean

Cos boolean objects can have a value of **true** or **false**. The **CosBoolean** methods include:

CosBooleanValue Gets the value of the specified boolean object.

CosNewBoolean Creates a new boolean object associated with the specified document and having the specified value.

CosDict

A Cos *dictionary* is an associative table whose elements are pairs of objects:

- The first element of a pair is the key, which is always a name object, a sequence of characters beginning with the forward slash (/) character.
- The second element is the Cos object representing the value.

See Section 3.2.6 in the [PDF Reference](#) for details.

The **CosDict** methods include

CosDictGet Gets the value of a dictionary key.

CosDictPut Sets the value of a dictionary key.

CosDictRemove Removes a key-value pair from a dictionary.

CosNewDict Creates a dictionary.

CosFixed

Fixed numbers may only be in decimal format. See Section 3.2.2 in the [PDF Reference](#), for details. The **CosFixed** methods include:

CosFixedValue Gets the value of the specified fixed number object.

CosNewFixed	Creates a new fixed number object associated with the document, having the specified value.
--------------------	---

CosInteger

Integers may be specified by signed or unsigned constants. See Section 3.2.2 in the [PDF Reference](#), for details. **CosInteger** methods include:

CosIntegerValue	Gets the integer value of the specified number object.
CosNewInteger	Creates a new integer object associated with the document, having the specified value.

CosName

A *name* is a sequence of non-white space characters. In code, a name is preceded by the forward slash (/) character indicating that it is a string literal, for example: /`AName`.

See Section 3.2.4 in the [PDF Reference](#), for details. The **CosName** methods include:

CosNameValue	Gets the value of the specified name object.
CosNewName	Creates a new name object associated with the document, having the specified value.

CosNull

There is only one **NULL** object, which is used to fill empty or uninitialized positions in arrays or dictionaries. See Section 3.2.8 in the [PDF Reference](#) for details.

CosNewNull is a method that gets a **NULL** Cos object.

CosStream

A *stream* is a sequence of characters that can be read a portion at a time. Streams are used for objects with large amounts of data, such as images, page content, or private data a plug-in creates. A stream consists of these elements, which are listed in their relative order in the stream object, starting at the beginning.

See Section 3.2.7 in the [PDF Reference](#), for a description of the stream object. **CosStream** methods include:

CosNewStream	Creates a new Cos stream, using data from an existing ASStm .
CosStreamLength	Gets a stream's length.
CosStreamOpenStm	Creates a new, non-seekable ASStm for reading data from a Cos stream.
CosStreamPos	Gets the byte offset of the start of a Cos stream's data in the PDF file.

CosString

A *string* is a sequences of characters, enclosed in parentheses. See Section 3.2.3 in the [PDF Reference](#) for details. **CosString** methods include:

CosNewString	Creates and returns a new Cos string object.
CosStringValue	Gets the value of Cos string object, and the string's length.

Encryption/Decryption

The Cos layer provides methods to encrypt and decrypt data in arbitrary memory blocks. The encryption and decryption uses Acrobat's built-in algorithm (RC4 from RSA Data Security, Inc.) and a key that can be specified. Methods include:

CosDecryptData	Decrypts data in a buffer using the specified encryption key.
CosEncryptData	Encrypts data in a buffer using the specified encryption key.

NOTE: These methods are not available in the Adobe PDF Library.

10 Handlers

Plug-ins can add new types of tools, annotations, actions, file systems, and so on, thereby expanding the number of object types that Acrobat supports. To accomplish this, plug-ins provide a collection of callback routines called *handlers*. Handlers perform the necessary functions for the objects, such as creating and destroying them, drawing, and handling mouse clicks, keyboard events, and other events as appropriate for their objects.

NOTE: These types of handlers are distinct from *exception handlers* (see [Chapter 12, “Handling Errors”](#)).

To add a new handler, a plug-in must write the callback routines, create the appropriate data structure containing the callbacks and other data, and pass the structure to Acrobat using the appropriate API method. Subsequently, Acrobat automatically calls the correct callbacks when it encounters an object of the type handled by the handler.

This chapter describes several types of handlers and shows the which data structures, callbacks and methods are involved in creating them.

It is possible to “subclass” existing handlers or to create entirely new types of handlers. For example, a plug-in could subclass the built-in text annotation handler by adding the ability to hide annotations. To accomplish this, the plug-in would :

- Obtain the built-in text annotation handler structure (using [AVAppGetAnnotHandlerByName](#)).
- Copy the structure before modifying it (not modifying the original).
- Replace the handler’s **Draw** callback with one that calls the built-in **Draw** callback (obtained from the structure) if annotations are visible, or simply return without drawing anything if annotations are hidden.
- Register the new handler (using [AVAppRegisterAnnotHandler](#) with a new type).

If a handler requires more data than provided in the predefined structures described in this section, you can append additional data to the predefined structures. To do this, create a new structure type with the predefined structure as its first member and the additional data as subsequent members. Before passing the expanded structure to the Acrobat method, cast the structure to the predefined structure type. Upon return of the structure from Acrobat, re-cast the structure to its expanded type to access the appended data.

Each handler data structure contains a **size** field, which specifies the structure’s size. This field provides future compatibility. Different versions of the structure have different sizes, allowing Acrobat to determine which version your plug-in was written to use.

NOTE: Regardless of whether your plug-in adds data to the predefined structures, it always must pass the size of the predefined structure (rather than the size of its expanded structure) in the `size` field.

Action Handlers

Support for new action types can be added by defining and registering an *action handler*. The Acrobat Weblink plug-in uses this ability to add support for URL links.

To add a new action type, a plug-in must provide a set of callbacks, specify them in the `AVActionHandlerProcs` structure, and call `AVAppRegisterActionHandler` to register them (see the *Acrobat Core API Reference* for details). The callbacks include ones that:

- Perform the action, such as setting the view to that specified by the destination (`AVActionPerformProc`).
- Allow the user to set the action's properties (necessary only if any properties can be set). (`AVActionDoPropertiesProc`).
- Initialize an action's dictionary with default values. (`AVActionFillActionDictProc`).
- Display a string containing brief instructions for the action. (`AVActionGetInstructionsProc`).
- Display various text strings to be used in dialogs. (`AVActionGetButtonTextProc`, `AVActionGetStringOneTextProc`, `AVActionGetStringTwoTextProc`).
- Copy the action (`AVActionCopyProc`).

For details on each of the callbacks in an action handler, see the description of `AVAppRegisterActionHandler` in the *Acrobat Core API Reference*.

Annotation Handlers

Support for new annotation types in the Acrobat viewer can be added by defining and registering an annotation handler. The Acrobat Movie plug-in, for example, uses this to support video annotations.

To add a new annotation type, a plug-in must provide a set of callbacks, specify them in the `AVAnnotHandler` structure, and register them with `AVAppRegisterAnnotHandler` (see the *Acrobat Core API Reference* for details). The callbacks include ones that:

- Draw the annotation (`AVAnnotHandlerDrawProc`).
- Handle mouse clicks in the annotation (`AVAnnotHandlerDoClickProc`).

- Control the cursor shape when the cursor is over the annotation ([AVAnnotHandlerAdjustCursorProc](#)).
- Determine whether or not a specified point is within the annotation ([AVAnnotHandlerPtInAnnotViewBBoxProc](#)).
- Return the rectangle bounding the region the annotation occupies ([AVAnnotHandlerGetAnnotViewBBoxProc](#)).
- Highlight (unhighlight) the annotation when it is added to (removed from) the selection ([AVAnnotHandlerNotifyAnnotAddedToSelectionProc](#), [AVAnnotHandlerNotifyAnnotRemovedFromSelectionProc](#)).
- Return the annotation's subtype ([AVAnnotHandlerGetTypeProc](#)).
- Get the annotation's layer ([AVAnnotHandlerGetLayerProc](#)). See “[Page View Layers](#)” on page [48](#) for related information.

AVCommand Handlers

Introduced in Acrobat 5.0, an **AVCommand** represents an action that the user can perform on the current document or the current selection in the current document. **AVCommands** are exposed to Acrobat through **AVCommand** handlers. A plug-in can add new command types in the Acrobat viewer by defining and registering an **AVCommand** handler. Commands can be executed interactively, programmatically, or through batch processing.

Creating an AVCommand Handler

AVCommand handlers consist of a series of callbacks contained in the **AVCommandHandlerRec** structure (see `AVExpt.h`).

To implement a command handler with minimal functionality, a plug-in should

- Initialize an instance of the **AVCommandHandlerRec** structure.
 - Allocate memory for the structure.
 - Fill in the size field.
 - Implement the **Work**, **Cancel**, and **Reset** callbacks.
- Register the **AVCommandHandlerRec** structure with Acrobat using [AVAppRegisterCommandHandler](#).

This is shown in the following example:

```
static AVCommandHandlerRec gAVCmdHandler;
const char *kCmdName = "MinimalCommand";
static ACCB1 AVCommandStatus ACCB2 DoWorkImpl (AVCommand cmd)
{
    // Minimal AVCommand. Does nothing.
    return kAVCommandDone;
}
```

```

void InitializeCommandHandler()
{
    memset (&gAVCmdHandler, 0, sizeof(AVCommandHandlerRec));
    gAVCmdHandler.size = sizeof(AVCommandHandlerRec);
    gAVCmdHandler.Work = ASCallbackCreateProto (AVCommandWorkProc,
                                                DoWorkImpl);
    AVAppRegisterCommandHandler (ASAtomFromString(kCmdName),
                                &gAVCmdHandler);
}

```

This procedure implements a valid **AVCommand** that plug-in clients can access through the **AVCommand** methods. For details on how clients can invoke the **AVCommands**, see “[Invoking AVCommands Programmatically](#)” on page 72.

Exposing AVCommands to the Batch Framework

Acrobat builds the list of commands that users see in the **Batch Sequences** and **Batch Edit Sequence** dialogs from an internal list of **AVCommands** referred to as the *global command list*.

Adding a Handler to the Global Command List

To expose a command to the batch framework, the **AVCommand** handler first must add an instance of the command to this global list using the **AVAppRegisterGlobalCommand** method.

```

AVCommand cmd = AVCommandNew (ASAtomFromString(kCmdName));
AVAppRegisterGlobalCommand (cmd);

```

Although this step can be performed at any time once the command handler has been registered, handlers commonly register commands from within the **AVCommandRegisterCommandsProc** callback (of the **AVCommandHandlerRec** structure), for example,

```

static ACCB1 void ACCB2 RegisterCommandsImpl (ASAtom handlerName)
{
    ASAtom cmdName = ASAtomFromString(kCmdName);
    AVCommand cmd;
    if (NULL == AVAppFindGlobalCommandByName (cmdName)) {
        cmd = AVCommandNew (cmdName);
        if (cmd)
            AVAppRegisterGlobalCommand (cmd);
    }
}

```

Supporting Properties

When building a list of batchable commands, Acrobat iterates through its internal command list, querying each command for the “CanBatch” and “GroupTitle” properties. To be exposed through the batch framework user interface, a command must support these properties (that is, return **true** and a valid **ASText** object, respectively, when Acrobat queries them).

To accomplish this, the **AVCommand** handler must implement the **GetProps** callback of the **AVCommandHandlerRec** structure.

If an **AVCommand** supports these properties, Acrobat queries a number of additional properties as the user interacts with the batch framework. Of these additional properties, only two are required: "Title" and "Generic Title". A command must provide the title strings that will be displayed in the **Batch Sequences** and **Batch Edit Sequence** dialogs. See the [Acrobat Core API Reference](#) for a complete description of the various **AVCommand** properties.

```
const char *kCmdTitle = "Command Title";
const char *kGroupTitle = "Group Title";
const char *kCmdGenericTitle = "Generic Title";

static ACCB1 AVCommandStatus ACCB2 GetPropsImpl (AVCommand cmd,
ASCab props)
{
    ASBool doItAll = false;
    ASText text;
    if (ASCabNumEntries( props ) == 0)
        doItAll = true;
    if (doItAll || ASCabKnown (props, kAVCommandKeyGroupTitle))
    {
        // Create a new text object and insert it into the ASCab
        text = ASTextNew();
        ASTextSetEncoded (text, kGroupTitle, (
            ASHostEncoding)PDGetHostEncoding());
        ASCabPutText( props, kAVCommandKeyGroupTitle, text);
    }
    if (doItAll || ASCabKnown (props, kAVCommandKeyCanBatch))
        ASCabPutBool (props, kAVCommandKeyCanBatch, true );
    if (doItAll || ASCabKnown (props, kAVCommandKeyGenericTitle))
    {
        // Create a new text object and insert it into the ASCab
        text = ASTextNew();
        ASTextSetEncoded (text, kCmdGenericTitle,
            (ASHostEncoding)PDGetHostEncoding());
        ASCabPutText (props, kAVCommandKeyGenericTitle, text);
    }
    if (doItAll || ASCabKnown (props, kAVCommandKeyTitle))
    {
        // Create a new text object and insert it into the ASCab
        text = ASTextNew();
        ASTextSetEncoded (text, kCmdTitle,
            (ASHostEncoding)PDGetHostEncoding());
        ASCabPutText (props, kAVCommandKeyTitle, text);
    }
}
```

At this point, the user will be able to use the command in batch sequences.

File Format Conversion Handlers

With Acrobat 5.0 and higher, a plug-in can add *file conversion handlers* to Acrobat (but not Acrobat Reader) for converting:

- To PDF from another file format (import)
- From PDF to another file format (export)

To add a new file conversion handler, a plug-in must provide a set of callbacks, specify them in the [AVConversionToPDFHandler](#) or [AVConversionFromPDFHandler](#) structures, and call [AVAppRegisterToPDFHandler](#) or [AVAppRegisterFromPDFHandler](#) to register them (see the *Acrobat Core API Reference* for details). It must specify the types of files it can convert and whether it can perform synchronous conversion (required for the handler to be accessible from the batch framework). Upon registration, the conversion handlers are automatically added to the respective **Open...** and **Save As...** dialogs.

The callbacks include ones that:

- Provide default settings for the conversion ([AVConversionDefaultSettingsProc](#)).
- Provide conversion parameter information ([AVConversionParamDescProc](#)).
- Display a settings dialog ([AVConversionSettingsDialogProc](#)).
- Convert a non-PDF file to or from a PDF file ([AVConversionConvertToPDFProc](#) or [AVConversionConvertFromPDFProc](#)).

The Acrobat 5.0 SDK includes the `FileConversion` plug-in example that converts files to PDF from other file formats.

File Specification Handlers

A file specification handler converts between a [PDFFileSpec](#) and an [ASPathName](#). Each file specification handler works only with a single file system, which the handler specifies.

To create a new file specification handler, a plug-in or application must provide callbacks that:

- Convert an [ASPathName](#) to a [PDFFileSpec](#). It is called by [PDFFileSpecNewFromASPath](#).
- Convert a [PDFFileSpec](#) to an [ASPathName](#).

See the description of [PDRegisterFileSpecHandlerByName](#) in the *Acrobat Core API Reference*, for details on each of the callbacks in a file specification handler.

Security Handlers

The issue of document security is complex enough that it is discussed in a separate chapter. For details on security handlers, see [Chapter 11, “Document Security.”](#)

Selection Servers

A *selection server* allows the selection of a certain type of data, such as annotations, text, or graphics. Plug-ins can create new selection servers to allow the selection of types of data not already supported. To add a new selection server, a plug-in must provide a set of callbacks, specify them in the [AVDocSelectionServer](#) data structure, and register them using [AVDocRegisterSelectionServer](#).

The callbacks include ones that:

- Return the selection type serviced by the handler ([AVDocSelectionGetTypeProc](#)).
- Highlight or unhighlight a selection ([AVDocSelectionHighlightSelectionProc](#)).
- Handle key presses ([AVDocSelectionKeyDownProc](#)).
- Delete the selection ([AVDocSelectionDeleteProc](#)).
- Cut the selection to the clipboard ([AVDocSelectionCutProc](#)).
- Copy the selection to the clipboard ([AVDocSelectionCopyProc](#)).
- Paste the selection from the clipboard ([AVDocSelectionPasteProc](#)).
- Select all data of the type they handle ([AVDocSelectionSelectAllProc](#)).
- Enumerate the items in the current selection ([AVDocSelectionEnumSelectionProc](#)).
- Scroll the view so that the current selection is available ([AVDocSelectionShowSelectionProc](#)).
- Determine whether or not the “Properties” menu item is enabled ([AVDocSelectionCanPropertiesProc](#)).
- Display the properties dialog for the selection, if the selection type has a properties dialog ([AVDocSelectionPropertiesProc](#)).

For a complete list of the callbacks in a selection server, see the description of [AVDocSelectionServer](#) in the [Acrobat Core API Reference](#). The `SelectionServer` sample plug-in in the Acrobat SDK shows an example of a selection server.

Tools

To add a new *tool*, a plug-in must provide a set of callbacks, specify them in the [AVTool](#) data structure, and register them using [AVAppRegisterTool](#)..

The callbacks include ones that:

- Activate the tool; that is, do whatever is necessary when the tool is selected ([ActivateProcType](#))
- Deactivate the tool; that is, do whatever is necessary when another tool is selected ([DeactivateProcType](#))
- Handle mouse clicks ([DoClickProcType](#)).
- Handle key presses ([DoKeyDownProcType](#)).
- Control the shape of the cursor ([AdjustCursorProcType](#)).
- Return the tool's name ([GetTypeProcType](#)).
- Indicate whether the tool stays active after it is used once ([IsPersistentProcType](#)).
- Determine whether the tool is enabled. For example, if a tool is meant to be used within documents, but there are no documents open, it probably makes no sense to activate the tool ([AVComputeEnabledProc](#)).

See the description of [AVTool](#) in the [Acrobat Core API Reference](#) for a complete list of the callbacks.

Window Handlers

When a plug-in creates a window, it can register the window, so that it behaves like other windows in the Acrobat viewer, for example, when the viewer is minimized or hidden. For each window that a plug-in provides, a *window handler* must be provided.

NOTE: Window handlers are used only in the Macintosh version of the Acrobat viewers. Windows and UNIX versions of the viewers instead use the platform's native window handling mechanisms.

To define a window handler, a plug-in must provide a set of callbacks, specify them in an [AVWindowHandler](#) structure, and pass the structure to [AVWindowNew](#) or [AVWindowNewFromPlatformThing](#). The window handler's callbacks are automatically called by the Acrobat viewer. Default behavior is used for any missing callbacks.

The callbacks include ones that:

- Handle mouse clicks in the window ([AVWindowMouseDownProc](#)).
- Handle keystrokes in the window ([AVWindowKeyDownProc](#)).
- Draw the window's contents ([AVWindowDrawProc](#)).

- Permit or prevent closing of the window ([AVWindowWillCloseProc](#)).
- Clean up after the window has been closed ([AVWindowDidCloseProc](#)).
- Do anything that must be done when the window is activated or deactivated ([AVWindowDidActivateProc](#), [AVWindowWillDeactivateProc](#)).
- Do anything that must be done when the window becomes responsible for handling keystrokes or loses responsibility for handling keystrokes ([AVWindowDidBecomeKeyProc](#), [AVWindowWillResignKeyProc](#)).
- Permit or constrain window size changes ([AVWindowWillBeResizedProc](#)).
- Determine whether the **Cut**, **Copy**, **Paste**, **Clear**, **SelectAll**, and **Undo** menu items are enabled ([AVWindowCanPerformEditOpProc](#)).
- Perform **Cut**, **Copy**, **Paste**, **Clear**, **SelectAll**, and **Undo** operations ([AVWindowPerformEditOpProc](#)).
- Control the shape of the cursor when it is within the window ([AVWindowAdjustCursorProc](#)).

For a complete list of callbacks in a window handler, see the description of [AVWindowHandler](#) in the *Acrobat Core API Reference*.

File Systems

Plug-ins can add new *file systems* to Acrobat, to access files on a device that cannot be accessed as a local hard disk, such as a socket or a modem line.

To add a new file system, a plug-in must provide a set of callbacks and specify them in the [ASFileSysRec](#) structure. This structure is passed as a parameter to calls that require a file system. Unlike some of the other handlers in this chapter, there is no explicit registration.

The callbacks include ones that:

- Open ([ASFileSysOpenProc](#)) or close ([ASFileSysCloseProc](#)) a file.
- Flush a file's buffered data to disk ([ASFileSysFlushProc](#)).
- Get or set the current position in a file ([ASFileSysSetPosProc](#), [ASFileSysGetPosProc](#)).
- Get or set a file's logical size ([ASFileSysGetEofProc](#) or [ASFileSysSetEofProc](#)).
- Read data from a file ([ASFileSysReadProc](#)).
- Write data to a file ([ASFileSysWriteProc](#)).
- Delete a file ([ASFileSysRemoveProc](#)).
- Rename a file ([ASFileSysRenameProc](#)).
- Get a file's name ([ASFileSysGetNameProc](#))

- Determine the amount of free space on a volume ([ASFileSysGetStorageFreeSpaceProc](#)).
- Get a file system's name ([ASFileSysGetFileSysNameProc](#)).
- Test whether two files are the same ([ASFileSysIsSameFileProc](#)).
- Get a pathname to a temporary file ([ASFileSysGetTempPathNameProc](#)).
- Copy a pathname (not the underlying file) ([ASFileSysCopyPathNameProc](#)).
- Convert between device-independent and device-dependent pathnames ([ASFileSysDiPathFromPathProc](#)).
- Dispose of a pathname (not the underlying file) ([ASFileSysDisposePathNameProc](#)).
- Flush data on a volume ([ASFileSysFlushVolumeProc](#)).
- Handle asynchronous I/O ([ASFileSysAsyncReadProc](#), [ASFileSysAsyncWriteProc](#)).
- Handle multiple read requests ([ASFileSysMReadRequestProc](#)).

For details on each of the callbacks in a file system, see the description of [ASFileSysRec](#) in the *Acrobat Core API Reference*.

Progress Monitors

Progress monitors provide feedback to a user on the progress of a time-consuming operation. Some potentially time-consuming methods in the core API require a progress monitor as a parameter. The Acrobat viewer has a default progress monitor, which generally is sufficient for plug-ins to use. The built-in progress monitor can be obtained using [AVAppGetDocProgressMonitor](#).

Plug-ins can use the default progress monitor or implement their own by providing a set of callbacks, specifying them in the [ASProgressMonitorRec](#) data structure, and passing a pointer to the structure to the methods that require a progress monitor. (There is no explicit registration method.)

NOTE: Prior to Acrobat 5.0, the [ProgressMonitorRec](#) structure was used.

Plug-ins can also use a progress monitor (either the built-in one or their own) to display progress when they carry out a time-consuming task. To do this, they simply call the progress monitor's callbacks directly.

NOTE: Plug-ins that perform time-consuming tasks should, in general, allow the user to cancel them (see [AVAppGetCancelProc](#)).

The progress monitor callbacks include ones that :

- Initialize the progress monitor and display it with a current value of zero ([PMBeginOperationProc](#)).

- Draw a full progress monitor, then remove the progress monitor from the display ([PMEndOperationProc](#)).
- Set the value that corresponds to a full progress monitor display ([PMSetDurationProc](#)).
- Set the current value of the progress monitor and update the display ([PMSetCurrValueProc](#)).
- Get the progress monitor's maximum value ([PMGetDurationProc](#)).
- Get the progress monitor's current value ([PMGetCurrValueProc](#)).

For details, see the description of **ASProgressMonitorRec** in the [Acrobat Core API Reference](#).

Transition Handlers

Transitions allow effects such as dissolves or wipe-downs when displaying a new page. New transition types can be added by defining and registering a transition handler.

To add a new transition, a plug-in must provide a set of callbacks, specify them in the **AVTransHandler** data structure, and register them using [AVAppRegisterTransHandler](#). The callbacks include ones that:

- Get the transition type ([AVTransHandlerGetTypeProc](#)).
- Perform the transition; that is, do whatever is necessary to change to the next page with this transition style) ([AVTransHandlerExecuteProc](#)).
- Fill in the transition dictionary in the PDF file ([AVTransHandlerInitTransDictProc](#), [AVTransHandlerCompleteTransDictProc](#)).
- Provide information for the user interface that sets the attributes of the transition ([AVTransHandlerGetUINameProc](#)).

For a complete list of the callbacks in a transition handler, see the description of **AVTransHandler** in the [Acrobat Core API Reference](#).

This chapter describes the document security features of the Acrobat core API. It discusses:

- *Encryption and decryption* of PDF files so that only authorized users can read them.
- *Security handlers*, which are the primary mechanism for controlling access to a file. They contain code that performs user authorization and sets permissions. Acrobat has a built-in security handler; plug-ins can alter Acrobat's security system by adding new security handlers.
- New security features in Acrobat 5.0.

Encryption and Decryption

Encryption is controlled by an *encryption dictionary* in the PDF file. The Acrobat core API uses RC4 (a proprietary algorithm provided by RSA Data Security, Inc.) to encrypt document data, and a standard (proprietary) method to encrypt, decrypt, and verify user passwords to determine whether or not a user is authorized to open a document. See Section 3.5, “Encryption,” in the [PDF Reference](#) for more information on the encryption used in PDF files. See also the section on “Encryption” in [PDF: Changes From Version 1.3 to 1.4](#) for updated information.

Each stream or string object in a PDF file is individually encrypted. This level of encryption improves performance because objects can be individually decrypted as needed rather than decrypting an entire file. All objects—except for the encryption dictionary (which contains the security handler's private data)—are encrypted using the RC4 algorithm. Adobe licenses from RSA Data Security, Inc. Plug-ins may not substitute another encryption scheme for RC4.

A plug-in that implements a security handler is responsible for encrypting the values it places into the encryption dictionary, and it may use any encryption scheme. If the security handler does not encrypt the values it places into the encryption dictionary, the values are in plain text.

The core API provides two Cos layer methods to encrypt and decrypt data using the RC4 algorithm from RSA Data Security, Inc: [CosEncryptData](#) and [CosDecryptData](#). Security handlers may use these methods to encrypt data they want to put into the PDF file's encryption dictionary and decrypt data when it is read from the dictionary. Security handlers may instead choose to ignore these methods and use their own encryption algorithms.

Security Handlers

The code that performs user authorization and sets permissions is known as a *security handler*. The core API has one built-in security handler. This security handler supports two passwords:

- A *user* password that allows a user to open and read a protected document with whatever permissions the owner chose
- An *owner* password that allows a document's owner to also change the permissions granted to users

Plug-ins can use the core API's built-in security handler, or they can write their own security handlers to perform user authorization in other ways (for example, by the presence of a specific hardware key or file, or by reading a magnetic card reader). A security handler provided by a plug-in can use the Acrobat viewer's built-in dialog boxes for entering passwords and for changing permissions.

Security handlers are responsible for:

- Setting permissions on a file.
- Authorizing access to a file.
- Setting up a file's encryption and decryption keys.
- Maintaining the encryption dictionary of the PDF file containing the document.

Security handlers are used when:

- A document is opened — The security handler must determine whether a user is authorized to open the file and set up the decryption key that is used to decrypt the PDF file. See [“Opening a File” on page 177](#).
- A document is saved — The security handler must set up the encryption key and write whatever extra security-related information it wants into the PDF file's encryption dictionary. See [“Saving a File” on page 179](#).
- A user tries to change a document's security settings — The security handler must determine whether or not the user is permitted to do the operation. See [“Setting a Document's Security” on page 180](#).

A document may have zero, one, or two security handlers associated with it. A document has zero security handlers if no security is used on the file. When security is applied to a file, or the user selects a different security handler for a secured file, the newly-chosen security handler is not put in place immediately. Instead this new security handler is simply associated with the document; it is a pending security handler until the document is saved.

The new security handler is not put in place immediately because it is responsible for decrypting the contents of the document's encryption dictionary, and that dictionary is re-encrypted in the correct format for the new security handler only when the document is saved. As a result, a document may have both a current and a new security handler associated with it.

NOTE: On Acrobat 5.0, the **Save** or **Save As...** menu item can be used to save the file.

On Acrobat versions prior to 5.0, the file must be saved with **Save As...** for reasons described in “[Saving a File](#)” on page 179.

A security handler has two names: one that is placed in each PDF file that is saved by the handler (for example, **ADBE_Crypt**), and another name that Acrobat can use in any user interface items in which the security handler appears (for example, **Acrobat Developer Technologies default encryption**). This is similar to the two-name scheme used for menu items: a language-independent name that the code can refer to regardless of the user interface language, and another name that appears in the user interface. See Chapter 2, “[Registering and Using Plug-in Prefixes](#),” in the *Acrobat Development Overview* for details on plug-in naming conventions.

Adding a Security Handler

Acrobat has a built-in security handler. Plug-ins can add other security handlers by:

- Writing a set of callback routines to perform security-related functions.
- Specifying the callbacks in a **PDCryptHandlerRec** structure. See the description of **PDCryptHandlerRec** in the *Acrobat Core API Reference* for details on each of the security handler callbacks.
- Registering the handler by passing the structure to **PDRegisterCryptHandlerEx**

NOTE: **PDRegisterCryptHandlerEx** was introduced with Acrobat 5.0, and is the same as **PDRegisterCryptHandler** but accepts an extra parameter for storing client data. There are a number of callbacks in the **PDCryptHandlerRec** structure which have similar names except for the addition of “**Ex**”. In general, the **Ex** calls are newer and should be preferred but the older ones are still allowed for compatibility. This chapter generally refers to only one of the routines, for convenience.

Data Used By Security Handlers

The following sections refer to three types of data used by security handlers:

- *Authorization data* is the data the security handler needs to determine the user’s authorization level for a particular file (for example, not authorized to open the file, authorized to access the file with user permissions, authorized to access the file with owner permissions). Passwords are a common type of authorization data.
- *Security data* is whatever internal data the security handler wants to keep around. It includes security info, and perhaps internal state variables, internal flags, seed values, and so on.
- *Security info* is a subset of the security data. Specifically, it is a collection of flags that contains the information that Acrobat uses to display the current permissions to the user. This information includes permissions and the user’s authorization level (user or owner).

Security Handler Callbacks

A security handler must provide callbacks that:

- Determine whether a user is authorized to open a particular file and what permissions the user has once the file is open ([PDCryptAuthorizeExProc](#)).
- Create and fill an authorization data structure, using whatever user interface is needed to obtain the data (displaying a dialog into which the user can type a password, for example) ([PDCryptGetAuthDataExProc](#)).
- Create, fill, and verify a security data structure ([PDCryptNewSecurityDataProc](#)).
- Extract the security information from the security data structure ([PDCryptGetSecurityInfoProc](#)) (optional)
- Allow the user to request different security settings, usually by displaying a dialog box. ([PDCryptDisplaySecurityDataProc](#))
- Set up the encryption key used to encrypt the file ([PDCryptNewCryptDataProc](#)).
- Fill ([PDCryptFillEncryptDictProc](#)) or read the PDF file's encryption dictionary.
- Display the current document's permissions (required with the Acrobat 5.0 callbacks [PDCryptAuthorizeExProc](#) and [PDCryptGetAuthDataExProc](#)).

New Security Features in Acrobat 5.0

With Acrobat 5.0 and higher, a finer granularity of permissions has been predefined for the objects supported by a PDF document. Plug-ins can call the [PDDocPermRequest](#) method to request whether a particular operation is authorized to be performed on a specified object in a document. For details, see “[Querying PDDoc Permissions](#)” on page 91. Earlier Acrobat versions supported a much more limited set of permissions (an OR of the [PDPerms](#) values listed in the [Acrobat Core API Reference](#)) that a plug-in could request using the [PDDocGetPermissions](#) method.

To support the [PDDocPermRequest](#) method, two new callback methods [PDCryptAuthorizeExProc](#) and [PDCryptGetAuthDataExProc](#) also were introduced with Acrobat 5.0. These callbacks replace the [PDCryptAuthorizeProc](#) and [PDCryptGetAuthDataProc](#) callbacks, respectively.

NOTE: Acrobat 5.0 continues to support security handlers written with the [PDCryptAuthorizeProc](#) and [PDCryptGetAuthDataProc](#) callback methods. If a security handler does not support the newer methods, Acrobat calls the older ones and interprets the results.

Acrobat 5.0 also includes optional security handling for batch operations (operations on one or more files). There are a number of new callbacks (indicated by [PDCryptBatch...](#)) that a security handler should provide to support batch

processing. These callbacks are part of a [PDCryptBatchHandler](#) structure. The [PDCryptHandlerRec](#) structure contains a new member [CryptBatchHandler](#), which points to this structure. To support batch processing, a security handler should provide a non-NULL value for [CryptBatchHandler](#), and implement the batch callbacks.

Prior to Acrobat 5.0, the maximum length of the encryption key that Acrobat accepted was 40 bits. Acrobat version 5.0 or higher accommodates an encryption key length of 128 bits. These length limitations are imposed to comply with export restrictions.

Opening a File

The core API has several methods for opening files. [PDDocOpen](#), or [PDDocOpenEx](#) (introduced with Acrobat 5.0 and containing an additional parameter) is always used to open PDF files, even when a plug-in calls AV layer methods such as

[AVDocOpenFromFileWithParams](#). As a result, the sequence of operations is largely the same regardless of whether the document is being opened from the PD layer or from the AV layer. The difference is that if you call [PDDocOpen](#) directly, you must pass your own authorization procedure ([PDAuthProc](#)), while AV layer methods always use Acrobat's built-in authorization procedure. (See ["Acrobat's Built-in Authorization Procedure" on page 178](#).)

The authorization procedure must implement the authorization strategy, such as giving the user three chances to enter a password. The [PDAuthProc](#) is not part of a security handler, but it must call the security handler's methods to authorize the user (for example, to get the password from the user and to check whether or not the password is valid).

The security-related steps to opening a file are:

1. Acrobat looks for an **Encrypt** key in the PDF document's trailer, to determine whether or not the document is encrypted. If there is no **Encrypt** key, Acrobat opens the document immediately.
2. If there is an **Encrypt** key, its value is an encryption dictionary. Acrobat gets the value of the **Filter** key in the dictionary to determine which security handler was used when the file was saved. It looks in the list of registered security handlers (which contains Acrobat's built-in handler and any handlers that plug-ins or applications have registered) for one whose name matches the name found in the PDF file.
3. If Acrobat finds no match, indicating that the necessary handler could not be found, it does not open the document.
If it finds a matching security handler, it calls that handler's [PDCryptNewSecurityDataProc](#) callback to extract and decrypt information from the PDF file's encryption dictionary.
4. Acrobat calls the security handler's authorize callback ([PDCryptAuthorizeExProc](#)) with **NULL** authorization data, and with the

requested permissions set to **PDPermReqOprOpen** or **pdPermOpen** (requesting that the user be allowed to open the file). This allows support for authorization schemes that do not need authorization data. For details, see “[Acrobat’s Built-in Authorization Procedure](#)” on page 178.

5. If authorization succeeds, the handler’s authorization callback must return the **PDPermReqStatus** (when the callback is **PDCryptAuthorizeExProc**) or **pdPermOpen** (when the callback is **PDCryptAuthorizeProc**) indicating that the user is permitted to open the file.
6. If authorization fails, the authorization procedure passed in the call to open the **PDDoc** is called.

NOTE: This authorization procedure is not the same as the security handler’s authorize callback, although it must, at some point, call the security handler’s callback. (All AV layer file opening methods use Acrobat’s built-in authorization procedure.)

7. If authorization still fails, the file is not opened.
8. If authorization succeeds, Acrobat calls the security handler’s **PDCryptNewCryptDataProc** callback to create the decryption key that is used to decrypt the file. The **PDCryptNewCryptDataProc** callback can construct the decryption key in any way it chooses, but generally performs some calculation based on the contents of the security data structure filled previously by the handler’s **PDCryptNewSecurityDataProc** callback.

Acrobat’s Built-in Authorization Procedure

Acrobat’s built-in authorization procedure works as follows:

1. It calls the security handler’s authorize callback (which is either **PDCryptAuthorizeExProc**, introduced with Acrobat 5.0, or the older **PDCryptAuthorizeProc**) to request that the user be allowed to open the file. It passes **NULL** authorization data, to handle the case where no authorization data is needed. It also passes:
 - **PDPermReqObjDoc** and **PDPermReqOprOpen** when calling **PDCryptAuthorizeExProc**.
 - **pdPermOpen** when calling **PDCryptAuthorizeProc**.
2. If the authorize callback returns **true**, the file is opened. Otherwise, the authorization procedure executes the following steps up to three times, to give the user three chances to enter a password, or whatever authorization the security handler uses.
 - It calls the security handler’s get authorization data callback (**PDCryptGetAuthDataExProc** or the older **PDCryptGetAuthDataProc**). This callback should obtain the authorization data using whatever user interface

(a dialog box to obtain a password, for instance) or other means necessary, and then create and fill the authorization data structure.

- It calls the security handler's authorize callback, passing the authorization data returned by the get authorization data callback. If authorization succeeds, the authorize callback returns the permissions granted to the user, and the authorization procedure returns.

NOTE: The security handler's authorize callback should use only the authorization data passed to it from the get authorization data callback. It should not, for example, display a dialog box itself to obtain a password from the user.

The authorize callback can access the encrypted PDF document, allowing it to encrypt the authorization data using a mechanism that depends on the document's contents. By doing this, someone who knows one document's password cannot easily find out which other documents use the same password. The authorize callback can return permissions that depend on the password as well as the permissions specified when encryption was set up. This allows, for example, more rights to be granted to someone who knows a document's owner password than to someone who knows the document's user password.

Saving a File

When saving a file, it is important to keep in mind that:

- When a user selects document encryption for the first time or has selected a different security handler for an already encrypted file, the newly-selected handler does not take effect until the document is saved.
- To be allowed to save a file, the user must have **PDPermReqOprModify** (available with Acrobat 5.0 and higher) or either **pdPermEdit** or **pdPermEditNotes** permission.
- On Acrobat 5.0 and above, **File->Save As** and **File->Save** both force a complete encrypted copy of the file to be written. In Acrobat versions prior to 5.0, users must use **Save As...** to save a file in an encrypted form for the first time, or when a different security handler was selected for an already encrypted file. **Save** did an incremental update, so only the last changes made to the file would be encrypted, and the remainder of the document would still be usable by anyone (or would not be able to be decrypted by the newly-selected security handler).

When a secured file is saved:

- If the file is being saved in an encrypted form for the first time or if a different security handler is selected, Acrobat calls the new security handler's **PDCryptNewSecurityDataProc** callback. This action creates a new copy of the new security handler's security data structure.
- If the file is being saved in an encrypted form for the first time or if a different security handler is selected, Acrobat calls the new security handler's **PDCryptUpdateSecurityDataProc** callback. This presents whatever user interface the security handler has for enabling the user to set permissions.

- Acrobat calls the new security handler's **PDCryptFillEncryptDictProc** callback to encrypt and write into the PDF file's encryption dictionary whatever data the security handler wants to save in the PDF file.
- Acrobat writes out the encrypted file.
- Acrobat sets the new security handler as the document's current security handler.

Setting a Document's Security

Acrobat calls the new security handler's **PDCryptUpdateSecurityDataProc** callback to present whatever user interface the security handler has for allowing the user to set security, passwords, and so forth.

When security is set, the security handler obtains the permissions and authorization data (such as passwords) to be used for the file. The settings do not take effect until the file is saved, as described in the previous item

NOTE: In Acrobat 5.0, users select **File-> Document Security...** to set security. On Acrobat versions prior to 5.0, the user set security using the **Security** button in the **Save As...** dialog.

Implementation Examples

This section describes the sequence of callbacks and how they would be used by a plug-in that uses public-private key technology.

Saving a File With a New Encryption Dictionary

To save a file with a new encryption dictionary, the following callbacks in the **PDCryptHandlerRec** are used:

1. **PDCryptNewSecurityDataProc** creates and initializes a security data structure. It is called with **encryptDict** (a Cos object) set either to **NULL** or to a valid encryption dictionary, in which case the fields of the encryption dictionary are read and placed into the security data structure.
2. **PDCryptUpdateSecurityDataProc** gets the current security data structure by calling the **PDDocGetNewSecurityData** method. It then makes a copy of the structure with which to work. This new copy is freed if an error or cancel condition is encountered. The user is requested to log in to their PKI infrastructure to access the user's keys and certificates.

If the security data structure was seeded with information from **encryptDict**, an internal authorize procedure is called. This procedure decrypts and examines the

data fields in the security data structure copy that are set to indicate the user's permissions and, possibly, information relating to the document symmetric key.

A user interface is provided to enable your plug-in to specify a list of recipients for the document. If all goes well, the `secDataP` argument to `PDCryptUpdateSecurityDataProc` is set to the copy of the security data structure, and the viewer frees the original security data structure.

3. `PDCryptFillEncryptDictProc` writes data from the security data structure into the encryption dictionary. When the viewer is done with the security data structure, it calls the `PDCryptFreeSecurityDataProc`.

Opening an Encrypted File

To open an encrypted file, the following callbacks in the `PDCryptHandlerRec` are used:

1. `PDCryptNewSecurityDataProc` is called as described in the previous section.
2. `PDCryptAuthorizeExProc` is called and returns `NULL` since the authorization permissions have not been determined. This callback should not present any user interface.
3. `PDCryptGetAuthDataExProc`. The plug-in does not use the authorization data structure, but instead only the security data structure. It calls an internal authorization procedure that determines the authorization level of the logged-in user. This authorization procedure is the same procedure as is called by `PDCryptUpdateSecurityDataProc` in the previous section.
4. `PDCryptAuthorizeEx` or `PDCryptAuthorize`. The authorization permissions have now been established (by the call to get the authorization data) and are returned. The viewer will then open the file.

Utility Methods

These user interface utility methods are provided for the Acrobat viewer:

<code>AVCryptGetPassword</code>	Displays the Acrobat viewer's standard dialog box that prompts a user to enter a password. Plug-ins can use this method to obtain a user's password when opening a file.
<code>AVCryptDoStdSecurity</code>	Displays a security dialog to the user, allowing the user to change the document's permissions.

Most Acrobat core API methods do not return error codes, but raise *exceptions* when errors occur. Exceptions are handled by *exception handlers*. The Acrobat viewers provide a default exception handler, but this handler is not able to back gracefully out of an unfinished operation. Therefore, plug-ins should add their own exception handlers to trap and handle various exceptions, typically performing some cleanup (such as releasing memory) when an exception occurs. Your exception handler can either absorb the exception or re-raise the exception to pass it along to the next handler on the exception handler stack.

Exception Handlers

Plug-ins can use the **DURING**, **HANDLER**, and **END_HANDLER** macros to define exception handlers. The code for which an exception handler is to be active appears between the **DURING** and **HANDLER** macros, while the exception handler code appears between the **HANDLER** and **END_HANDLER** macros. For example, the following code declares an error handler that is active only during the call to **AVDocOpenFromFile**:

```
DURING
    avd = AVDocOpenFromFile(asp, NULL, (char *)NULL);
HANDLER
    avd = NULL;
    errorCode = ERRORCODE;
    AVAlertNote("Error opening file");
END_HANDLER
```

If the method raises an exception, the handler code is executed; otherwise it is not executed. In the example shown, the handler sets the value of two variables and displays an error message to the user.

When an exception occurs, your handler can access the exception error code by using the **ERRORCODE** macro. The value returned by the **ERRORCODE** macro does not change until another exception is raised.

The exception error code contains the following information:

- Severity
- Exception system
- Error number

Your exception handler can use all of this information to decide how to respond to the exception. Your plug-in can extract information from an exception code with macros listed in the following table:

TABLE 12.1 *Exception handling macros*

ERROR CODE	Defined in	Description
ErrGetSeverity	AcroErr.h	Gets the severity of the error where severity is one of the following: ErrNoError - No error ErrWarning - Warning ErrSilent - Don't display a message ErrSuppressable - Display a message if the user has not suppressed errors ErrAlways - Always display a message, even if others are suppressed
ErrGetSystem	AcroErr.h	Gets the system that raised the exception, where system is one of the values listed in Table 12.2.
ErrGetCode ErrGetSignedCode	AcroErr.h	Gets the error number. Acrobat's built-in exceptions are defined in <code>AcroErr.h</code> . Use ErrGetSignedCode if the platform considers error codes to be signed.
ErrBuildCode	AcroErr.h	Builds an error code, given the severity, system, and error number
RERAISE	CorCalls.h	Re-raises the most recently raised exception and passes it to the next exception handler in the stack.

TABLE 12.2 *Exception system names*

Name	Description
ErrSysNone	General error and out of memory error
ErrSysCos	CosStore filters
ErrSysCosSyntax	Cos syntax errors
ErrSysPDDoc	PDDoc and family, Page tree, outlines
ErrSysPDPage	PDPage and family, thumbs, annots

TABLE 12.2 *Exception system names*

Name	Description
ErrSysPDMetadata	XAP Metadata
ErrSysPDModel	Global PD model
ErrSysAcroView	AcroView
ErrSysPage	Page parsing and RIPing
ErrSysPDFEdit	PDFEdit
ErrSysPDSEdit	PDSEdit
ErrSysFontSrv	Font server
ErrSysRaster	Rasterizer
ErrSysASFile	ASFile I/O
ErrSysXtn	Errors registered by plug-ins are automatically assigned to this error system
ErrSysXtnMgr	Extension Manager
ErrSysMDSystem	Platform-specific system errors
ErrSysMDApp	Platform-specific application errors

The following code example illustrates an exception handler that simply determines which system raised an exception and displays that information in a dialog box:

```
switch(ErrGetSystem(ERRORCODE))
{
    case ErrSysNone: strcpy(msg, "No memory");break;
    case ErrSysCos: strcpy(msg, "CosStore");break;
    case ErrSysCosSyntax: strcpy(msg, "Cos syntax");break;
    case ErrSysPDDoc: strcpy(msg, "PDDoc");break;
    ...
    default: strcpy(msg, "Unknown system");break;
}
AVAlertNote(msg);
```

Raising Exceptions

In addition to handling exceptions Acrobat raises, plug-ins can use **ASRaise** to raise exceptions. Plug-ins can raise any of the exceptions that Acrobat has defined, or they can raise their own exceptions.

NOTE: Your plug-in should use the **ASRegisterErrorString** method to define its own exceptions.

Use the **RERAISE** macro (see [Table 12.1](#)) when you don't want your exception handler to handle an exception, but want to pass the exception to the next exception handler on the stack.

NOTE: If code that calls **ASraise** gets control as a result of a non-Acrobat event (such as a drag and drop event on some platforms), **ASraise** fails. There is no Acrobat code in the stack to handle the exception.

Handling an Exception Later

You may have situations where there is some clean-up code that needs to be executed regardless of whether an error was raised along the way. Here's a way to handle this:

```
ASInt32 err = 0;
...
DURING
...
HANDLER
    err = ERRORCODE;
END_HANDLER
/* free, clean up, etc. */
...
if (err) ASRaise(err);
```

Returning From an Exception Handler

To return from a method within a **DURING...HANDLER** block, don't use a **return** statement. Instead, use the following macros (defined in `CorCalls.h`):

- **E_RETURN(x)** returns the value **x**
- **E_RTRN_VOID** does not return a value

These macros remove stack entries added to the stack by the **DURING** macro. (They must not be used outside a **DURING/HANDLER** block.) Using **return** instead would cause the stack to be in an inconsistent state.

The following code example illustrates the use of the **E_RTRN_VOID** macro (the error handler in this example simply displays an alert dialog):

```

DURING
    pdDoc = AVDocGetPDDoc(avDoc);
    rootBm = PDDocGetBookmarkRoot(pdDoc);
    if(PDBookmarkIsValid(rootBm)){
        parentBm = PDBookmarkGetByTitle(rootBm, "Contents", 8, 1);
        if(PDBookmarkIsValid(parentBm)){
            pdAction = PDBookmarkGetAction(parentBm);
            if (!PDActionIsValid(pdAction))
                E_RTRN_VOID
            dest = PDActionGetDest(pdAction);
            if (!PDViewDestIsValid(dest))
                E_RTRN_VOID
            PDViewDestGetAttr(dest, &fit, &initRect, &zoom);
            pageNum = PDViewDestGetPageNumber(dest, pdDoc) + 2;
        } else {
            AVAlertNote("No Contents Bookmark");
            E_RTRN_VOID
        }
    } else {
        AVAlertNote("No Root Bookmark");
        E_RTRN_VOID
    }
HANDLER
    AVAlertNote("Exception raised");
    return;
END_HANDLER

```

The **E_RETURN(x)** macro *must not* call a function that might raise an exception. For example:

```
E_RETURN(foo())
```

is dangerous, if there's any possibility that `foo` could raise an exception. The reason is that **E_RETURN** pops an exception frame off the stack before evaluating the expression to be returned. If this evaluation raises an exception, it does not call your handler. Instead it calls the next handler up the stack.

Therefore, if you need to call a function, it is best to do it this way:

```
result = foo();
E_RETURN(result);
```

This way, if `foo` raises an exception, your handler will be executed.

API Methods That Raise Exceptions

The [Acrobat Core API Reference](#) specifies some of the exceptions that may be raised by each method. However, it should not necessarily be considered a comprehensive list.

There are several general rules in determining which exceptions may be raised by methods:

- Methods that create new objects or otherwise allocate memory can generally raise out-of-memory exceptions.
- Cos methods can generally raise exceptions if storage is exhausted or file access fails.
- **ASFile** methods generally do not raise exceptions, unless otherwise specified.
- When in doubt, assume that a method can raise exceptions and surround it with a **DURING/HANDLER/END_HANDLER** construct to handle any exceptions that may be raised.

Exception Handler Caveats

Don't Use **goto** In a **DURING...HANDLER** Block

Jumping outside a **DURING ... HANDLER** block disrupts the stack frame, as in this bad example:

```
DURING
  ...
  goto error;
HANDLER
  ...
END_HANDLER
  ...
error:
```

This is a bug: the top stack frame has not been popped, so the frame is incorrect. Instead, the following makes sure the stack frame is set up correctly:

```
DURING
  ...
  ASRaise(myspecialerrorcode);
  ...
HANDLER
  if ERRORCODE == myspecialerrorcode
    goto error;
END_HANDLER
  ...
error:
```

Don't Nest Exception Handlers In a Single Function

In general, don't nest exception handlers within a single function. The exception handling macros change the call stack, and nesting them can disrupt the stack.

Your plug-in can safely nest an exception handler if the nested handler is in another function called inside the **DURING ... HANDLER** block, as in the following example:

```
DURING
...
MyFunction( );
...
HANDLER
...
END_HANDLER
...
void MyFunction(void) {
    ...
    DURING
    ...
    HANDLER
    ...
    END_HANDLER
    ...
}
```

If you insist on nesting exception handlers in a single function, don't return from the inner exception handler (either through a call to `return` in a handler or **E_RETURN** from body code). This would leave the exception stack out of sync with the call stack. Any errors raised in body code surrounded by the outer exception handler will restore the incorrect calling environment and lead to unpredictable results. For example:

```
{
DURING /* Places one frame on the exception stack */
pdoc = AVDocGetPDDoc(avdoc);
DURING /* Places a second frame on the stack */
rootBm = PDDocGetBookmarkRot(pdDoc);
if ( !PDBookmarkIsValid(rootBm) ){
    E_RTRN_VOID
    /*
        Returning here messes up the exception stack
        because two frames have been placed on the stack
        and E_RTRN_VOID only clears one of them before
        returning
    */
}
pdAction = PDBookMarkGetAction(parentBm);
HANDLER
    AVAlertNote( "Bad AVDoc" );
    return (1);
/*
    Returning here messes up the exception stack

```

```

        because there is still a frame on the stack from
        the outer DURING macro and it will not be cleared
        before the function returns
    */
END_HANDLER
HANDLER
    AVAAlertNote( "Bad PDDoc" );
END_HANDLER
}

```

Be Careful About Register Usage

The **DURING** and **HANDLER** macros use the standard C **setjmp/longjmp** mechanism. The **DURING** macro calls **setjmp**. An exception results in a **longjmp** to the context that was saved by the most recent **setjmp**. When a **longjmp** occurs, all registers, including those containing variables the compiler optimized into register variables, are restored to the values they held when the **setjmp** occurred.

As a result, the state of local variables that have been optimized into registers is unpredictable when the exception handler is invoked. To avoid this situation, declare all variables that are set in the main body of the code and used in the exception handler or beyond (if the handler lets execution continue) as **volatile**. This ensures that they are never optimized into register variables, but are always referenced from memory.

NOTE: Memory access is generally substantially slower than register access, so performance may be compromised if a variable is referenced frequently. Therefore, plug-ins should only declare as **volatile** variables whose value is needed in the exception handler or beyond.

When using **volatile**, be sure to place the keyword in the correct location, for example:

```
volatile myStruct* p = 0;
```

declares the instance of the structure to be volatile, while

```
myStruct* volatile p = 0;
```

declares the pointer itself to be volatile. In general, the second form is the one to use.

This document has been updated since its previous version to reflect new features and APIs in Acrobat 5.0, and improved in other ways, as described in the following sections.

New Features in Acrobat 5.0

The following new features are described in the sections noted:

- New security features, including new security handler features and new document permissions. See [Chapter 11, “Document Security”](#) and [“Querying PDDoc Permissions” on page 91](#).
- Batch processing and **AVCommand** handlers. See [“AVCommand Handlers” on page 163](#).
- File conversion to and from PDF. See [“File Format Conversion Handlers” on page 166](#).
- Transparency, a new feature introduced in PDF 1.4. See [“PDEExtGState” on page 133](#) and [“PDEXGroup” on page 140](#).
- Metadata features. See [“Metadata” on page 86](#).

New Core API Objects

The following new objects have been added, to implement some of the features mentioned above as well as other functions:

- [“ASCab” on page 57](#)
- [“ASText” on page 62](#)
- [“AVCommand” on page 72](#)
- [“AVConversion” on page 75](#)
- [“AVSweetPea” on page 79](#)
- [“PDEPS” on page 138](#)
- [“PDESofMask” on page 139](#)
- [“PDEXGroup” on page 140](#)
- [“PDSysEncoding” on page 141](#)

Other Changes in This Document

- Revised Acrobat Software Development Kit Documentation Roadmap
- Added links to method descriptions in the *Acrobat Core API Reference*.
- Updated references to the *PDF Reference*. They all refer to the correct sections in the *PDF Reference, second edition, version 1.3*.
- Reorganized the presentation of much of the material, and added a Preface.
- Removed most references to the PDF Library. The PDF Library is a separate SDK from the Acrobat SDK.
- New information on “[Enumerating Page Objects](#)” on page 116
- Added Appendix B, “Portable Document Format.”

A

Object Interrelationships

The following figures show how various object types can be obtained from other object types. Use them to help you find your way among the objects in the Acrobat core API.

FIGURE A.1 File I/O Object Interrelationships

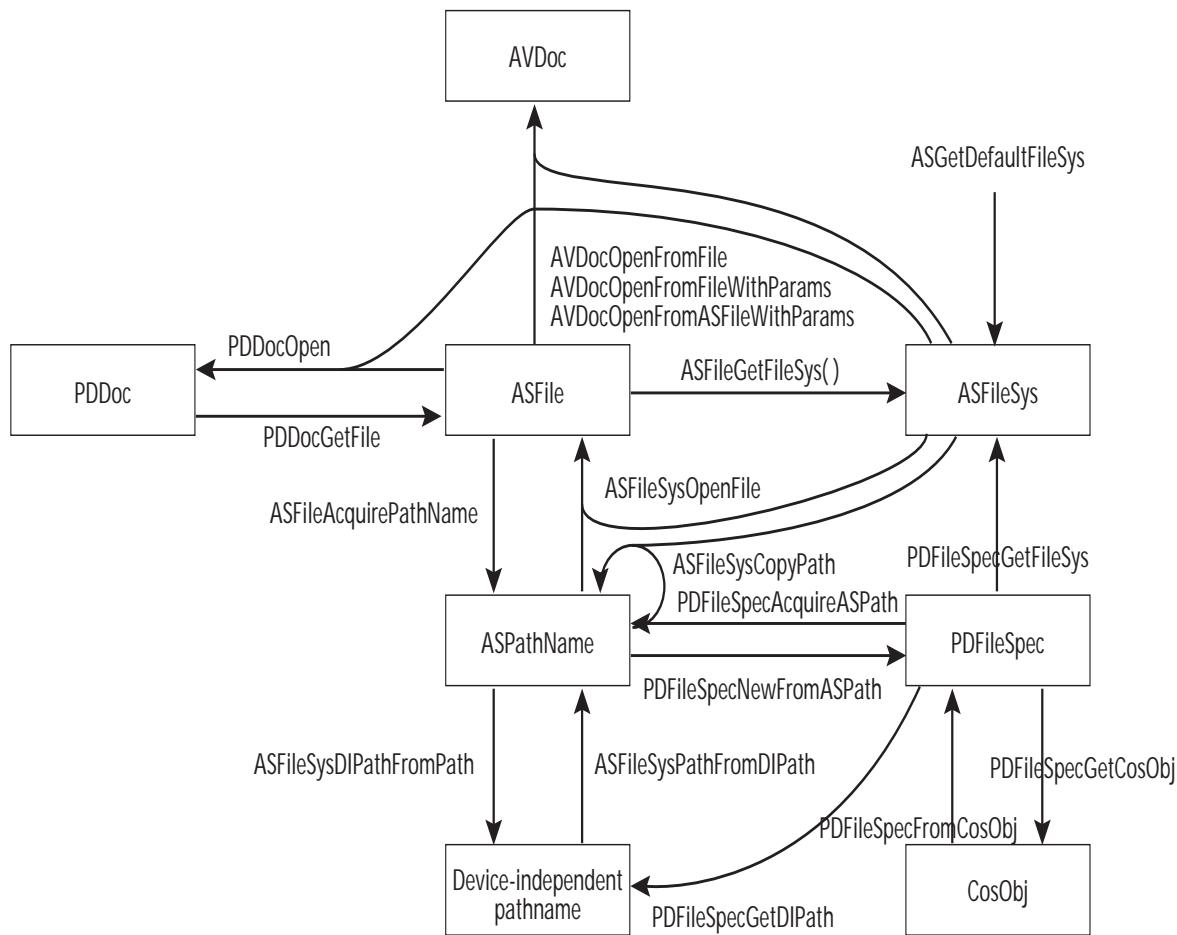
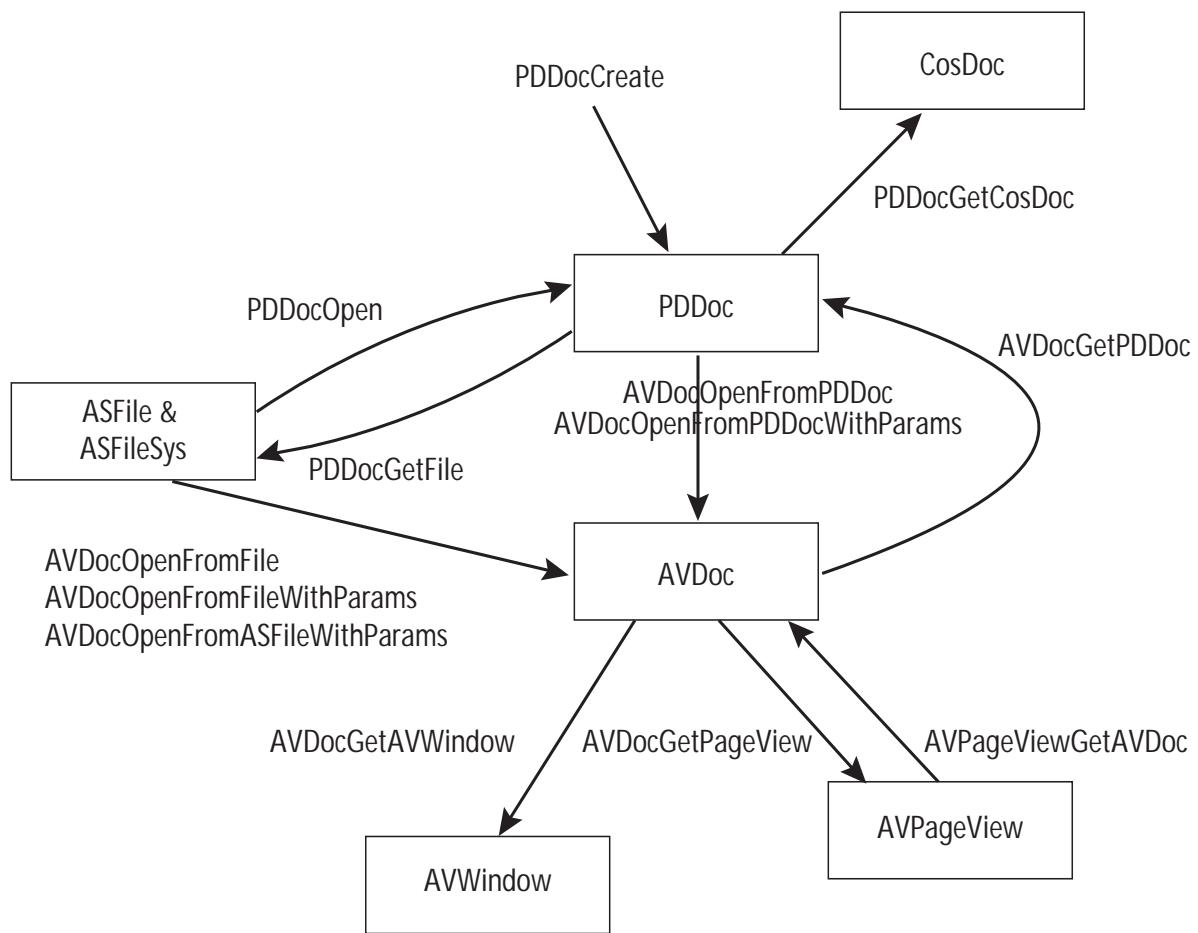


FIGURE A.2 Document Object Interrelationships

This Appendix provides a brief overview of PDF and the PDF structures. For details, see the [PDF Reference](#).

Relationship of Acrobat and PDF Versions

The following table shows how Acrobat and PDF versions are linked.

TABLE B.1 *PDF to Acrobat Version Compatibility*

PDF Version	Acrobat Version
1.0	2.0
1.1	2.1
1.2	3.0
1.3	4.0
1.4	5.0

Introduction To PDF

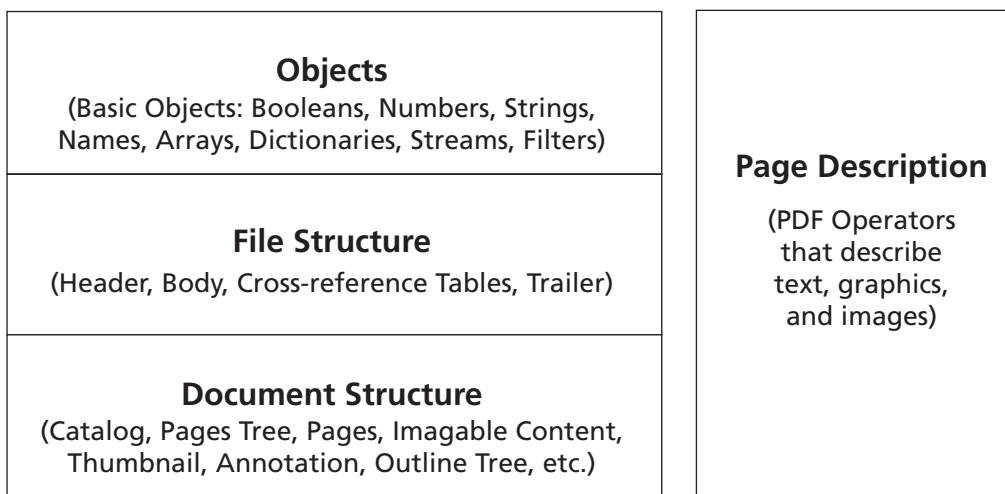
PDF is a means of representing text and graphics using the imaging model of the PostScript language. It describes the imaging required to draw a page or a collection of pages. A PDF file draws a page by placing “paint” on selected areas. Starting with a blank page, the page is drawn by using various marking operators to place marks on the page. Each new mark overlays any previous marks. Marks are painted figures defined by letter shapes (text) regions defined by combinations of lines and curves (line art), or sampled images (photographs or images). Unlike PostScript, a full language that is programmable, PDF does not contain procedures, variables, and control constructs. PDF uses a pre-defined set of high-level marking operators that can describe pages.

PDF handles images through image compression filters such as JPEG for color and grayscale images; CCITT Group 3 and Group 4, LZW, and Run Length compression for monochrome images; and LZW compression for text and graphics.

Fonts for text are described by a font descriptor. The font descriptor includes the font name, character metrics, and style information. This allows the accurate display of any fonts used in the document that may be missing on the reader’s system.

The following table shows the objects and structures of a PDF file.

FIGURE B.1 PDF File Structure

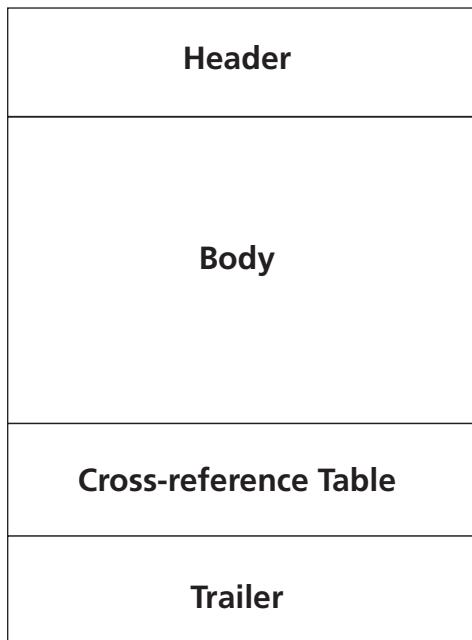


PDF Objects

The object types supported in PDF are similar to those supported by the PostScript language. There are seven basic types: booleans, numbers, strings, names, arrays, dictionaries, and streams, as well as a null object. Objects can be labeled and referred to by an ID (indirect objects).

File Structure

The PDF file structure consists of four sections: header, body, cross-reference table and a trailer. No line in a PDF file (except for those that are part of stream data) can be longer than 255 characters, and a line is delimited by a carriage return and linefeed, or a carriage return. The following table illustrates the structure of a PDF file.

FIGURE B.2 File Structure of a PDF File (not updated)

The one-line header specifies the version number of the PDF specification used in the file.

The body is a sequence of indirect objects (labeled objects) that describe the document. The objects are the basic PDF Object types (numbers, strings, dictionaries, etc.). The % symbol indicates a comment in the PDF file.

The cross-reference table contains information that enables random access to indirect objects in the file. For each indirect object, there is a one-line entry in the table that gives the location of the object in the file. To facilitate access to pages in a multi-page document, the cross-reference table can be used to locate and directly access pages and other objects in the document file.

The trailer includes the number of entries in the cross-reference table, a pointer to any other cross-reference sections, a catalog object for the document, and an info dictionary (optional) for the document.

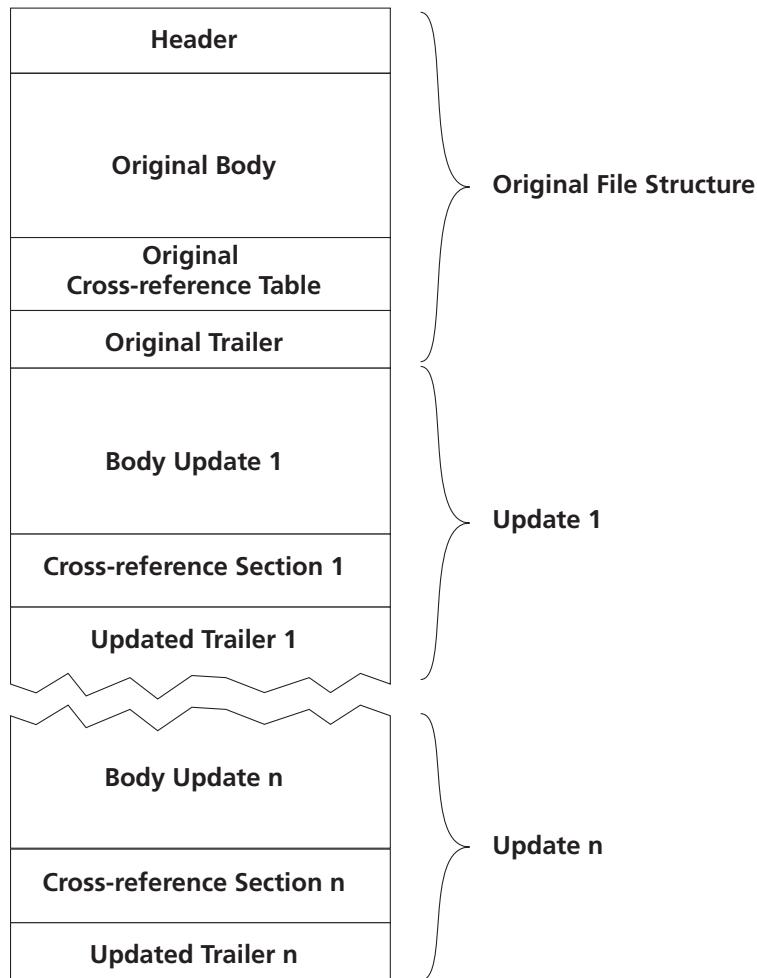
The PDF file is read from back to front and the trailer information permits the quick location of the cross-reference table, which in turn enables quick location of any object in the document.

A PDF file can be updated without rewriting the entire contents of the file. This is done by appending changes to the end of the file, while leaving the original contents intact.

NOTE: This may mean that a file with “deleted” elements will be larger than the original file. When the PDF file is updated, any new or changed objects are

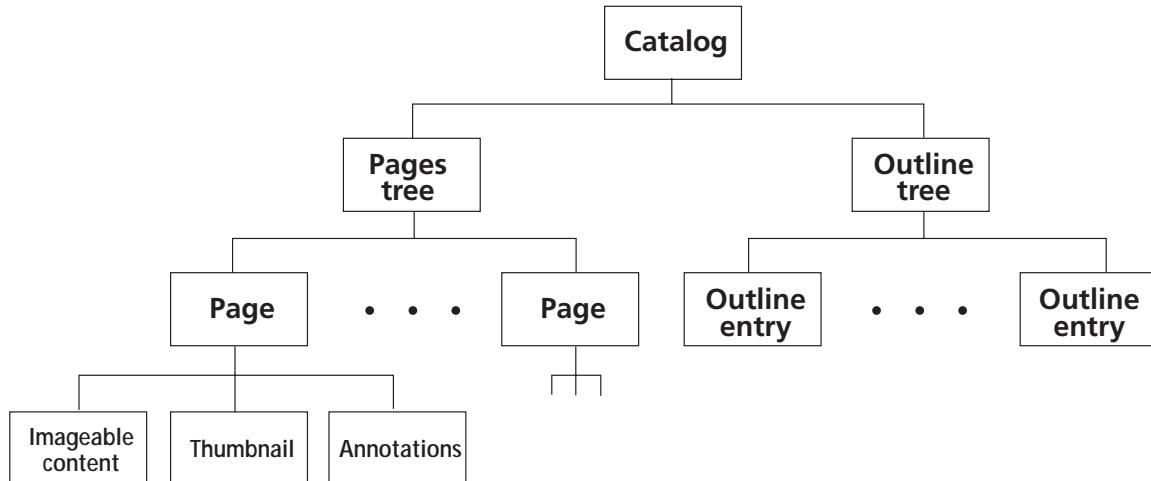
appended, an additional cross-reference table is added, and a new trailer is inserted. An appended file structure is shown in the following figure.

FIGURE B.3 File Structure of PDF File (after updating)



Document Structure

A PDF file contains pages with text, graphics, and images, along with other information such as thumbnails, text annotations, hypertext links, and bookmarks. It is organized into a catalog of a page tree and bookmark (or outline) tree, along with the pages, page contents and bookmark entries, as shown in the following figure.

FIGURE B.4 Document Structure

Page Contents

A PDF page contents is a sequence of graphic operators that generate marks that are applied to the current page, overlaying any previously made marks. The following table describes the four graphics objects.

TABLE B.2 Graphics objects

Object	Description
Path	An arbitrary shape made of straight lines, rectangles, and cubic curves.
Text	One or more character strings that can be placed anywhere on the page and in any orientation.
Image	A set of samples using a specified color model.
XObject	A PDF object referenced by name. The three types of XObjects are: <ul style="list-style-type: none"> • Image • Form • PostScript language form

Index

A

"about" box and splash screen 48
Acrobat Support (AS) layer 57
 ASAtom 57
 ASCab 57
 ASCallback 59
 ASExtension 59
 ASFile 60
 ASFileSys 60
 ASPathname 62
 ASStm 62
 ASText 62
 configuration 64
 errors 64
 fixed-point math 65
 HFT methods 66
 memory allocation 67
 platform-specific utilities 67
Acrobat Viewer (AV) layer 69
 AVActionHandler 70
 AVAlert 71
 AVAnnotHandler 71
 AVApp 71
 AVCommand 72
 AVConversion 75
 AVCrypt 75
 AVDoc 76
 AVGrafSelect 76
 AVMenu 76
 AVMenubar 77
 AVMenuItem 78
 AVPageView 79
 AVSweetPea 79
 AVSys 80
 AVTool 80
 AVToolBar 80
 AVToolButton 81
 AVWindow 82
Acrobat viewer's user interface 47

Acrobat viewers, controlling 51
action handlers 162
adding new object types 33
adjusting the cursor 45
Adobe Dialog Manager (ADM) 79
annotation handlers 162
annotation types, new 54
Apple events 46
applications, examples 51
AS layer 23
AV layer 22
AVCommand 163

B

batch processing 164

C

cabinets 57
callbacks 42
classes, PDFFedit 111
command handlers 163
complex types 27
controlling the Acrobat viewers 51
conversion handlers 166
converting file formats to and from PDF 75
coordinate systems 28, 48
 device space 29, 30
 machine port space 31
 user space 28, 30
core API
 handshaking and initialization 39
 mechanics 35
 objects 23
 organization 22
 types 26
core API organization
 AS layer 23
 AV layer 22

Cos layer 23
PD layer 22
PDFEdit 22
PDSEdit 23
platform-specific methods 23
Cos layer 23, 155
 CosArray 157
 CosBoolean 158
 CosDict 158
 CosDoc 157
 CosFixed 158
 CosInteger 159
 CosName 159
 CosNull 159
 CosObj 157
 CosStream 159
 CosString 160
 encryption/decryption 160
 file structure 156
Cos objects 28
creating HFTs 37
cross-platform dialog management 79

D

data types
 complex 27
 Cos 28
 opaque 28
 scalar 26
 simple 27
DDE messages 46
DEBUG 42
decryption 173
device space 29, 30
device space, coordinates of 29
dictionary access 54
document security 54, 173
 utility methods 181
drawing 48
drawing into another window 51

E

encoded text, specifying 62
encryption 173
encryption/decryption
 Cos layer 160
enumeration 45
error handling 183, 191
errors, Acrobat Support (AS) layer methods 64
event handling 45
examples of applications and plug-ins 51
exception handlers 183
exception handling 32
exporting HFTs 40
extracting text 53

F

file specification handlers 166
file systems 60
fixed-point math 65

H

handlers 161
 action 162
 annotation handlers 162
 command 163
 conversion 166
 file specification 166
 file systems 169
 progress monitors 170
 security 167, 174
 selection servers 167
 tools 168
 transition 171
 window 168
handling errors 183, 191
 adding new exceptions 186
 passing an exception 186
handling events 45
 adjust cursor 45
 key presses 46
 mouse clicks 45

handling exceptions 32
 handshaking 39
 handshaking with plug-ins 39
 help files 48
 HFT 35, 66
 creating 37
 exporting 40
 importing 40
 replacing entries in 37
 using 36
 HFT servers 36
 Host encoding 63
 host function table (HFT) 35, 66

I

importing HFTs 40
 indexed searching 52
 info dictionary access 54
 initialization of plug-ins 39
 initialization, plug-in 39
 initializing plug-ins 41
 integrating with an Acrobat viewer 21
 Interapplication Communication (IAC) 46
 interapplication communication (IAC) 21
 invoking AVCommands programmatically 72

K

key presses 46

L

language codes, setting 64
 layers
 Acrobat Support (AS) 57
 Acrobat Viewer (AV) 69
 Cos 155
 Portable Document (PD) 85
 logical structure 143

M

machine port space 28, 31

Macintosh utilities 67
 mechanics of the core API 35
 memory allocation 67
 menus and menu items 47
 message handling, adding 46
 Metadata 86
 method names, general format of 24
 methods, replacing 37
 mouse click processing 48
 mouse clicks 45

N

new annotation types 54
 notifications 44

O

object attributes 23
 object names, conventions for 23
 object types, adding new 33
 opaque data types 23
 opaque objects
 objects, opaque 23
 opaque types 28
 organization of core API 22

P

page view layers 48
 passing an exception 186
 path name 62
 PD layer 22
 PDF files
 private data in 33, 55
 PDF format, converting to and from 75
 PDFEdit 22, 109
 classes 111
 comparison with other core APIs 115
 debugging tools and techniques 126
 dump methods 128
 general methods 129
 hit testing 117
 matrix operations 119

- page creation 121
- PDEClip 129
- PDEColorSpace 129
- PDEContainer 130
- PDEContent 131
- PDEDeviceNColors 132
- PDEElement 132
- PDEExtGState 133
- PDEFont 134
- PDEForm 135
- PDEGroup 136
- PDEImage 136
- PDEObject 137
- PDEPath 137
- PDEPattern 137
- PDEPlace 138
- PDEShading 139
- PDEText 139
- PDEUnknown 140
- PDEXObject 141
- PDSysFont 141
- PDFFileSpec 93
- PDMModel layer 85
- PDSEdit 23, 143
 - PDSAttrObj 146
 - PDSClassMap 147
 - PDSElement 146
 - PDSMC 147
 - PDSObjr 147, 154
 - PDSRoleMap 147
 - PDSTreeRoot 146
- permissions, querying PDDoc 91
- platform-specific utilities 67
- plug-in initialization 39
- plug-in naming conventions 46
- plug-in prefixes 46
- plug-ins 21
 - examples 51
 - initialization 41
 - reducing conflicts among 49
 - unloading 42
- plug-ins, Reader-enabled 47
- Portable Document (PD) layer 85
 - general methods 86
- PDAAction 88
- PDAAnnot 88
- PDBead 89
- PDBBookmark 89
- PDCharProc 90
- PDDoc 91
- PDFFileSpec 93
- PDFForm 96
- PDGraphic 97
- PDIImage 97
- PDIInlineImage 98
- PDLINKAnnot 98
- PDNameTree 99
- PDNumTree 99
- PDPAGE 99
- PDPAGELabel 100
- PDPATH 101
- PDSTYLE 101
- PDTXT 101
- PDTXTAnnot 102
- PDTXTSelect 102
- PDTThread 104
- PDTThumb 104
- PDTTrans 104
- PDViewDestination 104
- PDWord 105
- PDWordFinder 106
- PDXObject 107
- private data in PDF files 33, 55
- progress monitors 170

Q

- quadrilaterals 32
- querying PDDoc permissions 91

R

- Reader-enabled plug-ins 47
- rectangles 32
- replacing entries in HFTs 37
- replacing methods 37

S

scalar types 26
searching, indexed 52
security 173
 document 54
 new features in Acrobat 5.0 176
security handlers 167, 174
 callbacks 176
selection servers 167
simple types 27
splash screen 48
streams 62
structure, logical 143

T

tagged PDF 143
text, extraction 53
toolbar 48
transition handlers 171
translating between user and device space 30
transparency 133
types, core API 26

U

Unicode 63
UNIX utilities 67
unloading plug-ins 42
user interface 47
 "about" box and splash screen 48
 help files 48
 menus and menu items 47
 toolbar 48
user space 28, 30
user space, coordinates of 28

W

window, drawing into another 51
Windows utilities 68

