Technical Note #5190
Version : Acrobat 6.0
# Contents

**Preface** ................................................................. 15

- Introduction ......................................................... 15
- Audience ............................................................... 15
- Assumptions .......................................................... 16
- How This Document Is Organized ................................ 16
- Related Documentation .............................................. 17
- Conventions Used in This Book .................................. 18

**Chapter 1  Core API Overview** .................................... 21

- Integrating With the Acrobat Viewers ............................ 21
- The Acrobat Core API ................................................ 22
- Core API Objects ...................................................... 23
- Core API Methods ..................................................... 24
- Data Types .............................................................. 26
  - Scalar Types ....................................................... 26
  - Simple Types ...................................................... 27
  - Complex Types ..................................................... 28
  - 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
Chapter 3 Plug-in Applications

Controlling the Acrobat Viewers
Drawing Into Another Window
Indexed Searching
Steps in the Acrobat Product’s Indexed Searching
Extracting Text
Providing Document Security
Modifying File Access
Creating New Annotation Types
Accessing the Info Dictionary

Acrobat Core API Overview
Chapter 4  Acrobat Support  .............................................. 55
  ASAtom ................................................................. 55
  ASCab ................................................................. 55
    ASCab Method Naming ........................................ 56
    Handling Pointers ............................................... 56
    ASCab Methods .................................................. 56
  ASCallback ......................................................... 57
  ASDate ............................................................... 57
  ASExtension ........................................................ 57
  ASFFile .............................................................. 58
  ASFFileSys .......................................................... 58
  ASPlatformPath ..................................................... 60
  ASStm ................................................................. 60
  ASText ............................................................... 60
  Configuration ....................................................... 62
  Errors ................................................................. 62
  Fixed-point Math .................................................. 62
    Fixed-point Utility Macros .................................... 63
    Fixed-point Mathematics Methods ............................ 63
    Fixed-point Matrix Methods .................................. 63
  HFT Methods ........................................................ 64
  Memory Allocation ................................................ 64
  Platform-Specific Utilities ......................................... 64
    Macintosh ......................................................... 64
    UNIX ............................................................... 65
    Windows .......................................................... 65

Chapter 5  Acrobat Viewer Layer  ..................................... 67
  General ............................................................... 68
  AVActionHandler .................................................. 68
  AVAlert .............................................................. 68
  AVAnnotHandler .................................................... 69
  AVApp ............................................................... 69
  AVCommand ........................................................ 70
    Invoking AVCommands Programmatically ..................... 70
## Chapter 6  Portable Document Layer  83

General PD Layer Methods ................................................. 84
Metadata ............................................................................. 84
   New Metadata Features in PDF 1.4 .................................. 85
   Metadata APIs in Acrobat 5.0 ........................................... 85
PDAction ................................................................. 86
PDAannot ................................................................. 86
PDBead ................................................................. 87
PDBookmark ............................................................ 87
PDCharProc ........................................................... 88
PDDoc ................................................................. 88
   Querying PDDoc Permissions ..................................... 89
   PDDoc Methods ...................................................... 90
PDFFileSpec ........................................................ 91
PDFont ................................................................. 91
PDForm ................................................................. 94
PDGraphic ............................................................. 94
PDImage ............................................................... 95
<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDInlinelImage</td>
<td>95</td>
</tr>
<tr>
<td>PDLinkAnnot</td>
<td>96</td>
</tr>
<tr>
<td>PDNameTree</td>
<td>96</td>
</tr>
<tr>
<td>PDNumTree</td>
<td>97</td>
</tr>
<tr>
<td>PDNameTree</td>
<td>97</td>
</tr>
<tr>
<td>PDPage</td>
<td>98</td>
</tr>
<tr>
<td>PDPageLabel</td>
<td>99</td>
</tr>
<tr>
<td>PDPath</td>
<td>100</td>
</tr>
<tr>
<td>PDStyle</td>
<td>101</td>
</tr>
<tr>
<td>PDText</td>
<td>101</td>
</tr>
<tr>
<td>PDTextAnnot</td>
<td>102</td>
</tr>
<tr>
<td>PDTextSelect</td>
<td>102</td>
</tr>
<tr>
<td>PDTThread</td>
<td>103</td>
</tr>
<tr>
<td>PDThumb</td>
<td>104</td>
</tr>
<tr>
<td>PDTrans</td>
<td>104</td>
</tr>
<tr>
<td>PDViewDestination</td>
<td>105</td>
</tr>
<tr>
<td>PDWord</td>
<td>105</td>
</tr>
<tr>
<td>PDWordFinder</td>
<td>106</td>
</tr>
<tr>
<td>PDXObject</td>
<td>107</td>
</tr>
</tbody>
</table>

### 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
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure Tree Root</td>
<td>151</td>
</tr>
<tr>
<td>Structure Elements</td>
<td>152</td>
</tr>
<tr>
<td>Adding Marked Content to an Element</td>
<td>152</td>
</tr>
<tr>
<td>Adding an Object Reference to an Element</td>
<td>153</td>
</tr>
<tr>
<td>Class Map</td>
<td>154</td>
</tr>
<tr>
<td>Role Map</td>
<td>154</td>
</tr>
<tr>
<td><strong>Chapter 9  Cos Layer</strong></td>
<td><strong>155</strong></td>
</tr>
<tr>
<td>Cos Objects: Direct and Indirect.</td>
<td>155</td>
</tr>
<tr>
<td>File Structure</td>
<td>156</td>
</tr>
<tr>
<td>Cos Objects in the Core API</td>
<td>156</td>
</tr>
<tr>
<td>CosDoc</td>
<td>157</td>
</tr>
<tr>
<td>CosObj</td>
<td>157</td>
</tr>
<tr>
<td>CosObjCollection</td>
<td>157</td>
</tr>
<tr>
<td>CosArray</td>
<td>158</td>
</tr>
<tr>
<td>CosBoolean</td>
<td>158</td>
</tr>
<tr>
<td>CosDict</td>
<td>158</td>
</tr>
<tr>
<td>CosFixed</td>
<td>159</td>
</tr>
<tr>
<td>CosInteger</td>
<td>159</td>
</tr>
<tr>
<td>CosName</td>
<td>159</td>
</tr>
<tr>
<td>CosNull</td>
<td>160</td>
</tr>
<tr>
<td>CosStream</td>
<td>160</td>
</tr>
<tr>
<td>CosString</td>
<td>160</td>
</tr>
<tr>
<td>Encryption/Decryption</td>
<td>161</td>
</tr>
<tr>
<td><strong>Chapter 10  Handlers</strong></td>
<td><strong>163</strong></td>
</tr>
<tr>
<td>Action Handlers</td>
<td>164</td>
</tr>
<tr>
<td>Annotation Handlers</td>
<td>164</td>
</tr>
<tr>
<td>AVCommand Handlers</td>
<td>165</td>
</tr>
<tr>
<td>Creating an AVCommand Handler</td>
<td>165</td>
</tr>
<tr>
<td>Exposing AVCommands to the Batch Framework</td>
<td>166</td>
</tr>
<tr>
<td>File Format Conversion Handlers</td>
<td>167</td>
</tr>
<tr>
<td>File Specification Handlers</td>
<td>168</td>
</tr>
<tr>
<td>Security Handlers</td>
<td>168</td>
</tr>
<tr>
<td>Selection Servers</td>
<td>168</td>
</tr>
<tr>
<td>Tools</td>
<td>169</td>
</tr>
</tbody>
</table>
Chapter 11 Document Security ................................................. 175
   Encryption and Decryption ........................................... 175
   Security Handlers ..................................................... 176
      Adding a Security Handler ....................................... 177
      Security Handler Callbacks .................................... 178
      New Security Features in Acrobat 5.0 ......................... 178
   Opening a File ....................................................... 179
   Acrobat’s Built-in Authorization Procedure .................... 180
   Saving a File ........................................................ 181
   Setting a Document’s Security ..................................... 181
   Implementation Examples ............................................ 182
      Saving a File With a New Encryption Dictionary ........... 182
      Opening an Encrypted File ...................................... 182
   Utility Methods ....................................................... 183

Chapter 12 Handling Errors .................................................. 185
   Exception Handlers .................................................. 185
   Handling an Exception Later ....................................... 188
   Returning From an Exception Handler ......................... 188
   API Methods That Raise Exceptions ............................. 189
      Do Not Use goto In a DURING...HANDLER Block .......... 190
      Do Not Nest Exception Handlers In a Single Function ...... 190
      Be Careful About Register Usage .............................. 192

Chapter 13 Changes For This Revision .................................... 195
   Changes in This Document ......................................... 195
   Compatability with API Changes in Acrobat 6 .................. 195
      New API Categories .............................................. 195
      Data Type Changes .............................................. 197
   Support for Backward Compatibility ............................ 198
   Testing With the TEST_OLD_VERSION Macro .................... 198
<table>
<thead>
<tr>
<th>Appendix A</th>
<th>Object Interrelationships</th>
<th>201</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix B</td>
<td>Portable Document Format</td>
<td>203</td>
</tr>
<tr>
<td>Relationship of Acrobat and PDF Versions</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>Introduction To PDF</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>PDF Objects</td>
<td>204</td>
<td></td>
</tr>
<tr>
<td>File Structure</td>
<td>204</td>
<td></td>
</tr>
<tr>
<td>Document Structure</td>
<td>206</td>
<td></td>
</tr>
<tr>
<td>Page Contents</td>
<td>207</td>
<td></td>
</tr>
</tbody>
</table>

| Index | 209 |
Preface

NOTE: This document has not yet been updated with information about new features in Acrobat 6.0. That information will be incorporated in the R1 release of the Acrobat SDK (July 2003).

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 Adobe 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

Audience

The primary audience of this document is Acrobat and Adobe 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 with 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 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 6.0.

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 Adobe 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, fourth edition, version 1.5 describes PDF version 1.5 in detail, including PDF object types, file format, and document structure.
- ADM Programmer’s Guide and Reference 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 Book

The Acrobat documentation uses text styles according to the following conventions.

<table>
<thead>
<tr>
<th>Font</th>
<th>Used for</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>monospaced</td>
<td>Paths and filenames</td>
<td>C:\templates\mytmpl.fm</td>
</tr>
<tr>
<td></td>
<td>Code examples set off from plain text</td>
<td>These are variable declarations:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>AVMenu commandMenu, helpMenu;</code></td>
</tr>
<tr>
<td>monospaced bold</td>
<td>Code items within plain text</td>
<td>The <code>GetExtensionID</code> method ...</td>
</tr>
<tr>
<td></td>
<td>Parameter names and literal values in reference documents</td>
<td>The enumeration terminates if <code>proc</code> returns <code>false</code>.</td>
</tr>
<tr>
<td>monospaced italic</td>
<td>Pseudocode</td>
<td><code>ACCB1 void ACCB2 ExeProc(void) { do something }</code></td>
</tr>
<tr>
<td></td>
<td>Placeholders in code examples</td>
<td><code>AFSimple_Calculate(cFunction, cFields)</code></td>
</tr>
<tr>
<td>blue</td>
<td>Live links to Web pages</td>
<td>The Acrobat Solutions Network URL is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>http://partners/adobe.com/asn/</code></td>
</tr>
<tr>
<td></td>
<td>Live links to sections within this document</td>
<td>See Using the SDK.</td>
</tr>
<tr>
<td></td>
<td>Live links to other Acrobat SDK documents</td>
<td>See the <em>Acrobat Core API Overview</em>.</td>
</tr>
<tr>
<td></td>
<td>Live links to code items within this document</td>
<td>Test whether an <code>ASAtom</code> exists.</td>
</tr>
<tr>
<td>bold</td>
<td>PostScript language and PDF operators, keywords, dictionary key names</td>
<td>The <code>setpagedevice</code> operator</td>
</tr>
<tr>
<td></td>
<td>User interface names</td>
<td>The <code>File</code> menu</td>
</tr>
<tr>
<td>Font</td>
<td>Used for</td>
<td>Examples</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>italic</td>
<td>Document titles that are not live links</td>
<td>Acrobat Core API Overview</td>
</tr>
<tr>
<td></td>
<td>New terms</td>
<td>User space specifies coordinates for...</td>
</tr>
<tr>
<td></td>
<td>PostScript variables</td>
<td>filename deletefile</td>
</tr>
</tbody>
</table>
The Acrobat core API is a set of interfaces you can use to write plug-ins that integrate with Acrobat and Adobe Reader. This chapter introduces the core API, describing its object orientation and organization, and a number of other concepts fundamental to understanding the API.

Integrating With the Acrobat Viewers

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

- Create plug-ins that are dynamically linked to the Acrobat viewer and extend the viewer’s functionality.
- Write 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.

Through IAC interfaces, an application can control the viewer in the same 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.
The 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 Layer*

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 Layer*

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 the reverse. 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:

\[ \text{<layer><object><verb><thing>}, \]

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>Creates a new unattached object.</td>
<td>AVMenuNew</td>
</tr>
<tr>
<td>AddNew</td>
<td>Creates a new object using the specified parameters and adds the new object to the current object.</td>
<td>PDBookmarkAddNewChild</td>
</tr>
<tr>
<td>Add</td>
<td>Adds the second object as a child to the current object.</td>
<td>PDBookmarkAddChild</td>
</tr>
<tr>
<td>Create</td>
<td>Creates a new object of a given type.</td>
<td>PDDocCreatePage</td>
</tr>
<tr>
<td>Destroy</td>
<td>Destroys the specified object and releases its resources immediately.</td>
<td>PDBookmarkDestroy</td>
</tr>
<tr>
<td>Open</td>
<td>Opens an object from storage or a stream.</td>
<td>AVDocOpenFromfile</td>
</tr>
<tr>
<td>Close</td>
<td>Destroys an object that was opened. Closes the underlying storage or stream.</td>
<td>ASFileClose</td>
</tr>
<tr>
<td>Acquire</td>
<td>Obtains a shareable resource from a parent object. Or, increments a reference counter for an object. The shared object is not destroyed until all acquirers have released it.</td>
<td>AVMenuItemAcquire</td>
</tr>
<tr>
<td>Release</td>
<td>Releases a shared object.</td>
<td>PDPagesRelease</td>
</tr>
</tbody>
</table>
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:

```c
PDDoc doc;
PDPage page;

doc = PDDocOpenFromASFile("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.

---

**TABLE 1.1 Verbs in API method names**

| **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 does not 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`. |
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 PDAnnotIsValid. 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>
<thead>
<tr>
<th>Type</th>
<th>Size (in bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASBool</td>
<td>2</td>
<td>boolean</td>
</tr>
<tr>
<td>ASUns8</td>
<td>1</td>
<td>unsigned char</td>
</tr>
<tr>
<td>ASUns16</td>
<td>2</td>
<td>unsigned short</td>
</tr>
<tr>
<td>ASUns32</td>
<td>4</td>
<td>unsigned long</td>
</tr>
<tr>
<td>ASInt8</td>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>ASInt16</td>
<td>2</td>
<td>signed short</td>
</tr>
</tbody>
</table>
Core API Overview

Data Types

Named Numeric Types

Acrobat 6.0 introduced named numeric types that are used in place of generic numeric types in methods and data structures. This change is intended to support a future change from 16-bit to 32-bit values. For example:

- **ASArraySize**
- **ASByteCount**
- **AVDevCoord**
- **AVDevSize**
- **AVScreenCoord**

Currently, the change is entirely cosmetic; that is, in this release, methods and data can be compiled without error if they still use the generic types. However, future releases may conditionally compile the named types (as 16-bit or 32-bit values) depending on the release version. It is recommended that you update your code to use the new named numeric types.

For complete details of the named numeric types, see the *Acrobat Core API Reference*.

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 `PDFontGetMetrics` returns font metrics by filling out a complex structure (`PDFontMetrics`).
- 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).
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.
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:

```c
ASFixedRect userRect;
AVRect deviceRect;

 PDAnnotGetRect(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:

```c
ASFixedRect userRect;
AVRect deviceRect;

AVPageViewSetPageNum(pageView, annotPageNum);
 PDAnnotGetRect(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.
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 Adobe 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).
Using HFTs

Plug-ins must use the `ASEExtensionMgrGetHFT` 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 Adobe Reader methods, see Replaceable Methods in the Lists section 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_REPLACED_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.
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:

```c
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**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handshakeVersion</td>
<td>The version of the handshaking data structure used. Currently HANDSHAKE_V0200.</td>
</tr>
<tr>
<td>extensionName</td>
<td>The ASAtom corresponding to the plug-in’s name. You can covert the plug-in’s name to an ASAtom using ASAtomFromString. The name should be less than 25 characters.</td>
</tr>
<tr>
<td>PluginExportHFTs</td>
<td>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.</td>
</tr>
<tr>
<td>PluginImportReplaceAndRegister</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.
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:

```c
ACCB1 ASBool ACCB2 PluginExportHFTs(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 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 `PluginExportHFTs` 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 `PluginExportHFTs` 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). 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:

```c
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 \texttt{ACCB1} and \texttt{ACCB2} macros, for example:

\begin{verbatim}
  static ACCB1 const char* ACCB2 getThingName(Thing* foo);
\end{verbatim}

Declare any function pointer \texttt{typedef}s using the \texttt{ACCBPROTO1} and \texttt{ACCBPROTO2} macros, for example:

\begin{verbatim}
  typedef ACCBPROTO1 const char* (ACCBPROTO2* fooProc) (Thing* foo);
\end{verbatim}
You can use this example as a template to set up an AVMenuItem using callbacks:

```c
/* 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);
    AVMenuItemSetExecuteProc(menuItem,
                ASCallbackCreateProc(AVExecuteProc, &executeMyItem), NULL);
    AVMenuItemSetComputeEnabledProc(menuItem,
                ASCallbackCreateProc(AVComputeEnabledProc, &isMyItemEnabled),
                NULL);
    AVMenuItemSetComputeMarkedProc(menuItem,
                ASCallbackCreateProc(AVComputeMarkedProc, &isMyItemMarked), NULL);
}

static 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
specify 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 PDPaths)
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 `PDPathEnum`, 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 Adobe Reader Differences

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

**NOTE:** Adobe 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 Adobe 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 Adobe 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 Adobe 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 Adobe 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.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Page contents</td>
</tr>
<tr>
<td>LINK_LAYER (1)</td>
<td>Links</td>
</tr>
<tr>
<td>NOTE_LAYER (3)</td>
<td>Closed notes. Open notes are just above this.</td>
</tr>
</tbody>
</table>

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 164 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 74
- “AVMenubar” on page 75
- “AVMenultem” on page 75

For a discussion of tool buttons, see
- “AVTool” on page 78
- “AVToolBar” on page 78
- “AVToolButton” on page 79

### 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 “PDPage” on page 100.

---

**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 102, “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 58
- “File Specification Handlers” on page 168

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 “PDAannot” on page 86.

For information on adding a handler for a new annotation type, see “Annotation Handlers” on page 164.

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 88.

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.”
Acrobat Support

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. ASAtom methods include, for example:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASAtomFromString</td>
<td>Converts a string to an ASAtom</td>
</tr>
<tr>
<td>ASAtomGetString</td>
<td>Gets a string, given an ASAtom</td>
</tr>
<tr>
<td>ASAtomExistsForString</td>
<td>Tests whether an ASAtom exists for the specified string</td>
</tr>
</tbody>
</table>

---

**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 store 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 178).
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 (for example, 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, for example:

<table>
<thead>
<tr>
<th>ASCabMethod</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCabCopy</td>
<td>Copies all key-value pairs from one cabinet into another.</td>
</tr>
<tr>
<td>ASCabDestroy</td>
<td>Destroys a cabinet and all its key-value pairs.</td>
</tr>
<tr>
<td>ASCabDetachCab</td>
<td>Retrieves a cabinet stored as a key in another cabinet, and removes the key.</td>
</tr>
</tbody>
</table>
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.

**ASDate**

Date objects represent a particular date and time. All date objects are guaranteed to give accurate representation of UTC time (not adjusted for leap seconds, as the addition of leap seconds to the international calendar does not happen according to a well-defined rule).

The related `ASTimeSpan` object represents an exact time span, measured in seconds.

Methods allow you to set, clear, compare, add, and subtract dates and times in various formats.

**ASExtension**

An `ASExtension` represents a specific plug-in. A unique `ASExtension` object is created for each plug-in when it is loaded. If the plug-in fails to initialize, the `ASExtension` remains, but is marked as dead. The `ASEnumExtensions` method allows you to iterate over all plug-in objects.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCabGetInt</td>
<td>Get various types of values from a cabinet.</td>
</tr>
<tr>
<td>ASCabGetPointer</td>
<td></td>
</tr>
<tr>
<td>ASCabGetStringCopy</td>
<td></td>
</tr>
<tr>
<td>ASCabNew</td>
<td>Creates a new, empty cabinet.</td>
</tr>
<tr>
<td>ASCabRemove</td>
<td>Removes a key from a cabinet, destroying the associated value.</td>
</tr>
</tbody>
</table>
**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.

**ASFile** is related to the **ASFileSys** object, the **ASPlatformPath** object, and the **ASPathName** structure.

An **ASFile** is an opaque representation of an open file. It is similar to an **ASStm**, 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 **ASStm** 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 **ASStm** 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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASFileAcquirePathName</strong></td>
<td>Acquires a file’s path name</td>
</tr>
<tr>
<td><strong>ASFileGetFileSys</strong></td>
<td>Gets the file system on which a file resides</td>
</tr>
<tr>
<td><strong>ASAtomGetString</strong></td>
<td>Gets a string, given an <strong>ASAtom</strong></td>
</tr>
<tr>
<td><strong>ASFileRead</strong></td>
<td>Reads data from a file</td>
</tr>
</tbody>
</table>

**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 **ASFileSysRec** structure contains pointers to callback procedures used by the **ASFileSys** and **ASFile** methods. See “File Systems” on page 171 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 Acrobat’s
device-independent pathname representation to a file system-specific `ASPathName`, or named location for a particular type of device. The `ASPathName` uses the `ASFileSys` structure pointers for callback, and is specific to a given `ASFileSys`.

The device-independent representation is as defined in Section 3.10 in the *PDF Reference*. 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 91 for discussion of `PDFFileSpecHandler`s.

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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ASFileSysCreatePathName</code></td>
<td>Creates an <code>ASPathName</code> based on the input type and file specification.</td>
</tr>
<tr>
<td><code>ASFileSysDIPathFromPath</code></td>
<td>Converts a platform-independent pathname to a platform-dependent pathname.</td>
</tr>
<tr>
<td><code>ASGetDefaultFileSys</code></td>
<td>Gets a platform’s default file system.</td>
</tr>
<tr>
<td><code>ASPathFromPlatformPath</code></td>
<td>Converts a platform-specific pathname to an <code>ASPathName</code>.</td>
</tr>
<tr>
<td><strong>NOTE:</strong></td>
<td>Do not use this method on the Macintosh platform. Instead, call</td>
</tr>
<tr>
<td></td>
<td><code>ASFileSysCreatePathName</code>.</td>
</tr>
</tbody>
</table>
ASPlatformPath

An opaque object used to retrieve a platform-specific path object. This object was added in Acrobat 6.0 to improve device-independent path referencing.

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 160).

ASStm methods allow you to open and close streams, and to read and write data.

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 the following table can be mapped to one of the two cases outlined above.

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoded</td>
<td>A multi-byte string terminated with a single null character and coupled with a specific host encoding indicator.</td>
</tr>
<tr>
<td>ScriptText</td>
<td>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.</td>
</tr>
<tr>
<td>Unicode</td>
<td>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.</td>
</tr>
</tbody>
</table>
ASText 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, for example:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDText</td>
<td>A string of text pulled out of a PDF document. This is either a big-endian</td>
</tr>
<tr>
<td></td>
<td>Unicode string pre-pended with the bytes 0xFE 0xFF or a string in PDF</td>
</tr>
<tr>
<td></td>
<td>document encoding. In the Unicode case, this also may include language</td>
</tr>
<tr>
<td></td>
<td>and country identifiers. ASText objects strip language and country</td>
</tr>
<tr>
<td></td>
<td>information out of the PDText string and track them separately.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASTextCmp</td>
<td>Compares two ASText objects, ignoring language and country information.</td>
</tr>
<tr>
<td>ASTextFromEncoded</td>
<td>Creates a new text object from a null-terminated multi-byte string in the</td>
</tr>
<tr>
<td></td>
<td>specified host encoding.</td>
</tr>
<tr>
<td>ASTextFromSizedEncoded</td>
<td>Creates a new text object from a multi-byte string in the specified host</td>
</tr>
<tr>
<td></td>
<td>encoding and of the specified length.</td>
</tr>
<tr>
<td>ASTextFromSizedUnicode</td>
<td>Creates a new ASText from the specified Unicode string. This string is not</td>
</tr>
<tr>
<td></td>
<td>expected to have 0xFE 0xFF prepended, or country/language identifiers.</td>
</tr>
<tr>
<td>ASTextGetCountry</td>
<td>Gets the country associated with a piece of text stored in an ASText object.</td>
</tr>
</tbody>
</table>
The `ASGetConfiguration` method allows a plug-in to determine the Acrobat viewer under which it is running. Because Adobe 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 representative error methods include:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ASGetExceptionErrorCode</code></td>
<td>Gets the code of the most recently raised exception. For convenience, your plug-in may use the <code>ERRORCODE</code> macro described in “Exception Handlers” on page 185.</td>
</tr>
<tr>
<td><code>ASRaise</code></td>
<td>Raises an exception.</td>
</tr>
<tr>
<td><code>ASRegisterErrorString</code></td>
<td>Registers a defined error and an associated descriptive string.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASFixedRoundToInt32</td>
<td>Converts a fixed point number to an ASInt32 rounding the result.</td>
</tr>
<tr>
<td>ASFixedToFloat</td>
<td>Converts a fixed point number to a floating point number.</td>
</tr>
<tr>
<td>ASFixedTruncToInt32</td>
<td>Converts a fixed point number to an ASInt32 truncating the result.</td>
</tr>
<tr>
<td>ASFloatToFixed</td>
<td>Converts a floating point number to a fixed point number.</td>
</tr>
<tr>
<td>ASInt32ToFixed</td>
<td>Converts an ASInt32 to a fixed point number.</td>
</tr>
</tbody>
</table>

Fixed-point Mathematics Methods

The fixed-point mathematics methods are described here:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASFixedDiv</td>
<td>Divides two fixed point numbers.</td>
</tr>
<tr>
<td>ASFixedMul</td>
<td>Multiplies two fixed point numbers.</td>
</tr>
</tbody>
</table>

Fixed-point Matrix Methods

The fixed-point matrix methods are described here.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASFixedMatrixConcat</td>
<td>Multiplies two matrices.</td>
</tr>
<tr>
<td>ASFixedMatrixInvert</td>
<td>Inverts a matrix.</td>
</tr>
<tr>
<td>ASFixedMatrixTransformRect</td>
<td>Calculates the coordinates of a rectangle's corner points in another coordinate system.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASExtensionMgrGetHFT</td>
<td>Gets an HFT by name.</td>
</tr>
<tr>
<td>HFTNew</td>
<td>Creates an HFT.</td>
</tr>
<tr>
<td>HFTReplaceEntry</td>
<td>Replaces a replaceable method in an HFT.</td>
</tr>
<tr>
<td>HFTServerNew</td>
<td>Creates an HFT server.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASmalloc</td>
<td>Allocates a block of memory.</td>
</tr>
<tr>
<td>ASfree</td>
<td>Frees a block of memory.</td>
</tr>
<tr>
<td>ASrealloc</td>
<td>Reallocates (changes the size of) a block of memory.</td>
</tr>
</tbody>
</table>

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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVAppHandleAppleEvent</td>
<td>Handles an Apple event.</td>
</tr>
<tr>
<td>RectToAVRect</td>
<td>Converts a Macintosh Rect to an AVRect.</td>
</tr>
</tbody>
</table>
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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnixSysGetCwd</td>
<td>Gets the current working directory.</td>
</tr>
</tbody>
</table>

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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WinAppGetModalParent</td>
<td>Gets the appropriate parent for any modal dialogs created by a plug-in.</td>
</tr>
<tr>
<td>WinAppRegisterInterface</td>
<td>Registers a COM interface.</td>
</tr>
<tr>
<td>WinAppRegisterModelessDialog</td>
<td>Registers modeless dialogs with the viewer so that the dialog gets the correct messages.</td>
</tr>
</tbody>
</table>
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 PDFEdit section of the core API, described in Chapter 7, “PDFEdit—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)."

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 **PDTextSelect** object is included in the figure because of its close association with the AV selection methods.

**Figure 5.1 Acrobat Viewer Objects**
NOTE: The AVUndo object, added in Acrobat 6.0, is not shown.

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.

AVALert

AVALert provides platform-independent support for displaying simple dialog boxes. Methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVALert</td>
<td>Displays a dialog box containing a string, icon, and up to three buttons.</td>
</tr>
<tr>
<td>AVALertNote</td>
<td>Displays a dialog box containing a string and an OK button.</td>
</tr>
</tbody>
</table>
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 164 for details on creating new annotation types.

An AVAnnotHandler is generally accessed through AVApp methods, but it has two methods of its own, AVAnnotHandlerGetInfo and AVAnnotHandlerDeleteInfo.

AVApp

AVApp represents the Acrobat viewer application itself. From the application layer, you can control the appearance of Acrobat, whether Acrobat appears, and the size of the application window. Your application has access to the menubar and the toolbar through this object. The application layer also provides access to the visual representation of a PDF file on the screen, that is, an AVDoc.

AVApp methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVAppGetActionHandlerByType</td>
<td>Gets the action handler whose type is specified.</td>
</tr>
<tr>
<td>AVAppGetActiveDoc</td>
<td>Gets the frontmost document view.</td>
</tr>
<tr>
<td>AVAppGetMenubar</td>
<td>Gets the viewer application’s menu bar, toolbar, or a specific tool.</td>
</tr>
<tr>
<td>AVAppGetToolBar</td>
<td></td>
</tr>
<tr>
<td>AVAppGetToolByName</td>
<td></td>
</tr>
<tr>
<td>AVAppGetPreference</td>
<td>Get and set user preferences for the application.</td>
</tr>
<tr>
<td>AVAppSetPreference</td>
<td></td>
</tr>
<tr>
<td>AVAppRegisterCommandHandler</td>
<td>Registers an AVCommand handler to implement an AVCommand with the specified name. See “AVCommand” on page 70 and “AVCommand Handlers” on page 165 for details.</td>
</tr>
<tr>
<td>AVAppRegisterForPageViewClicks</td>
<td>Registers a procedure to call each time a mouse click occurs.</td>
</tr>
<tr>
<td>AVAppRegisterFromPDFHandler</td>
<td>Registers a handler (AVConversionFromPDFHandler or AVConversionToPDFHandler) to export from PDF to other file formats, or to import other file formats to PDF.</td>
</tr>
<tr>
<td>AVAppRegisterToPDFHandler</td>
<td></td>
</tr>
</tbody>
</table>
**AVCommand**

An **AVCommand** represents an action that the user can perform on the current document or the current selection in the current document. An **AVCommand** can be added to a command sequence and executed either interactively or by means of batch processing, using the method **AVCommandExecute**.

**Invoking AVCommands Programmatically**

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

1. Instantiate the command.
   
   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 165.

2. Configure the command by setting required and optional parameters. See Configuring Commands below.

3. Run the command, using **AVCommandExecute** or **AVCommandWork**.

**Configuring Commands**

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.

```c
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,

```c
// 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 Configuration Parameters example, a plug-in can create ASCabs to hold the appropriate parameters; then it can create empty ASText objects to
hold the parameter values and put these values in the ASCabs. The following example uses this approach to set Document Summary title and subject values:

```c
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);
```

**AVCommand Methods**

**AVCommand** methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVCommandCancel</td>
<td>Cancels the specified command.</td>
</tr>
<tr>
<td>AVCommandExecute</td>
<td>A wrapper method that repeatedly calls AVCommandWork until the command returns a status code other than kAVCommandworking.</td>
</tr>
<tr>
<td>AVCommandGetAVDoc</td>
<td>Gets the AVDoc from a command’s inputs ASCab.</td>
</tr>
<tr>
<td>AVCommandGetInputs</td>
<td>Gets or sets the input parameters of the specified command.</td>
</tr>
<tr>
<td>AVCommandSetInputs</td>
<td>Gets or sets the input parameters of the specified command.</td>
</tr>
<tr>
<td>AVCommandGetParams</td>
<td>Gets or sets the parameter set for the specified command.</td>
</tr>
<tr>
<td>AVCommandSetParams</td>
<td>Creates a new command of the specified type.</td>
</tr>
<tr>
<td>AVCommandShowDialog</td>
<td>Instructs the command to bring up its configuration dialog and gather parameter information from the user.</td>
</tr>
</tbody>
</table>

**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 167.
The **AVConversion** methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVConversionConvertToPDFWithHandler</td>
<td>Converts a PDF document to another file format using the specified handler.</td>
</tr>
<tr>
<td>AVConversionConvertToPDF</td>
<td>Converts the specified file to a PDF document using whatever converter is found.</td>
</tr>
<tr>
<td>AVConversionConvertToPDFWithHandler</td>
<td>Converts a file to a PDF document using the specified handler.</td>
</tr>
</tbody>
</table>

### 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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVAuthOpen</td>
<td>Determines if a user is authorized to open a document.</td>
</tr>
<tr>
<td>AVCryptDoStdSecurity</td>
<td>Displays a security dialog to the user, allowing the user to change a document's print, edit, and copy permissions.</td>
</tr>
<tr>
<td>AVCryptGetPassword</td>
<td>Displays the standard dialog box that lets a user enter a password. The PDDocAuthorize or PDDocPermRequest method (see “PDDoc” on page 88) actually checks the password.</td>
</tr>
</tbody>
</table>

### 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 88), an **AVDoc** has a window associated with it. The **AVDoc** methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVDocClose</td>
<td>Closes a document.</td>
</tr>
<tr>
<td>AVDocDoSave</td>
<td>Replacable method that allows plug-ins to implement their own save functionality.</td>
</tr>
<tr>
<td>AVDocFromPDDoc</td>
<td>Gets the <strong>AVDoc</strong> associated with a <strong>PDDoc</strong>.</td>
</tr>
</tbody>
</table>
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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVGrafSelectCreate</td>
<td>Creates a graphics selection.</td>
</tr>
<tr>
<td>AVGrafSelectDestroy</td>
<td>Destroys a graphics selection.</td>
</tr>
<tr>
<td>AVGrafSelectGetBoundingRect</td>
<td>Gets a graphics selection's bounding rectangle.</td>
</tr>
</tbody>
</table>

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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVMenuAddMenuItem</td>
<td>Adds a menu item at a specified location in a menu.</td>
</tr>
<tr>
<td>AVMenuGetName</td>
<td>Gets the language-independent menu name.</td>
</tr>
<tr>
<td>AVMenuNew</td>
<td>Creates a new menu.</td>
</tr>
<tr>
<td>AVMenuRemove</td>
<td>Removes a menu from the menu bar.</td>
</tr>
<tr>
<td>AVMenuRelease</td>
<td>Releases a previously acquired menu.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVMenuBarAcquireMenuByName</td>
<td>Acquires the menu with the specified name.</td>
</tr>
<tr>
<td>AVMenuBarAcquireMenuItemByName</td>
<td>Acquires the menu item with the specified name.</td>
</tr>
<tr>
<td>AVMenuBarAddMenu</td>
<td>Adds a menu to the menubar.</td>
</tr>
<tr>
<td>AVMenuBarHide</td>
<td>Hides the menubar.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AVMenuItemExecute</code></td>
<td>Executes a menu item’s action by invoking the <code>ExecuteProc</code> procedure.</td>
</tr>
<tr>
<td><code>AVMenuItemGetName</code></td>
<td>Gets the language-independent name of the menu item.</td>
</tr>
<tr>
<td><code>AVMenuItemNew</code></td>
<td>Creates a new menu item.</td>
</tr>
<tr>
<td><code>AVMenuItemSetExecuteProc</code></td>
<td>Sets the procedure called when the menu item is selected by the user.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVPageViewGetFirstVisiblePageNum</td>
<td>Returns the page number of the first page visible</td>
</tr>
<tr>
<td>AVPageViewGetPage</td>
<td>Gets the PDPage for a page view</td>
</tr>
<tr>
<td>AVPageViewPointToDevice</td>
<td>Transforms a points coordinates from user space to device space. See “Understanding Coordinate Systems” on page 28.</td>
</tr>
<tr>
<td>AVPageViewZoomTo</td>
<td>Sets the zoom factor</td>
</tr>
</tbody>
</table>

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 ADM Programmer’s Guide and Reference. The AVSweetPea methods include, for example:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVSweetPeaGetBasicSuiteP</td>
<td>Accesses the basic ADM suite.</td>
</tr>
<tr>
<td>AVSweetPeaGetPluginRef</td>
<td>Gets a reference to the ADM plug-in itself (not the plug-in you are currently developing).</td>
</tr>
<tr>
<td>AVSweetPeaIsADMAvailable</td>
<td>Determines whether ADM is available.</td>
</tr>
</tbody>
</table>
AVSys

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVSysAllocTimeStringFromTimeRec</td>
<td>Gets a string representing the date and time.</td>
</tr>
<tr>
<td>AVSysBeep</td>
<td>Beeps.</td>
</tr>
<tr>
<td>AVSysGetStandardCursor</td>
<td>Gets the specified cursor.</td>
</tr>
<tr>
<td>AVSysMouseIsStillDown</td>
<td>Tests whether the mouse button is still being pressed.</td>
</tr>
</tbody>
</table>

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 79). 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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVToolGetType</td>
<td>Gets a tool’s type.</td>
</tr>
<tr>
<td>AVToolIsPersistent</td>
<td>Indicates whether a tool is persistent or is one shot.</td>
</tr>
</tbody>
</table>

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, for example:

<table>
<thead>
<tr>
<th>AVToolBarAddButton</th>
<th>Adds a button to the toolbar.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVToolBarGetButtonByName</td>
<td>Gets the button that has the specified name.</td>
</tr>
<tr>
<td>AVToolBarNewFlyout</td>
<td>Creates a new sub-toolbar for use as a tool button flyout.</td>
</tr>
</tbody>
</table>

**AVToolButton**

An AVToolButton 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 78). 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.
ACROBAT VIEWER LAYER

AVUndo

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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVToolButtonExecute</td>
<td>Executes a button’s action by invoking the AVExecuteProc callback.</td>
</tr>
<tr>
<td>AVToolButtonGetName</td>
<td>Gets the name of a button.</td>
</tr>
<tr>
<td>AVToolButtonNew</td>
<td>Creates a new button.</td>
</tr>
<tr>
<td>AVToolButtonRemove</td>
<td>Removes (but does not destroy) a button from the toolbar.</td>
</tr>
<tr>
<td>AVToolButtonSetExecuteProc</td>
<td>Sets the procedure to call when the user clicks the button.</td>
</tr>
<tr>
<td>AVToolButtonSetFlyout</td>
<td>Attaches a sub-toolbar to a tool button.</td>
</tr>
</tbody>
</table>

AVUndo

The AVUndo object represents an undo record for a document. An undo record allows a client to associate private data with a particular AVDoc for the purpose of undoing and redoing changes to the document. The client provides private data that encapsulates the changes, and an AVUndoHandler that contains callbacks which interpret the data when Undo and Redo commands are issued.

The AVUndo object itself has an AVUndoNew method, and methods for getting and setting properties, such as AVUndoGetData and AVUndoSetData. However, you initiate and control an undo or redo operation from the document (which you can obtain with AVUndoGetAVDoc), using methods such as AVDocBeginUndoOperation, AVDocEndUndoOperation, and AVDocClearUndos.

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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>AVWindowNew</em></td>
<td>Creates a new window.</td>
</tr>
<tr>
<td><em>AVWindowNewFromPlatformThing</em></td>
<td>Creates a new window from a platform-native window pointer or handle.</td>
</tr>
<tr>
<td><em>AVWindowHide</em></td>
<td>Hides a window.</td>
</tr>
<tr>
<td><em>AVWindowDrawNow</em></td>
<td>Updates a window.</td>
</tr>
</tbody>
</table>
Portables Document Layer

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 PDTextSelect object is shown in Figure 5.1, “Acrobat Viewer Objects.”

NOTE: New objects added in Acrobat 6.0 that implement optional content (layers) are not shown. These include:

PDOCCConfig
PDOCCContext
PDOCG
PDOCMD

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 bookkeeping 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 complete 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, for example:

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

For a complete listing, see General PD functions in the Acrobat Core API Reference.

### 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 Adobe's Extensible Metadata Platform (XMP) (formerly referred to as “XAP”).

See Section 10.2 in the PDF Reference for details about the use of metadata in PDF 1.4 and later.

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):

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

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 8.5 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 164) that are responsible for interpreting an action's data and carrying out the action. **PDAction** methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDActionGetDest</td>
<td>Gets an action's destination view.</td>
</tr>
<tr>
<td>PDActionGetSubtype</td>
<td>Gets an action's subtype.</td>
</tr>
<tr>
<td>PDActionNew</td>
<td>Creates a new action.</td>
</tr>
</tbody>
</table>

**PDAnnot**

This is the abstract superclass for all annotations (see Section 8.4, “Annotations,” in the *PDF Reference*). Acrobat viewers have two built-in annotation classes: **PDTextAnnot** 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 164). There are no objects of type **PDAnnot**, but you-in 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**
- **CastToPDTextAnnot**
- **CastToPDLinkAnnot**

The **PDAnnot** methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDAnnotGetRect</td>
<td>Gets an annotation's size and location.</td>
</tr>
</tbody>
</table>
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 8.3.2, “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, for example:

- 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 8.2.2, “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 86) 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

**PDBBookmark** methods include the following:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDBBookmarkAddNewChild</td>
<td>Adds a new child to a bookmark.</td>
</tr>
<tr>
<td>PDBBookmarkAddNewSibling</td>
<td>Adds a new sibling to a bookmark.</td>
</tr>
<tr>
<td>PDBBookmarkDestroy</td>
<td>Destroys a bookmark and all of its children.</td>
</tr>
<tr>
<td>PDBBookmarkFromCosObj</td>
<td>Converts an appropriate Cos object to a bookmark.</td>
</tr>
<tr>
<td>PDBBookmarkGetAction</td>
<td>Gets a bookmark’s action.</td>
</tr>
<tr>
<td>PDBBookmarkGetCosObj</td>
<td>Gets the Cos object associated with a bookmark.</td>
</tr>
<tr>
<td>PDBBookmarkGetParent</td>
<td>Gets a bookmark’s parent bookmark.</td>
</tr>
<tr>
<td>PDBBookmarkSetOpen</td>
<td>Opens or closes a bookmark.</td>
</tr>
<tr>
<td>PDBBookmarkSetTitle</td>
<td>Sets a bookmark’s title.</td>
</tr>
</tbody>
</table>

**PDCharProc**

A **PDCharProc** is a character procedure, a stream of graphic operators (see “PDGraphic” on page 94) 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 **I**. 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 **PDFFontEnumCharProcs** 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 **ASFile**. Also, every **AVDoc** has an associated **PDDoc**, although a **PDDoc** may not be associated with an **AVDoc**.
NOTE: An **ASFile** may have zero or more underlying files, so a PDF file does not always correspond to a single disk file. For example, an **ASFile** may provide access to PDF data in a database.

A plug-in may create a new document or open a document using an **ASFileSys** and an **ASPlatformPath**. 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 **PDInsertAll** flag. Using this flag wipes out the previous forms data and replaces it with the information from the file being inserted.

**Querying PDDoc Permissions**


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
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 (`PDPermReqOpr`) 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 `PD Cryp tHandler` to support the finer granularity of permissions that plug-ins can query.

**PDDoc Methods**

`PDDoc` methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDDocAuthorize</code></td>
<td>Adds permissions to the specified document, if permitted.</td>
</tr>
<tr>
<td><code>PDDocClose</code></td>
<td>Closes an open document.</td>
</tr>
<tr>
<td><code>PDDocCreate</code></td>
<td>Creates a new document.</td>
</tr>
<tr>
<td><code>PDDocCreatePage</code></td>
<td>Creates a new page.</td>
</tr>
<tr>
<td><code>PDDocCreateStructTreeRoot</code></td>
<td>Creates a new <code>StructTreeRoot</code> element.</td>
</tr>
<tr>
<td><code>PDDocCreateWordFinderUCS</code></td>
<td>Creates a word finder (see “PDWordFinder” on page 106) for extracting text in UCS format from a PDF file.</td>
</tr>
<tr>
<td><code>PDDocGetInfo</code></td>
<td>Gets a value from a document’s Info dictionary.</td>
</tr>
<tr>
<td><code>PDDocInsertPages</code></td>
<td>Inserts pages from another document.</td>
</tr>
<tr>
<td><code>PDDocGetNumPages</code></td>
<td>Gets the number of pages in the document.</td>
</tr>
<tr>
<td><code>PDDocGetWordFinder</code></td>
<td>Gets the word finder associated with a document.</td>
</tr>
<tr>
<td><code>PDDocOpen</code></td>
<td>Opens a <code>PDDoc</code> from an ASFileSys and an ASPathName.</td>
</tr>
</tbody>
</table>
**PDFSpec**

A **PDFSpec** 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 86). 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

**PDFSpecs** can be created from **ASPathName**s or from Cos objects. Methods are also provided to get **ASPathName**s and device-independent pathnames from **PDFSpecs**. The **PDFSpec** methods include, for example:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PDFSpecAcquireASPath</strong></td>
<td>Acquires an <strong>ASPathName</strong> for the specified file specification and relative path.</td>
</tr>
<tr>
<td><strong>PDFSpecGetDIPath</strong></td>
<td>Gets the device-independent pathname from a file specification.</td>
</tr>
<tr>
<td><strong>PDFSpecGetDoc</strong></td>
<td>Gets the <strong>PDDoc</strong> that contains the file specification.</td>
</tr>
<tr>
<td><strong>PDFSpecGetFileSyncName</strong></td>
<td>Gets the name of the file system to which the <strong>PDFSpec</strong> belongs.</td>
</tr>
</tbody>
</table>

**PDFont**

A **PDFont** is a font that is used to draw text on a page. It corresponds to a font resource in a PDF file (see Chapter 5, “Text,” in the PDF Reference).

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

A **PDFont** 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 **PDFont** 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, MMType1, 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 CMaps.

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, “CIDSystemInfo 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:

<table>
<thead>
<tr>
<th>Technical Note #</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5092</td>
<td>CID-Keyed Font Technology Overview</td>
</tr>
<tr>
<td>5014</td>
<td>Adobe CMap and CIDFont Files Specification</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Technical Note #</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5078</td>
<td>Adobe-Japan1-2 Character Collection for CID-Keyed Fonts</td>
</tr>
<tr>
<td>5079</td>
<td>Adobe-GB1-0 Character Collection for CID-Keyed Fonts</td>
</tr>
<tr>
<td>5080</td>
<td>Adobe-CNS1-0 Character Collection for CID-Keyed Fonts</td>
</tr>
<tr>
<td>5093</td>
<td>Adobe-Korea1-0 Character Collection for CID-Keyed Fonts</td>
</tr>
<tr>
<td>5094</td>
<td>Adobe CJK Character Collections and CMaps for CID-Keyed Fonts</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDFontGetName</td>
<td>Gets the PostScript name for a Type 1 or Type 3 font, and the “styled” name for a TrueType font.</td>
</tr>
<tr>
<td>PDFontGetSubtype</td>
<td>Gets the font’s subtype.</td>
</tr>
<tr>
<td>PDFontGetWidths</td>
<td>Gets a font’s character widths.</td>
</tr>
</tbody>
</table>

---

**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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDFormEnumPaintProc</td>
<td>Enumerates a form’s drawing operations.</td>
</tr>
<tr>
<td>PDFormGetBBox</td>
<td>Gets a form’s bounding box.</td>
</tr>
<tr>
<td>PDFormGetMatrix</td>
<td>Gets a form’s transformation matrix.</td>
</tr>
</tbody>
</table>

---

**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: **PDPath**, **PDText**, and **PDInlineImage**. In addition to these three objects, there are also operators in the content stream. These operators are: **Save**, **Restore**, references to **XObject**s (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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDGraphicGetBBox</code></td>
<td>Gets a graphic’s bounding box.</td>
</tr>
<tr>
<td><code>PDGraphicGetCurrentMatrix</code></td>
<td>Gets the current transformation matrix in effect for a graphic object.</td>
</tr>
<tr>
<td><code>PDGraphicGetState</code></td>
<td>Gets the graphics state associated with a graphic object.</td>
</tr>
</tbody>
</table>

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*.

---

**PDImage**

A `PDImage` 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 `PDImage`. The `PDImage` methods include the following:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDImageGetAttrs</code></td>
<td>Gets the attributes of an image.</td>
</tr>
<tr>
<td><code>PDImageSelectAlternate</code></td>
<td>Selects an alternate image to use.</td>
</tr>
</tbody>
</table>

---

**PDInlineImage**

A `PDInlineImage` is an image whose data is stored in the page description's contents stream instead of being stored as an image resource (see `PDImage`). `PDInlineImage` 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 `PDInlineImage` methods include the following:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDInlineImageGetAttrs</code></td>
<td>Gets an inline image’s attributes.</td>
</tr>
</tbody>
</table>
A **PDLinkAnnot** corresponds to a link annotation (see Sections 8.4.5, "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 **PDPage** methods.

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

- **CastToPDAnnot**
- **CastToPDTextAnnot**
- **CastToPDLinkAnnot**

The **PDLinkAnnot** methods include, for example:

- **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, for example:

- **PDNameTreeGet** Retrieves an object from the name tree.
- **PDNameTreeNew** Creates a new name tree in the document.
PDNumTree

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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDNumTreeFromCosObj</td>
<td>Creates a type cast of a CosObj to a number tree.</td>
</tr>
<tr>
<td>PDNumTreePut</td>
<td>Puts a new entry in the number tree. If an entry with this number is already in the tree, it is replaced.</td>
</tr>
</tbody>
</table>

PDOCCConfig

The PDOCCConfig object represents an optional-content configuration structure, used to maintain a set of visibility states and other optional-content information in a PDF file for future use. A document has a default configuration, saved in the D entry in the OCProperties dictionary, and can have a list of other configurations, saved as an array in the Configs entry in the OCProperties dictionary.

Configurations are typically used to initialize the optional-content group (PDOCG) ON-OFF states for an optional-content context (PDOCCContext). The OCG order in the configuration is the order in which the groups appear in the Layers panel of Acrobat 6.0. The configuration can define a set of mutually exclusive OCGs, called a radio button group.

PDOCCConfig methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDOCCConfigSetInitState</td>
<td>Sets the initial ON-OFF states of optional-content groups in an optional-content configuration.</td>
</tr>
<tr>
<td>PDOCCConfigSetOCGOrder</td>
<td>Sets the UI-display order of optional-content groups (OCGs) in an optional-content configuration. This is the order in which the group names are displayed in the Layers panel of Acrobat 6.0.</td>
</tr>
</tbody>
</table>

PDOCCContext

The PDOCCContext object represents an optional-content context in a document, within which document objects such as words or annotations are visible or hidden. The context keeps track the ON-OFF states of all of the optional-content groups (OCGs, represented by the PDOCG object) in a document. Content is or is not visible with respect to the OCG states stored in a specific context. The context does not correspond to any explicit PDF specification.
The **PDDoc** has a default context that it uses for on-screen drawing and that determines the default state for any other drawing or content enumeration. The context has flags that control whether to draw or enumerate content that *is* marked as optional (**PDOCDrawEnumType**), and whether to draw content that is *not* marked as optional (**NonOCDrawing**).

There can be more than one **PDOCContext** object, representing different combinations of OCG states. You can change the states of OCGs within any context. You can build contexts with your own combination of OCG states, and issue drawing or enumeration commands using that context instead of the document’s default context. For example, you can pass an optional-content context to **PDPageDrawContentsWithParams** through the **PDDrawParams** structure. You can save the resulting state information as part of the configuration, but the context itself has no corresponding PDF form, and is not saved.

**PDOCContext** methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PDOCContextSetOCGStates</strong></td>
<td>Sets the ON-OFF states for the given optional-content groups (OCGs) in the given optional-content context.</td>
</tr>
<tr>
<td><strong>PDOCContextApplyAutoStateChanges</strong></td>
<td>Finds optional-content groups whose ON-OFF states should be toggled, based on usage application directives contained in the configuration’s <strong>AS</strong> array, and applies the changes within the given context. The <strong>AS</strong> array defines how usage entries are used to automatically manipulate the OCG states. It associates an event (<strong>View</strong>, <strong>Print</strong>, or <strong>Export</strong>) with a list of OCGs and a category, or list of usage keys identifying OCG usage dictionary entries.</td>
</tr>
</tbody>
</table>

**PDOCG**

The **PDOCG** object represents an optional-content group. This corresponds to a PDF **OCG** dictionary representing a collection of graphic objects that can be made visible or invisible. Any graphic content of the PDF can be made optional, including page contents, XObjects, and annotations. The specific content objects in the group have an **OC** entry in the PDF. The group itself is a named object that you can manipulate in the **Layers** panel of Acrobat 6.0.

In the simplest case, the group’s ON-OFF state makes the associated content visible or hidden. The ON-OFF state of a group can be toggled for a particular context (**PDOCContext**), and a set of states is kept in a configuration (**PDOCConfig**). The visibility can depend on more than one group in an optional-content membership dictionary.
PDOCDrawEnumType.

A group has an Intent entry, an ASAtom value broadly describing the intended use, either View or Design. A group’s content is considered to be optional (that is, the group’s state is considered in its visibility) if any intent in its list matches an intent of the context. The intent list of the context is usually set from the intent list of the document configuration.

A Usage dictionary entry provides more specific intended usage information than an intent entry. Possible key values are:

- CreatorInfo
- Language
- Export
- Zoom
- Print
- View
- User
- PageElement

The usage value can act as a kind of metadata, describing the sort of things that belong to the group: for example, text in French, fine detail on a map, or a watermark. The usage values can also be used by the AutoState mechanism to make decisions about what groups should be on and what groups should be off. The AutoState mechanism considers the usage information in the OCGs, the AS array of the configuration, and external factors; for example, the language the app is running in, the current zoom level on the page, or if the page is being printed.

PDOCG methods include, for example:

- PDOCGSetCurrentState
- PDOCGSetInitialState
- PDOCGSetUsageDictEntry

PDOCMDD

The PDOCMD object represents an optional-content membership dictionary (OCMD) that allows the visibility of optional content to depend on the states in a set of optional-content groups (PDOCG). The object corresponds to the PDF OCMD dictionary.

An OCMD collects a set of OCGs. It sets a visibility policy, so that content in the member groups is visible only when all groups are ON or OFF, or when any of the groups is ON or OFF. This makes it possible to set up complex dependencies among groups.
An optional-content context can contain a stack of OCMD objects, which you can manipulate with `PDCCContextPopOCMD`, `PDCCContextPushOCMD`, and `PDCCContextResetOCMDStack`.

An OCMD can be associated with an annotation (`PDAannotSetOCMD`) or a PDE element (`PDEElementSetOCMD`).

`PDCCMD` methods include, for example:

- `PDCCMDGetVisPolicy` Gets the optional-content membership dictionary's visibility policy, which determines the visibility of content with respect to the ON-OFF state of OCGs listed in the dictionary.
- `PDCCMDSMakeContentVisible` Makes content that uses any of a set of optional-content membership dictionaries visible in a given optional-content context. The method manipulates the states of optional-content groups in the dictionaries so that any content controlled by any of the dictionaries will be visible in the given context. There can be more than one combination of states that satisfies the request.

---

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

`PDPage` methods include, for example:

- `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.
**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDPageLabelEqual</code></td>
<td>Compares two page labels to see if they are equivalent.</td>
</tr>
<tr>
<td><code>PDPageLabelGetStart</code></td>
<td>Gets the starting page number for the page label specified.</td>
</tr>
</tbody>
</table>

**PDPath**

A `PDPath` 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”

`PDPath` methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDPathEnum</code></td>
<td>Enumerates a path’s operators, calling one of several user-supplied callbacks for each.</td>
</tr>
<tr>
<td><code>PDPathGetPaintOp</code></td>
<td>Determines which paint/close/clip operators are used for the path.</td>
</tr>
</tbody>
</table>
**PDStyle**

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDStyleGetColor</td>
<td>Gets a style's color.</td>
</tr>
<tr>
<td>PDStyleGetFontSize</td>
<td>Gets a style's font size.</td>
</tr>
</tbody>
</table>

**PDText**

A **PDText** 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 **PDText** 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. **PDText** methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDTextEnum</td>
<td>Enumerates the strings of a text object.</td>
</tr>
<tr>
<td>PDTextGetState</td>
<td>Gets the text state for a text object.</td>
</tr>
</tbody>
</table>

**PDTextAnnot**

A **PDTextAnnot** corresponds to a PDF text annotation. For details, see “Text Annotations” in Section 8.4.5, “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**
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 PDTextSelectRangeRec 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 PDTextSelectRangeRec structures to PDTextSelectCreateRanges
- Supply a list of word highlights to PDTextSelectCreateWordHilite

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 PDTextSelectCreatePageHilite.

Once a plug-in creates a text selection, it can make it the current selection using AVDocSetSelection.
**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 8.3.2, “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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PDThreadNew</strong></td>
<td>Creates a thread.</td>
</tr>
<tr>
<td><strong>PDThreadGetFirstBead</strong></td>
<td>Gets an article thread’s first bead.</td>
</tr>
<tr>
<td><strong>PDThreadIsValid</strong></td>
<td>Tests whether a thread is valid.</td>
</tr>
</tbody>
</table>

**PDThumb**

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

**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).
**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 8.3.3, “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 8.2, “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, for example:

- **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 102).

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 cannot 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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDWordGetAttr</td>
<td>Gets a bit field containing information on the types of characters in a word.</td>
</tr>
<tr>
<td>PDWordGetCharacterTypes</td>
<td>Gets the character type for each character in a word.</td>
</tr>
<tr>
<td>PDWordGetCharOffset</td>
<td>Returns the offset of a word from the beginning of the page.</td>
</tr>
<tr>
<td>PDWordGetString</td>
<td>Converts a word to a null-terminated string and converts ligatures to their constituent characters.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDWordFinderAcquireWordList</code></td>
<td>Finds all words on a page, and returns one or more tables containing the words.</td>
</tr>
<tr>
<td><code>PDWordFinderGetNthWord</code></td>
<td>Gets the nth word in the word list obtained using <code>PDWordFinderAcquireWordList</code>.</td>
</tr>
</tbody>
</table>

**PDXObject**

This object corresponds to a PDF XObject (see Section 4.9, “Form XObjects,” in the PDF Reference). `PDXObject` objects currently used by Acrobat viewers are of one of the two XObject subclasses: `PDImage` and `PDForm`. You can use any `PDXObject` method on these objects.

The `PDXObject` methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDXObjectGetData</code></td>
<td>Passes the data from an XObject to a user-supplied procedure.</td>
</tr>
<tr>
<td><code>PDXObjectGetSubtype</code></td>
<td>Gets the subtype of an XObject (PDImage or PDForm).</td>
</tr>
</tbody>
</table>
Portable Document Layer

PDXObject
PDFEdit—Creating and Editing Page Content

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 section 3.6.2 in the PDF Reference). It can also handle Form XObject appearances represented by the AP (appearances) key in an annotation dictionary. See section 8.4.1 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:
PDFEdit—Creating and Editing Page Content

Overview of PDFEdit

- 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, “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.
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:

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

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:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEClip</td>
<td>Container of PDEPath and PDEText objects describing paths and charpaths.</td>
</tr>
<tr>
<td>PDEColorSpace</td>
<td>Color space attribute of a PDEElement. Color spaces are listed in page ColorSpace resources.</td>
</tr>
</tbody>
</table>
Example

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

```c
/* 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 */
ASUns8 hello[] = "Hello!";
PDETextAdd(pdeText, kPDETextRun, 0, hello, 6, pdeFont, &gstateP, sizeof(gstateP), NULL, 0, &m, NULL);
PDEContentAddElem(pdeContent, kPDEAfterLast, (PDEElement)pdeText);
PDERelease(pdeText);
```

<table>
<thead>
<tr>
<th><strong>PDEExtGState</strong></th>
<th>Extended Graphics State attribute of a PDEElement. ExtGStates are listed in page ExtGState resources.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PDEFont</strong></td>
<td>A font. Part of a text's font information is listed in page Font resources.</td>
</tr>
</tbody>
</table>
/* 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 PDFont 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 PDEText object corresponding to this text also has a set of attributes. However, these attributes are get and set with PDEText 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 PDETextCreate. 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 PDETextRunSetFont.
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 \texttt{ASStm} object methods in the core API; but as noted in “Why PDFEdit?” on page 109, the streams are not easy to manipulate.

\texttt{ASStm} 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 \texttt{PDEContent} object, containing various \texttt{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 \texttt{PDEElement}s. 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.

\begin{verbatim}
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) == kPDEText)
        ...
\end{verbatim}

\textbf{NOTE:} Two core PD layer methods were available in pre-4.0 Acrobat to describe the objects in a page description, namely: \texttt{PDPageEnumContents} and \texttt{PDPageEnumResources}. 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 **PDPages**. 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 **PDEText** 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 **PDPage'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 **PDPage**, and then calling **PDWordFinderAcquireWordList**.

The design of PDFEdit assumes that characters and show strings—not words—are the basic elements of a **PDEText** 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 PDPageAddCosContents

**PDPageAddCosContents** 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 **PDPageGetCosObj** 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. **PDETextIsAtPoint** and **PDETextIsAtRect** 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.
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; PDETextRunSetMatrix 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 10.5, “Marked Content,” in the PDF Reference. The PDEPlace object
marks a location in a sequence of **PDEElement**s. The **PDEContainer** object marks a
group of 0 or more **PDEElement**s.

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

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

**Cos Objects and Documents**

Many of the **PDEObject**s 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.

**XObject**s and **PDEObject**s

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 **PDEElement**s 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.

**NOTE:** For the Adobe PDF Library, use a consistent value for the client ID throughout your code; the value 0 is recommended.

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:

```c
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=FONTNAME */
/* 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 *)FONTNAME);

/* 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(FONTNAME);
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 PDEText object as a text run, */
/* and add to PDEContent */
PDETextAdd(pdeText, kPDETextRun, 0, (unsigned char *)buf, strlen(buf), gFont, &gstate, sizeof(gstate), &tState, sizeof(PDETextState), &matrix, NULL);
/* Add to end of contents */
PDEContentAddElem(pdecontent, kPDEAfterLast, (PDEElement)pdeText);

/* Release the PDEText 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 */
pPath = 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 **PDPlace** 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 10.5, “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, \texttt{pdecontent}, to a file:

```c
/* Dump the content object */
PDEContent pdecontent;
...
PDEObjectDump((PDEObject)pdecontent, 10,
    ASCallbackCreateProto(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 \texttt{PDEContent} object, showing its flags and number of elements, among other information. It also includes the objects inside the \texttt{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
```
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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PDELogDump</strong></td>
<td>Enumerates <strong>PDEObjects</strong>.</td>
</tr>
</tbody>
</table>
## 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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEDefaultGState</td>
<td>Fills out a structure with the default graphic state.</td>
</tr>
<tr>
<td>PDEEnumElements</td>
<td>Enumerates all PDEElement objects in a given stream.</td>
</tr>
<tr>
<td>PDEMergeResourcesDict</td>
<td>Merges two resources dictionaries.</td>
</tr>
</tbody>
</table>

## 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 a PDEClip associated with it by using the PDEElementSetClip method. The PDEClip methods include the following:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEClipAddElem</td>
<td>Adds an element to a clipping path.</td>
</tr>
<tr>
<td>PDEClipCreate</td>
<td>Creates an empty clip object.</td>
</tr>
<tr>
<td>PDEClipGetElem</td>
<td>Gets an element from a clip object.</td>
</tr>
<tr>
<td>PDEClipGetNumElems</td>
<td>Gets the number of path and charpath elements in a clip object.</td>
</tr>
<tr>
<td>PDEClipRemoveElems</td>
<td>Removes one or more elements from a clip object.</td>
</tr>
</tbody>
</table>

## 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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEColorSpaceCreate</td>
<td>Creates a new color space object of the specified type.</td>
</tr>
</tbody>
</table>
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 Section 10.5, “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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEContainerCreate</td>
<td>Creates a container object.</td>
</tr>
<tr>
<td>PDEContainerGetContent</td>
<td>Gets the PDEContent for a PDEContainer.</td>
</tr>
<tr>
<td>PDEContainerGetDict</td>
<td>Gets the marked content dictionary for a container.</td>
</tr>
<tr>
<td>PDEContainerGetMCTag</td>
<td>Obtains the marked content tag for a container.</td>
</tr>
<tr>
<td>PDEContainersetContent</td>
<td>Sets the content for a container.</td>
</tr>
</tbody>
</table>

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 PDEText or PDEPath.

The PDEContent methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEContentAddElem</td>
<td>Inserts an element into a PDEContent.</td>
</tr>
<tr>
<td>PDEContentCreate</td>
<td>Creates an empty content object.</td>
</tr>
<tr>
<td>PDEContentGetElem</td>
<td>Gets the requested element from a content.</td>
</tr>
<tr>
<td>PDEContentGetNumElems</td>
<td>Gets the number of elements in a PDEContent.</td>
</tr>
<tr>
<td>PDEContentGetResources</td>
<td>Obtains the number of resources of a specified type and, optionally, pointers to the resource objects.</td>
</tr>
<tr>
<td>PDEContentRemoveElem</td>
<td>Removes an element from a PDEContent.</td>
</tr>
</tbody>
</table>
**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEDeviceNColorsCreate</td>
<td>Creates an object that can be used to store ( n ) color components when in a <strong>PDEDeviceNColors</strong> color space.</td>
</tr>
<tr>
<td>PDEDeviceNColorsGetColorValue</td>
<td>Gets the value of a color component of a <strong>PDEDeviceNColors</strong> color space.</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEElementCopy</td>
<td>Makes a copy of an element.</td>
</tr>
<tr>
<td>PDEElementGetBBox</td>
<td>Obtains the bounding box for an element.</td>
</tr>
<tr>
<td>PDEElementGetClip</td>
<td>Gets the current clip for an element.</td>
</tr>
<tr>
<td>PDEElementGetGState</td>
<td>Obtains the graphics state information for an element.</td>
</tr>
<tr>
<td>PDEElementGetMatrix</td>
<td>Obtains the transformation matrix for an element.</td>
</tr>
</tbody>
</table>
PDFEdit—Creating and Editing Page Content

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 4.3.4 in the `PDF Reference`, for information on extended graphics states.

```c
// 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 | kPDEFlatnessWasSet |
                       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:

```c
// 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 Chapter 7 in the `PDF Reference`.

PDEExtGState Methods

The `PDEExtGState` methods include the following:

| `PDEExtGStateCreate`          | Creates a new `PDEExtGState` from a `Cos` object.  |
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 **PDFont**; a **PDEFont** is associated with a particular document.

See Chapter 5 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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEFontCreate</td>
<td>Creates a new <strong>PDEFont</strong> from specified parameters.</td>
</tr>
<tr>
<td>PDEFontCreateFromCosObj</td>
<td>Creates a <strong>PDEFont</strong> corresponding to a Cos object.</td>
</tr>
<tr>
<td>PDEFontCreateWithParams</td>
<td>Creates a new <strong>PDEFont</strong> from parameters.</td>
</tr>
<tr>
<td>PDEFontGetAttrs</td>
<td>Obtains the attributes for a font object.</td>
</tr>
<tr>
<td>PDEFontGetCosObj</td>
<td>Obtains a Cos object for a <strong>PDEFont</strong>.</td>
</tr>
<tr>
<td>PDEFontGetWidths</td>
<td>Gets the widths for a font object.</td>
</tr>
<tr>
<td>PDEFontSubsetNow</td>
<td>Subsets a <strong>PDEFont</strong> in a CosDoc.</td>
</tr>
</tbody>
</table>

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*. A `PDEContent` may be obtained from a `PDEForm` to edit the form’s display list. The `PDEForm` methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDEFormCreateFromCosObj</code></td>
<td>Creates a new form from a Cos object.</td>
</tr>
<tr>
<td><code>PDEFormGetContent</code></td>
<td>Obtains a <code>PDEContent</code> object for a form.</td>
</tr>
<tr>
<td><code>PDEFormGetCosObj</code></td>
<td>Obtains a Cos object from a form.</td>
</tr>
</tbody>
</table>

---

**PDEGroup**

An in-memory representation of objects in a `PDEContent`. It has no state and is not represented in any way in a PDF content stream (that is, `PDEContent`).

When used in a `PDEClip`, this object is used to associate `PDEText` objects into a single clipping object. The `PDEGroup` methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDEGroupCreate</code></td>
<td>Creates a <code>PDEGroup</code> object.</td>
</tr>
<tr>
<td><code>PDEGroupSetContent</code></td>
<td>Sets the <code>PDEContent</code> for a <code>PDEGroup</code>. The existing <code>PDEContent</code> is released by this method.</td>
</tr>
</tbody>
</table>

---

**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 `PDEImageSetDataStm` 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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PDEImageCreate</strong></td>
<td>Creates an image object from a stream or buffer of image data.</td>
</tr>
<tr>
<td><strong>PDEImageCreateFromCosObj</strong></td>
<td>Creates an image object from a Cos object.</td>
</tr>
<tr>
<td><strong>PDEImageDataIsEncoded</strong></td>
<td>Determines if image data is encoded.</td>
</tr>
<tr>
<td><strong>PDEImageGetAttrs</strong></td>
<td>Obtains attributes for an image.</td>
</tr>
<tr>
<td><strong>PDEImageGetData</strong></td>
<td>Gets an image's data.</td>
</tr>
<tr>
<td><strong>PDEImageGetFilterArray</strong></td>
<td>Obtains the filter array for an image.</td>
</tr>
<tr>
<td><strong>PDEImageIsCosObj</strong></td>
<td>Determines if an image is represented by a Cos object.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PDEAcquire</strong></td>
<td>Increments the reference count for an object.</td>
</tr>
<tr>
<td><strong>PDEAddTag</strong></td>
<td>Adds an identifier–value pair to an object.</td>
</tr>
<tr>
<td><strong>PDEGetTag</strong></td>
<td>Obtains an object's value for a given client ID.</td>
</tr>
<tr>
<td><strong>PDERemoveTag</strong></td>
<td>Removes an object’s value for a given client ID.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEPathAddSegment</td>
<td>Adds a segment to a path.</td>
</tr>
<tr>
<td>PDEPathCreate</td>
<td>Creates an empty path element.</td>
</tr>
<tr>
<td>PDEPathGetData</td>
<td>Obtains size of path data and, optionally, path data.</td>
</tr>
<tr>
<td>PDEPathSetPaintOp</td>
<td>Sets fill and stroke attributes of a path.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEPatternCreate</td>
<td>Creates a pattern object.</td>
</tr>
<tr>
<td>PDEPatternGetCosObj</td>
<td>Obtains a Cos object corresponding to a pattern object.</td>
</tr>
</tbody>
</table>

**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.

See Section 10.5, “Marked Content,” in the PDF Reference, for information on marked content.

A PDEPlace object allows marking a particular group of objects in a PDF file, rather than a place, with information.

PDEPlace methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEPlaceCreate</td>
<td>Creates a place object.</td>
</tr>
<tr>
<td>PDEPlaceGetDict</td>
<td>Obtains the marked content dictionary for a PDEPlace.</td>
</tr>
<tr>
<td>PDEPlaceSetMCTag</td>
<td>Sets the marked content tag for a PDEPlace.</td>
</tr>
</tbody>
</table>
PDEPS

A **PDEPS** is a pass-through PostScript object. **PDEPS** methods include, for example:

- **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, for example:

- **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, for example:

- **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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDETextAdd</td>
<td>Adds a character or text run to a PDEText object.</td>
</tr>
<tr>
<td>PDETextCreate</td>
<td>Creates an empty text object.</td>
</tr>
<tr>
<td>PDETextGetBBox</td>
<td>Obtains the bounding box of a character or text run.</td>
</tr>
<tr>
<td>PDETextGetGState</td>
<td>Obtains the graphics state of a character or text run.</td>
</tr>
<tr>
<td>PDETextGetNumChars</td>
<td>Obtains the number of characters in a text object.</td>
</tr>
<tr>
<td>PDETextGetTextState</td>
<td>Obtains the text state of a character or text run.</td>
</tr>
<tr>
<td>PDETextRunGetCharOffset</td>
<td>Obtains the character offset of the first character of a text run.</td>
</tr>
<tr>
<td>PDETextRunSetFont</td>
<td>Sets the font of a text run.</td>
</tr>
<tr>
<td>PDETextRunSetTextState</td>
<td>Sets the text state of a text run.</td>
</tr>
<tr>
<td>PDETextSplitRunAt</td>
<td>Splits a text run into two text runs.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEXGroupAcquireColorSpace</td>
<td>Acquires the color space of the transparency group.</td>
</tr>
</tbody>
</table>
PDFEdit—Creating and Editing Page Content

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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDEXObjectCreate</code></td>
<td>Creates a new <code>PDEXObject</code> from a <code>Cos</code> object.</td>
</tr>
<tr>
<td><code>PDEXObjectGetCosObj</code></td>
<td>Gets a <code>Cos</code> object corresponding to a <code>PDEXObject</code>.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PDSysEncodingCreateFromBaseName</code></td>
<td>Create an encoding object from base name.</td>
</tr>
<tr>
<td><code>PDSysEncodingGetWMode</code></td>
<td>Returns the writing mode (0 for horizontal writing; 1 for vertical).</td>
</tr>
</tbody>
</table>

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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDEnumSysFonts</td>
<td>Enumerates all the system fonts.</td>
</tr>
<tr>
<td>PDFindSysFont</td>
<td>Finds the system font that matches given attributes.</td>
</tr>
<tr>
<td>PDSysFontGetAttrs</td>
<td>Obtains the attributes of a system font.</td>
</tr>
<tr>
<td>PDSysFontGetEncoding</td>
<td>Obtains encoding of single byte encoded system font.</td>
</tr>
<tr>
<td>PDSysFontGetName</td>
<td>Obtains the PostScript or TrueType styled name for a system font.</td>
</tr>
</tbody>
</table>
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 10.6, “Logical Structure,” 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 PDTextSelect object methods provided in the PD layer of the core API.
- Text on a page, using PDText object methods in the PDFEdit 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.
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
PDSEdit—Creating and Editing Logical Structure

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, for example:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDSTreeRootCreateClassMap</td>
<td>Creates a PDSClassMap in a structure tree root.</td>
</tr>
<tr>
<td>PDSTreeRootCreateRoleMap</td>
<td>Creates and sets a PDSRoleMap of a tree root.</td>
</tr>
<tr>
<td>PDSTreeRootGetKid</td>
<td>Gets the child at an array index in a structure tree root.</td>
</tr>
</tbody>
</table>

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, for example:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDSElementAddAttrObj</td>
<td>Associates an attribute object with an element at the element’s current revision value.</td>
</tr>
<tr>
<td>PDSElementGetClass</td>
<td>Gets the class name of an element.</td>
</tr>
</tbody>
</table>

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, suitiing 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, for example:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDSAttrObjCreate</td>
<td>Creates a new attribute object.</td>
</tr>
</tbody>
</table>
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. PDSMCs are identical to the PDFEdit class PDEContainer. PDSMCs may be nested. The PDSMC object has one method:

PDSMCGetInfo

 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, for example:

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

```c
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 PDFEdit 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 PDFEdit class *PDEContainer*. Be sure to cast objects...
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 does not 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 does not exist
- Adds a structure element to it

```c
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 ASUns8*)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.

```c
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.

```c
PDSElement XObjectElement;
PDSElementCreate(pdDoc, &XObjectElement);
#define TEXT_ELEMENT "A text element"
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 Section 10.7, “Tagged PDF” in the PDF Reference.
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*.

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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CosDocClose</td>
<td>Closes a Cos document.</td>
</tr>
<tr>
<td>CosDocCreate</td>
<td>Creates an empty Cos document.</td>
</tr>
<tr>
<td>CosDocGetRoot</td>
<td>Gets a document’s root Cos object, the catalog dictionary</td>
</tr>
<tr>
<td>CosDocOpenWithParams</td>
<td>Opens a Cos document.</td>
</tr>
<tr>
<td>CosDocSaveToFile</td>
<td>Sets a single element in an array.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CosObjCopy</td>
<td>Copies a CosObj from one document to another (or the same document).</td>
</tr>
<tr>
<td>CosObjGetDoc</td>
<td>Gets the CosDoc containing the object (indirect or non-scalar objects only).</td>
</tr>
<tr>
<td>CosObjGetType</td>
<td>Gets an object’s type.</td>
</tr>
<tr>
<td>CosObjIsIndirect</td>
<td>Tests whether an object is indirect. See Section 3.2.9 in the <em>PDF Reference</em> for details on indirect objects.</td>
</tr>
</tbody>
</table>

**CosObjCollection**

The **CosObjCollection** is an opaque structure representing a set of Cos objects associated with a particular Cos document. The initial value of a variable of type **ObjCollection** is undefined. Use **CosNewObjCollection** to create a collection.

Any indirect object whose generation number is zero and which is not a stream may be added to at most one **CosObjCollection**. When the file is saved, all the objects in a given collection are stored together in the PDF file, in one or more “object streams” (see the *PDF Reference*), which are normally compressed in order to reduce file size.
Collections allow grouping of objects that are likely to have a similar usage pattern. If one of them is required (and therefore decompressed), they are likely to all be required, but it is possible that none will be required. For example, there could be a collection that stores bookmarks. If the user never opens the bookmarks panel in Acrobat, none of the objects need to be decoded. If the user opens the bookmarks panel, it is likely they will all be needed, and having them together in the PDF file reduces the amount of time needed to load them all.

The **CosObjCollection** methods allow you to compare collections and iterate over the objects in a collection.

---

### 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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CosArrayGet</strong></td>
<td>Gets a single element from an array.</td>
</tr>
<tr>
<td><strong>CosArrayInsert</strong></td>
<td>Inserts an element into an array.</td>
</tr>
<tr>
<td><strong>CosArrayPut</strong></td>
<td>Sets a single element in an array.</td>
</tr>
<tr>
<td><strong>CosNewArray</strong></td>
<td>Creates an array.</td>
</tr>
</tbody>
</table>

---

### CosBoolean

Cos boolean objects can have a value of **true** or **false**. The **CosBoolean** methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CosBooleanValue</strong></td>
<td>Gets the value of the specified boolean object.</td>
</tr>
<tr>
<td><strong>CosNewBoolean</strong></td>
<td>Creates a new boolean object associated with the specified document and having the specified value.</td>
</tr>
</tbody>
</table>

---

### 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CosDictGet</strong></td>
<td>Gets the value of a dictionary key.</td>
</tr>
<tr>
<td><strong>CosDictPut</strong></td>
<td>Sets the value of a dictionary key.</td>
</tr>
<tr>
<td><strong>CosDictRemove</strong></td>
<td>Removes a key-value pair from a dictionary.</td>
</tr>
<tr>
<td><strong>CosNewDict</strong></td>
<td>Creates a dictionary.</td>
</tr>
</tbody>
</table>

**CosFixed**

Fixed numbers may only be in decimal format. See Section 3.2.2 in the *PDF Reference*, for details. The **CosFixed** methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CosFixedValue</strong></td>
<td>Gets the value of the specified fixed number object.</td>
</tr>
<tr>
<td><strong>CosNewFixed</strong></td>
<td>Creates a new fixed number object associated with the document, having the specified value.</td>
</tr>
</tbody>
</table>

**CosInteger**

Integers may be specified by signed or unsigned constants. See Section 3.2.2 in the *PDF Reference*, for details. The **CosInteger** methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CosIntegerValue</strong></td>
<td>Gets the integer value of the specified number object.</td>
</tr>
<tr>
<td><strong>CosNewInteger</strong></td>
<td>Creates a new integer object associated with the document, having the specified value.</td>
</tr>
</tbody>
</table>

**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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CosNameValue</strong></td>
<td>Gets the value of the specified name object.</td>
</tr>
</tbody>
</table>
### 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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CosNewStream</code></td>
<td>Creates a new Cos stream, using data from an existing ASStm.</td>
</tr>
<tr>
<td><code>CosStreamLength</code></td>
<td>Gets a stream’s length.</td>
</tr>
<tr>
<td><code>CosStreamOpenStm</code></td>
<td>Creates a new, non-seekable ASStm for reading data from a Cos stream.</td>
</tr>
<tr>
<td><code>CosStreamPos</code></td>
<td>Gets the byte offset of the start of a Cos stream’s data in the PDF file.</td>
</tr>
</tbody>
</table>

### CosString

A string is a sequence of characters, enclosed in parentheses. See Section 3.2.3 in the *PDF Reference* for details. `CosString` methods include, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CosNewString</code></td>
<td>Creates and returns a new Cos string object.</td>
</tr>
<tr>
<td><code>CosStringValue</code></td>
<td>Gets the value of Cos string object, and the string’s length.</td>
</tr>
</tbody>
</table>
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, for example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CosDecryptData</code></td>
<td>Decrypts data in a buffer using the specified encryption key.</td>
</tr>
<tr>
<td><code>CosEncryptData</code></td>
<td>Encrypts data in a buffer using the specified encryption key.</td>
</tr>
</tbody>
</table>

**NOTE:** These methods are not available in the Adobe PDF Library.
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 destroy 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 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 49 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:

```c
static AVCommandHandlerRec gAVCmdHandler;
const char *kCmdName = "MinimalCommand";
static ACCB1 AVCommandStatus ACCB2 DoWorkImpl (AVCommand cmd)
{
    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 70.

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.

```c
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,

```c
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.

```c
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, ( 
        AHostEncoding)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, 
        (AHostEncoding)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, 
        (AHostEncoding)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 Adobe Reader) for converting:

- To PDF from another file format (import)
Handlers

File Specification Handlers

- 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).

File Specification Handlers

A file specification handler converts between a PDFileSpec 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 PDFileSpec. It is called by PDFileSpecNewFromASPath.
- Convert a PDFileSpec 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.
Handlers

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`).
Handlers

Window Handlers

- Return the tool’s name (`GetTypeIdProcType`).
- 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`).
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).

For a complete list of callbacks in a window handler, see the description of AVWindowHandler in the Acrobat Core API Reference.
• 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 (**PMSendTimeDurationProc**).

• Set the current value of the progress monitor and update the display (**PMSendTimeCurrentValueProc**).

• Get the progress monitor’s maximum value (**PMGetDurationProc**).

• Get the progress monitor’s current value (**PMGetCurrentValueProc**).

For details, see the description of **ASProgressMonitorRec** in the *Acrobat Core API Reference*. 

**Acrobat Core API Overview**
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.

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 179.
- 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 181.
- 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 181.

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:** In 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 181.

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 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 89. 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 `AVDocOpenFromASFileWithParams`. 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 180.)

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 180.

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

**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 `PDPPermRegOprModify` (available with Acrobat 5.0 and higher) or either `pdPermEdit` or `pdPermEditNotes` permission.
- In 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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AVCryptGetPassword</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>AVCryptDoStdSecurity</strong></td>
<td>Displays a security dialog to the user, allowing the user to change the document’s permissions.</td>
</tr>
</tbody>
</table>
Handling Errors

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**

<table>
<thead>
<tr>
<th>ERROR CODE</th>
<th>Defined in</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrGetSeverity</td>
<td>AcroErr.h</td>
<td>Gets the severity of the error where severity is one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ErrNoError</strong> - No error</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ErrWarning</strong> - Warning</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ErrSilent</strong> - Do not display a message</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ErrSuppressable</strong> - Display a message if the user has not suppressed errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ErrAlways</strong> - Always display a message, even if others are suppressed</td>
</tr>
<tr>
<td>ErrGetSystem</td>
<td>AcroErr.h</td>
<td>Gets the system that raised the exception, where system is one of the values listed in Table 12.2.</td>
</tr>
<tr>
<td>ErrGetCode</td>
<td>AcroErr.h</td>
<td>Gets the error number. Acrobat's built-in exceptions are defined in AcroErr.h. Use ErrGetSignedCode if the platform considers error codes to be signed.</td>
</tr>
<tr>
<td>ErrGetSignedCode</td>
<td>AcroErr.h</td>
<td></td>
</tr>
<tr>
<td>ErrBuildCode</td>
<td>AcroErr.h</td>
<td>Builds an error code, given the severity, system, and error number</td>
</tr>
<tr>
<td>RERAISE</td>
<td>CorCalls.h</td>
<td>Re-raises the most recently raised exception and passes it to the next exception handler in the stack.</td>
</tr>
</tbody>
</table>

**Table 12.2 Exception system names**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrSysNone</td>
<td>General error and out of memory error</td>
</tr>
<tr>
<td>ErrSysCos</td>
<td>CosStore filters</td>
</tr>
<tr>
<td>ErrSysCosSyntax</td>
<td>Cos syntax errors</td>
</tr>
<tr>
<td>ErrSysPDDoc</td>
<td>PDDoc and family, Page tree, outlines</td>
</tr>
<tr>
<td>ErrSysPDPage</td>
<td>PDPage and family, thumbs, annots</td>
</tr>
</tbody>
</table>
The following code example illustrates an exception handler that simply determines which system raised an exception and displays that information in a dialog box:

```c
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 do not 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:

```c
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, do not 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.
Handling Errors

Exception Handler Caveats

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

Do Not 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:
```

Do Not Nest Exception Handlers In a Single Function

In general, do not 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:

```c
DURING ...
    MyFunction();
    ...
HANDLER ...
    ...
END_HANDLER ...
    void MyFunction(void) {
        ...
        DURING ...
            HANDLER ...
            END_HANDLER ...
        }
```

If you insist on nesting exception handlers in a single function, do not 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
    AVAlertNote("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 \texttt{volatile} variables whose value is needed in the exception handler or beyond.

When using \texttt{volatile}, be sure to place the keyword in the correct location, for example:

\begin{verbatim}
volatile myStruct* p = 0;
\end{verbatim}

declares the instance of the structure to be volatile, while

\begin{verbatim}
myStruct* volatile p = 0;
\end{verbatim}

declares the pointer itself to be volatile. In general, the second form is the one to use.
Handling Errors

Exception Handler Caveats
Changes For This Revision

Changes in This Document

- Revised Acrobat Software Development Kit Documentation Roadmap
- Updated references to the PDF Reference. They all refer to the correct sections in the PDF Reference, fourth edition, version 1.5.
- Updated to introduce the following major new features that were added to the Core API in Acrobat version 6.0:
  - Programmatic time and date manipulation (ASDate, ASTimeSpan)
  - Improved platform-independent path and file addressing (ASPlatformPath)
  - Programmatic control of undo/redo functionality (AVUndo)
  - Cos object collections for improved efficiency (CosObjCollection)
  - Creation and manipulation of optional content (layers) in a PDF (PDOCConfig, PDOCContext, PDOCG, PDOCMD)

Compatibility with API Changes in Acrobat 6

There are roughly 400 new APIs in Acrobat 6, and many existing APIs have been modified. The total number of APIs is now approximately 2000. Some APIs—including new and existing—will only work with certain configurations of Acrobat 6 and not with others. In addition, the Tools menu is now aligned with actual tools and toolbar configurations; as a result, many of the capabilities that used to be available under the Tools menu are now available under the new Advanced menu. It is not recommended that third parties use the Tools menu for placement of items; use the new Advanced menu instead.

This section provides an overview of the changes and their implications for existing plug-ins. For complete details, see Upgrading Plug-ins from Acrobat 5.0 to Acrobat 6.0.

New API Categories

New APIs in Acrobat 6 fall into the following general categories (see the last section of the Acrobat Core API Reference, “API Changes,” for details):

- Compound documents — Acrobat 6 supports the importing of many different file formats. Acrobat 6 can serve as a single repository/display mechanism for a rich variety of multimedia types, graphic formats, and fonts. See the PDFBinder plug-in sample.
- Layers/optional content — Acrobat 6 brings layered content and conditionalized display of information to Acrobat for the first time. The new PDOCG object and its
Changes For This Revision
Compatability with API Changes in Acrobat 6

Associated APIs are used to implement this new optional-content grouping capability. Examples of how the OC API can be used are in the SnippetRunner sample. A description of the ProcSet used to build OC into PDF through Distiller is in Acrobat Distiller API Reference.

- New text and font handling capabilities — Acrobat’s text manipulation API has been enhanced with the new PDETextItem API. Text manipulation has also been enhanced with expanded support for the ASText object (most character strings are now handled as ASText objects rather than as C-style character strings). Support for Unicode has been expanded.

- Digital signatures — A new public security API has been added to enable third-parties to more effectively use the Acrobat authentication capability. This new API, PubSec, is documented in the Digital Signature API Reference. This API is actually a “layer” above DigSig and relies on it. The DigSig API can still be used if required. Support for Microsoft’s CryptoAPI (MSCAPI) on Windows provides tight integration with Microsoft’s security API. See the DocSign sample plug-in.

- Security/encryption — Support for document security has been enhanced from the API level, mostly through the addition of more cryptographic filters. Certified documents have their own security model.

- Review and commenting — Support for annotations has been extended.

- Text extraction and manipulation (new Wordy algorithm) — More powerful text search and manipulation capabilities have been added to the API with enhancements to WordFinder. See the TextExtraction plug-in sample.

- Color management — The Adobe AcroColor API, while present in earlier versions of Acrobat, has been “exposed” in Acrobat 6. This new API is documented in a new document, AcroColor API Reference.

- New printing APIs — Support for separations and an accompanying API has been added to Acrobat 6.

- Object compression API — The PDF file format has been modified to enable powerful compression techniques to be applied to PDF files, resulting in much smaller file sizes. Access to these capabilities are provided through the API. The new CosObjCollection API provides support for this compression.

- Document level undo/redo stack — The API has been expanded to support multiple undo/redo. See the Stamper sample.

- New metadata calls — Support for Acrobat’s metadata API has been expanded.

- Date and time API — Acrobat now provides a cross-platform date and time API.

- UUID API — Support has been included for Universal Unique IDentifiers.

- New enterprise installer capabilities — Enterprise installation of Acrobat 6 has been made more flexible, including the ability to allow third parties to have their plug-ins automatically installed through the Acrobat updater.
● Save PDF files as XML — A new plug-in enables users to save any PDF file as an XML file. You must, however, provide your own DTD, which will be selectable from Save As.... See Using the Save as XML Plug-in and the samples for details.

● Help system enhancements— Plug-in developers can now write help systems to support their plug-ins directly from within Acrobat using the HowTo window. Support for PDF-based help has been expanded. See Acrobat Help for details.

Data Type Changes

There have been extensive changes to the data types of existing APIs. Some of these changes are to the fundamental data types, and some are new abstract types that clarify the meaning and use of parameters.

16-to-32 Bit Changes

The most profound changes to the existing API is the migration of most 16-bit integer data types to 32-bits for use with Acrobat 6. This includes, for example, the changing of many ASInt16s to AVDevCoord and other variables that are defined as 16- or 32-bit, depending on the level of the SDK:

```
#if !defined(ACRO_SDK_LEVEL) || (ACRO_SDK_LEVEL < 0x00060000)
typedef ASInt16 AVSDKDependentInteger;
#else
typedef ASInt32 AVSDKDependentInteger;
```

AVSDKDependentInteger is then used to set the following variables, which have been “retrofitted” into the appropriate existing APIs, providing the compatibility layer.

AVDevSize — Denotes a size in the page view’s device space.

AVDevCoord — Contains an x or y coordinate in the page view’s device space.

AVWindowCoord — Contains an x or y coordinate in the windows space, where (0, 0) is at the top left and units are in pixels.

AVScreenCoord — Contains an x or y coordinate in the screen space, where (0, 0) is at the top left of the main monitor.

ASText Object Changes

Most pointers to character strings have been replaced with the ASText object.

New Abstract Types

New abstract types have been retrofitted into the API to clarify the meaning and use of parameters. These are listed separately in the “API Changes” for version 6 section of the Acrobat Core API Reference. These types are referred to as “cosmetic” types since they can be compiled without error if the generic types are still used.
Support for Backward Compatibility

To support the changes to the data types in the API, mechanisms have been provided to allow backward compatibility.

ACRO_SDK_LEVEL Macro

The ACRO_SDK_LEVEL macro is set to the version of the API with which you want to compile. This macro is primarily used in two new header files, AVExpTObsolete1.h and AVExpTObsolete2.h. These files contain types used in former versions of Acrobat. By defining ACRO_SDK_LEVEL to 0x00060000, you use the latest types and routines while retaining the ability to compile existing plug-in code without changes.

AVCompat.cpp File

The AVCompat.cpp file is also used to provide backward compatibility by supporting the passing of current structures and callbacks to older versions of Acrobat. AVCompat.cpp is used to provide backward compatibility by converting new structures, callbacks, and calls into old structures, callbacks, and calls and converting the results back to the new structures.

NOTE: If you do not need to run on Acrobat 5 or earlier, there is no reason to add AVCompat.cpp to your plug-in.

Testing With the TEST_OLD_VERSION Macro

In PIMain.c, the macro TEST_OLD_VERSION can be set to test a plug-in's compatibility with earlier versions of Acrobat. It is set to 0xFFFFFFFF by default, which is the value required for shipping Acrobat 6 plug-ins. However, the macro can be set to previous versions of Acrobat to simulate those execution environments. For example, it can be set to 0x00050000 to simulate Acrobat 5.0.

NOTE: While the simulation approximates earlier Acrobat execution environments, it is not the same as the real thing. A plug-in should still be tested against the actual products before shipping.
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>
<thead>
<tr>
<th>PDF Version</th>
<th>Acrobat Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>1.1</td>
<td>2.1</td>
</tr>
<tr>
<td>1.2</td>
<td>3.0</td>
</tr>
<tr>
<td>1.3</td>
<td>4.0</td>
</tr>
<tr>
<td>1.4</td>
<td>5.0</td>
</tr>
<tr>
<td>1.5</td>
<td>6.0</td>
</tr>
</tbody>
</table>

**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, or 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**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Page Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Objects: Booleans, Numbers, Strings, Names, Arrays, Dictionaries, Streams, Filters</td>
<td>(PDF Operators that describe text, graphics, and images)</td>
</tr>
<tr>
<td>File Structure</td>
<td></td>
</tr>
<tr>
<td>(Header, Body, Cross-reference Tables, Trailer)</td>
<td></td>
</tr>
<tr>
<td>Document Structure</td>
<td></td>
</tr>
<tr>
<td>(Catalog, Pages Tree, Pages, Imagable Content, Thumbnail, Annotation, Outline Tree, etc.)</td>
<td></td>
</tr>
</tbody>
</table>

**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.
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.
**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.
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**

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path</td>
<td>An arbitrary shape made of straight lines, rectangles, and cubic curves.</td>
</tr>
<tr>
<td>Text</td>
<td>One or more character strings that can be placed anywhere on the page and in any orientation.</td>
</tr>
<tr>
<td>Image</td>
<td>A set of samples using a specified color model.</td>
</tr>
<tr>
<td>XObject</td>
<td>A PDF object referenced by name. The three types of XObjects are:</td>
</tr>
<tr>
<td></td>
<td>● Image</td>
</tr>
<tr>
<td></td>
<td>● Form</td>
</tr>
<tr>
<td></td>
<td>● PostScript language form</td>
</tr>
</tbody>
</table>
Index

A
"about" box and splash screen 48
Acrobat Support (AS) layer 55
   ASAtom 55
   ASCab 55
   ASCallback 57
   ASDate 57
   ASExtension 57
   ASDate 57
   ASExtension 57
   ASFile 58
   ASFileSys 58
   ASPlatformPath 60
   ASSm 60
   ASText 60
   configuration 62
   errors 62
   fixed-point math 62
   HFT methods 64
   memory allocation 64
   platform-specific utilities 64
Acrobat Viewer (AV) layer 67
   AVActionHandler 68
   AVAlert 68
   AVAnnotHandler 69
   AVApp 69
   AVCommand 70
   AVConversion 72
   AVCrypt 73
   AVDoc 73
   AVGrafsSelect 74
   AVMenu 74
   AVMenuBar 75
   AVNenuitem 75
   AVPageView 76
   AVSweetPea 77
   AVSys 78
   AVTool 78
   AVToolBar 78
   AVToolButton 79
   AVUndo 80
   AVWindow 80
Acrobat viewer’s user interface 48
Acrobat viewers, controlling 51
   action handlers 164
   adding new object types 33
   adjusting the cursor 46
   Adobe Dialog Manager (ADM) 77
   annotation handlers 164
   annotation types, new 54
   Apple events 47
   applications, examples 51
   AS layer 23
   AV layer 22
   AVCommand 165

B
batch processing 166

C
cabinets 55
callbacks 42
classes, PDFEdit 111
command handlers 165
complex types 28
controlling the Acrobat viewers 51
conversion handlers 167
converting file formats to and from PDF 72
coordinate systems 28, 49
   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
Index

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 158
CosBoolean 158
CosCollection 157
CosDict 158
CosDoc 157
CosFixed 159
CosInteger 159
CosName 159
CosNull 160
CosObj 157
CosStream 160
CosString 160
encryption/decryption 161
file structure 156
Cos objects 28
creating HFTs 37
cross-platform dialog management 77

data types
  complex 28
  Cos 28
  opaque 28
  scalar 26
  simple 27
dates and times 57
DDE messages 47
DEBUG 42
decryption 175
device space 29, 30
device space, coordinates of 29
dictionary access 54
document security 53, 175
  utility methods 183
drawing 49
drawing into another window 51

e
encoded text, specifying 60
encryption 175
encryption/decryption
  Cos layer 161
enumeration 45
error handling 185, 195
errors, Acrobat Support (AS) layer methods 62
event handling 46
examples of applications and plug-ins 51
exception handlers 185
exception handling 32
exporting HFTs 40
extracting text 53

f
file specification handlers 168
file systems 58
fixed-point math 62

h
handlers 163
  action 164
  annotation handlers 164
  command 165
  conversion 167
  file specification 168
  file systems 171
  progress monitors 172
  security 168, 176
  selection servers 168
  tools 169
  transition 173
  window 170
handling errors 185, 195
  adding new exceptions 188
  passing an exception 188
handling events 46
  adjust cursor 46
key presses 46
mouse clicks 46
handling exceptions 32
handshaking 39
handshaking with plug-ins 39
help files 49
HFT 35, 64
creating 37
exporting 40
importing 40
replacing entries in 37
using 36
HFT servers 36
Host encoding 61
host function table (HFT) 35, 64

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) 47
interapplication communication (IAC) 21
invoking AVCommands programmatically 70

K
key presses 46

L
language codes, setting 62
layers
Acrobat Support (AS) 55
Acrobat Viewer (AV) 67
Cos 155
Portable Document (PD) 83
logical structure 143

M
machine port space 28, 31
Macintosh utilities 64
mechanics of the core API 35
memory allocation 64
menus and menu items 48
message handling, adding 47
Metadata 84
method names, general format of 24
methods, replacing 37
mouse click processing 49
mouse clicks 46

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
optional content 98
organization of core API 22

P
page view layers 49
passing an exception 188
path names 60
PD layer 22
PDF files
private data in 33, 54
PDF format, converting to and from 72
PDFEdit 22, 109
classes 111
class comparison with other core APIs 115
debbuging tools and techniques 127
dump methods 129
general methods 130
hit testing 117
matrix operations 119
page creation 121
PDEClip 130
PDEColorSpace 130
PDEContainer 131
PDEContent 132
PDEDeviceNColors 133
PDEElement 133
PDEExtGState 134
PDEFont 135
PDEForm 136
PDEGroup 136
PDEImage 136
PDEObject 137
PDEPath 137
PDEPattern 138
PDEPlace 138
PDESHading 139
PDEText 139
PDEUnknown 140
PDEXObject 141
PDSysFont 141
PDFFileSpec 91
PDModel layer 83
PDSEdit 23, 143
PDSAttrObj 146
PDSClassMap 147
PDEElement 146
PDSMC 147
PDSObj 147, 153
PDSRoleMap 147
PDSTreeRoot 146
permissions, querying PDDoc 89
platform-specific utilities 64
plug-in initialization 39
plug-in naming conventions 47
plug-in prefixes 47
plug-ins 21
examples 51
initialization 41
reducing conflicts among 50
unloading 42
plug-ins, Adobe Reader-enabled 47
Portable Document (PD) layer 83
general methods 84
PDAction 86
PDAnnot 86
PDBead 87
PDBookmark 87
PDCharProc 88
PDDoc 88
PDFFileSpec 91
PDFForm 94
PDGraphic 94
PDImage 95
PDInlineImage 95
PDLinkAnnot 96
PDNameTree 96
PDNumTree 97
PDOCConfig 97
PDOCContext 97
PDOCG 98
PDOCMD 99
PDPage 100
PDPageLabel 101
PDPath 101
PStyle 102
PDText 102
PDTextAnnot 102
PDTextSelect 103
PDThread 104
PDTThumb 104
PDTTrans 105
PDViewDestination 105
PDWord 105
PDWordFinder 106
PDXObject 107
private data in PDF files 33, 54
progress monitors 172
Q
quadrilaterals 32
querying PDDoc permissions 89
R
Reader-enabled plug-ins 47
rectangles 32
replacing entries in HFTs 37
replacing methods 37

S
scalar types 26
searching, indexed 52
security 175
  document 53
  new features in Acrobat 5.0 178
security handlers 168, 176
  callbacks 178
selection servers 168
simple types 27
splash screen 48
streams 60
structure, logical 143

T
tagged PDF 143
text, extraction 53
times and dates 57
toolbar 48
transition handlers 173
translating between user and device space 30
transparency 134
types, core API 26

U
Unicode 60
UNIX utilities 65
unloading plug-ins 42
user interface 48
  "about" box and splash screen 48
  help files 49
  menus and menu items 48
  toolbar 48
user space 28, 30

user space, coordinates of 28

W
window, drawing into another 51
Windows utilities 65