## Contents

### Preface
- Description .................................................. 11
- Audience ..................................................... 11
- Related Documents .......................................... 11
- How This Document is Organized ........................... 12
- Conventions Used in This Book .............................. 13

### Chapter 1  Getting Started .................................. 15
- What Is a Plug-in? ............................................. 15
- Certified Plug-ins ........................................... 16
- Adobe Reader-enabled Plug-ins ............................ 16
- SDK Platforms ............................................... 16
- Acrobat 7 Configurations .................................... 17
- New API Overview ............................................ 18
  - New APIs and Their Implications for Existing Plug-ins 18
- Understanding Your Target Application .................... 18
  - Rights-enabled Documents in Adobe Reader ............ 19
- Which APIs Work With Which Products? .................. 20

### Chapter 2  Plug-in Overview ................................. 23
- Host Function Tables ......................................... 23
  - Using HFTs ................................................ 24
  - HFT Servers ............................................... 24
  - Creating a New HFT ....................................... 25
  - Rules to Follow ............................................ 25
- Interaction Between Plug-ins and Acrobat or Adobe Reader 26
  - Plug-in Loading and Initialization ....................... 26
  - Handshaking and Initialization .......................... 27
  - Exporting HFTs ............................................ 28
  - Importing HFTs and Registering for Notifications .... 28
  - Initialization .............................................. 29
  - Unloading .................................................. 29
  - Callbacks .................................................. 29
Contents

Notifications ................................................. 30
Handling Events ........................................... 31
Mouse Clicks ................................................. 31
Adjust Cursor ................................................ 31
Key Presses .................................................. 32
Plug-in Prefixes ............................................. 32
Changing the Acrobat or Adobe Reader User Interface ........................................... 32
  Adding or Removing Menus and Menu Items ........................................... 32
  Modifying the Toolbar .................................... 33
  Controlling the “About” Box and Splash Screen .................................... 33
  Placing Plug-in Help Files In a Standard Location .................................. 33
  User Interface Guidelines ................................ 33
Acquiring and Releasing Objects ............................................. 34
Memory Allocation ........................................... 34
Using Modal and Modeless Dialogs ........................................... 34
PDF File Integrity ............................................ 34
Security ...................................................... 35
Debugging ..................................................... 35
Page View Layers ............................................ 36
Drawing ....................................................... 37
Reducing Conflicts Among Plug-ins ........................................... 37
  Naming ....................................................... 37

Chapter 3  Basic Plug-in Components ......................... 39

Overview ...................................................... 39
A Plug-in's Life Cycle ....................................... 40
StarterInit.cpp: A Closer Look ................................ 40
  Callbacks ................................................... 40
  PIHandshake .............................................. 42
Defining Callback Functions .................................... 42
  Using ASCallbackCreateProto and Other Macros .................................. 42
  Defining Callback Types ................................... 43
  Providing An Extension Name ................................ 44
PIMain.c: A Closer Look .................................... 44
  PluginMain ................................................ 44
  PISetupSDK ................................................ 45

Chapter 4  Menus and Menu Items ............................ 47
## Chapter 5 Exception Handling ................................. 55
- Exception Handling Overview .................................. 55
- Exception Handling Macros ..................................... 56
- Returning From a DURING...HANDLER Block ............. 56
- Using ASRaise ......................................................... 57
- ShowException: An Error Display Method .................. 57
- General Rules to Follow ......................................... 58

## Chapter 6 Toolbars and Buttons .............................. 61
- Toolbar And Buttons ............................................... 61
- ASAtoms ................................................................. 62
- Adding A Button To the Toolbar ............................... 62
- Error Handling in the Example ................................. 62
- The Complete Example .......................................... 63

## Chapter 7 Documents and Files ............................... 65
- Documents ............................................................ 65
- Opening a PDF File For Viewing .............................. 66
  - Creating a Platform-Independent Path Name ............ 66
  - Accessing AVDoc Objects Already Opened ............. 67
- Bridging Core API Layers ...................................... 67
- Opening a PDDoc Object ......................................... 68
- Accessing Non-PDF Files ....................................... 69

## Chapter 8 Page Views and Contents .......................... 73
- Page Views ........................................................... 73
  - Displaying a Page in a Document .......................... 74
  - Coordinate Systems ............................................. 75
Contents

PDF Pages ................................................................. 76
Accessing Page Contents with PDFEdit .......................... 76
Accessing PDE Objects .................................................. 76
Initialization .............................................................. 77
Accessing a Page's Content ............................................. 77
The Complete Example .................................................. 79

Chapter 9  Finding Words in a Document ......................... 81
Finding Words ............................................................ 81
PDWord ................................................................. 81
PDWordFinder ............................................................ 82
Example of Finding and Displaying Words ....................... 82
Initialization .............................................................. 82
Creating the PDWordFinder ........................................... 82
Enumerating the Words ................................................ 83
The Complete Example .................................................. 84

Chapter 10  Annotations .................................................. 87
About Annotations ....................................................... 87
Finding Annotations ..................................................... 87
Iterating Through the Pages ......................................... 88
Finding the Annotations on a Page. ............................... 88
Acting on the Annotation ............................................... 88
The Complete Example .................................................. 88
Getting and Setting the Contents of a Text Annotation ....... 90
Checking for Text Annotations ....................................... 90
Exception Handling ..................................................... 90
Getting and Setting the Annotation Text ........................ 90
The Complete Example .................................................. 91
Creating a New Text Annotation ..................................... 92

Chapter 11  Bookmarks .................................................... 95
About Bookmarks ......................................................... 95
Bookmark Organization ............................................... 95
Bookmark Methods ..................................................... 96
Finding Bookmarks ....................................................... 96
The Root Bookmark ..................................................... 96
Visiting All Bookmarks ................................................ 96

Acrobat SDK Plug-in Guide
Contents

Searching for A Bookmark by Title ........................................ 97
Creating New Bookmarks ......................................................... 99
PDF Objects in the Bookmark Tree ........................................... 100

Chapter 12  Replacement, Notifications, and Event Registration .............. 103
Replacing Built-In Methods in HFTs ....................................... 103
Replacing AVAppCanQuit ......................................................... 104
Notifications ............................................................... 106
Registering For a Notification .............................................. 106
Registering For Events ............................................................. 107
Registering For AVPageView Click Events ................................. 108

Chapter 13  Exporting and Importing Host Function Tables ..................... 109
Exporting an HFT ................................................................. 109
The Exported Methods ......................................................... 109
Accessing the Methods ........................................................... 110
HFT Method Definitions .......................................................... 110
The Callbacks ................................................................. 112
The Header File ................................................................. 114
The Source File ................................................................. 115
Importing an HFT ................................................................. 118

Chapter 14  Cos Objects and Methods ......................................... 119
Cos Objects in PDF Files ......................................................... 120
Direct and Indirect Objects ...................................................... 120
Cos Object Types ................................................................. 121
Strings ................................................................. 121
Arrays ............................................................... 121
Names ................................................................. 121
Dictionaries ................................................................. 122
Streams ................................................................. 122
Strings ................................................................. 123
Creating a New String ............................................................ 123
Getting the String From the Cos Object ...................................... 123
Arrays ................................................................. 123
Creating a Simple Direct Array ................................................... 123
Retrieving Array Values .......................................................... 124
**Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionaries</td>
<td>125</td>
</tr>
<tr>
<td>Creating a Simple Direct Dictionary</td>
<td>125</td>
</tr>
<tr>
<td>Names</td>
<td>126</td>
</tr>
<tr>
<td>Creating a New Name Value</td>
<td>126</td>
</tr>
<tr>
<td>Getting an Annotation's Subtype</td>
<td>127</td>
</tr>
<tr>
<td>Streams</td>
<td>127</td>
</tr>
<tr>
<td>An Unencoded Page Content Stream</td>
<td>127</td>
</tr>
<tr>
<td>Creating a Stream Using a Memory Buffer</td>
<td>128</td>
</tr>
<tr>
<td>Creating a Stream From an Existing Data File</td>
<td>133</td>
</tr>
<tr>
<td>Creating a New Document, New Page, and Page Content Stream</td>
<td>135</td>
</tr>
<tr>
<td>The Complete Example</td>
<td>138</td>
</tr>
<tr>
<td>Reading a Stream's Data</td>
<td>146</td>
</tr>
<tr>
<td>Page Content Streams</td>
<td>147</td>
</tr>
</tbody>
</table>

**Chapter 15 Other Plug-in Topics.** ........................................ 149

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding Message Handling</td>
<td>149</td>
</tr>
<tr>
<td>Creating Thumbnail Images</td>
<td>149</td>
</tr>
</tbody>
</table>

**Appendix A Upgrading Plug-ins From Acrobat 6 to Acrobat 7.** ........... 151

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Information</td>
<td>151</td>
</tr>
<tr>
<td>Backward Compatibility</td>
<td>151</td>
</tr>
<tr>
<td>Development Header Files Changes for Acrobat 7</td>
<td>152</td>
</tr>
<tr>
<td>Detecting Supported APIs on Acrobat Professional, Acrobat Standard, and Adobe Reader</td>
<td>153</td>
</tr>
<tr>
<td>Updating the UI</td>
<td>154</td>
</tr>
<tr>
<td>Migrating to XML Forms</td>
<td>154</td>
</tr>
<tr>
<td>Adobe Dialog Manager in Acrobat 7</td>
<td>155</td>
</tr>
<tr>
<td>Updates to ASCoord and AVRect</td>
<td>157</td>
</tr>
<tr>
<td>Upgrading Plug-ins on Windows</td>
<td>158</td>
</tr>
<tr>
<td>Reader-enabling Windows Plug-ins</td>
<td>158</td>
</tr>
<tr>
<td>Upgrading Plug-ins on a Macintosh</td>
<td>159</td>
</tr>
<tr>
<td>Target settings</td>
<td>159</td>
</tr>
<tr>
<td>Project Settings</td>
<td>160</td>
</tr>
<tr>
<td>Reader-enabling Macintosh Plug-ins</td>
<td>160</td>
</tr>
</tbody>
</table>

**Appendix B Registering and Using Plug-in Prefixes.** .................... 161

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtaining a Developer Prefix</td>
<td>161</td>
</tr>
<tr>
<td>Using the Developer Prefix</td>
<td>161</td>
</tr>
<tr>
<td>Plug-in Name</td>
<td>161</td>
</tr>
</tbody>
</table>
## Appendix C  Platform-Specific Techniques  ........................................ 165

- Platform-dependent Data in the API .................................................................. 165
- Portability Techniques ......................................................................................... 166
  - Macintosh Techniques ..................................................................................... 166
  - Developing a Macintosh Mach-O Plug-in ......................................................... 166
  - Locating and Loading Plug-ins ....................................................................... 168
  - Memory ............................................................................................................ 169
  - Resource File Considerations ........................................................................ 169
  - Macros ............................................................................................................ 170
  - Macintosh-Only Methods ............................................................................... 170
  - Interapplication Communication ................................................................. 170
  - Debugging ....................................................................................................... 171
- Windows Techniques .......................................................................................... 171
  - Developing a Windows Plug-in ....................................................................... 171
  - Plug-in Implementation ................................................................................... 171
  - Locating and Loading Plug-Ins ...................................................................... 171
  - Why a Plug-in Might Not Load ....................................................................... 172
  - Macros and Project Settings .......................................................................... 173
  - Interapplication Communication ................................................................... 173
  - Debugging ....................................................................................................... 174
  - Handling the Thread Local Storage (TLS) Limit ........................................... 174
  - Using Modal Dialogs ....................................................................................... 175

## Appendix D  Plug-in Header Files ................................................................. 177

- Acrobat SDK Header Files ................................................................................. 177
- API Header Files .............................................................................................. 178
- ADM Header Files ........................................................................................... 183
Preface

Description

This guide describes basic Acrobat® plug-in development concepts and shows how Acrobat interacts with plug-ins. It also shows how your plug-in can manipulate and enhance the Acrobat and Adobe Reader® user interface and the contents of underlying PDF documents.

This guide also describes how to upgrade plug-ins from Acrobat 6 to Acrobat 7, provides platform-specific techniques for developing plug-ins, and lists the Acrobat SDK header files.

**NOTE:** The following Acrobat 6.0 documents have been replaced by this guide in the Acrobat 7.0 documentation:
- Acrobat Development Overview
- Acrobat Plug-in Tutorial
- Upgrading Plug-ins From Acrobat 5 to Acrobat 6

Audience

The primary audience of this document is Acrobat and Adobe Reader plug-in developers. 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.

**NOTE:** While this document is aimed at plug-in developers, some of the content is relevant to developers writing applications using the Adobe PDF Library. For more information, see the *PDF Library Overview* (licensed separately).

Related Documents

For more information, see the following SDK documents:
- *Acrobat and PDF Library API Overview* contains the overview information on API objects as well as general information about using the APIs.
- *Acrobat and PDF Library 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 SDK User’s Guide* provides an overview of the Acrobat SDK and the supporting documentation.
Preface

How This Document is Organized

- *Developing for Adobe Reader* provides an introduction to those portions of the Adobe Acrobat Software Development Kit (SDK) that pertain to your development efforts for Adobe Reader.
- *Guide to SDK Samples* provides an overview of the samples provided with the Acrobat SDK.
- *SnippetRunner Cookbook* provides an overview of the plug-in snippets provided with the Acrobat SDK.
- *PDF Reference, fifth edition, Version 1.6* provides a description of the PDF file format, as well as suggestions for producing efficient PDF files. It is intended for application developers who wish to produce PDF files directly, or who wish to gain a detailed understanding of the PDF file format.

Additional documents that you should have available for reference are listed below.
- *Adobe Reader-enabled Plug-ins* describes the steps required to enable a plug-in to be loaded by Adobe Reader.
- *PostScript Language Reference, third edition* describes the syntax and semantics of the PostScript® language and the Adobe imaging model.

How This Document is Organized

This guide is organized as follows:

- **Chapter 1, “Getting Started,”** provides an introduction to plug-in development and discusses issues that you should consider before beginning to develop your plug-in.
- **Chapter 2, “Plug-in Overview,”** discusses a number of topics basic to plug-in development.
- **Chapter 3, “Basic Plug-in Components,”** through **Chapter 14, “Cos Objects and Methods,”** explain how to use the Acrobat Core API to write plug-ins for Acrobat and Adobe Reader. These chapters describe basic Acrobat development concepts, shows how Acrobat interacts with plug-ins, and shows how your plug-in can manipulate and enhance the Acrobat user interface as well as the contents of underlying PDF documents.
- **Chapter 15, “Other Plug-in Topics,”** provides general information and rules to follow on several plug-in development topics.
- **Appendix A, “Upgrading Plug-ins From Acrobat 6 to Acrobat 7,”** provides an overview of what is new in the Acrobat 7 API and discusses what developers need to know when upgrading plug-ins to Acrobat 7.
- **Appendix B, “Registering and Using Plug-in Prefixes,”** lists rules for naming certain parts of your plug-in, as well as data placed in PDF files.
- **Appendix C, “Platform-Specific Techniques,”** contains platform-specific development information for the Macintosh and Windows platforms and provides guidelines that can help make plug-ins more portable among the various supported platforms.

- **Appendix D, “Plug-in Header Files,”** provides descriptions of the Acrobat SDK header files.

## Conventions Used in This Book

The Acrobat and PDF Library 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>AMenu commandMenu, helpMenu;</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</td>
<td>The enumeration terminates if <code>proc</code> returns <code>false</code>.</td>
</tr>
<tr>
<td></td>
<td>reference documents</td>
<td></td>
</tr>
<tr>
<td>monospaced italic</td>
<td>Pseudocode</td>
<td><code>ACCB1 void ACCB2 ExeProc(void)</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 Adobe Solutions Network URL is: <a href="http://partners/adobe.com/asn/">http://partners/adobe.com/asn/</a></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 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,</td>
<td>The <code>setpagedevice</code> operator</td>
</tr>
<tr>
<td></td>
<td>keywords, dictionary key names</td>
<td></td>
</tr>
<tr>
<td></td>
<td>User interface names</td>
<td>The <code>File</code> menu</td>
</tr>
</tbody>
</table>
## Conventions Used in This Book

<table>
<thead>
<tr>
<th>Font</th>
<th>Used for</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>italic</td>
<td>Document titles that are not live links</td>
<td>Acrobat and PDF Library 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 <strong>deletefile</strong></td>
</tr>
</tbody>
</table>
Getting Started

As a plug-in developer, you can add functionality to Adobe Reader and Acrobat that your organization or other companies can use. An Acrobat plug-in is a program like any other, but uses the resources of Acrobat or Adobe Reader as a host environment. This means that a plug-in does not require complex user interface elements. However, it must perform certain basic functions to let the Adobe Reader or Acrobat know of its presence.

This chapter discusses issues that you should consider before beginning to develop your plug-in.

- What Is a Plug-in?
- Certified Plug-ins
- Adobe Reader-enabled Plug-ins
- SDK Platforms
- Acrobat 7 Configurations
- New API Overview
- Understanding Your Target Application
- Which APIs Work With Which Products?

What Is a Plug-in?

Plug-ins are dynamically-linked extensions to Acrobat. They can hook into the user interface in a number of ways and can register to be called when a variety of events occur in the application.

A plug-in is a program written in ANSI C/C++ and uses the Acrobat public API. It adds functionality to Acrobat Professional, Acrobat Standard, or Adobe Reader. A plug-in program file goes into a Plug_ins folder/directory and is initialized upon startup of Acrobat or Adobe Reader.

There are three types of plug-ins:

1. Regular plug-ins. The samples in the SDK are of this type.
2. Reader-enabled plug-ins. These are plug-ins that are developed with permission from Adobe and that require special processing to load under Adobe Reader.
3. Certified plug-ins. See “Certified Plug-ins”

Plug-ins are:

- DLLs on Windows. Note, however, that plug-in names must end in .API, not .DLL.
Certified Plug-ins

Certified plug-ins have undergone extensive testing to ensure that they do not compromise the integrity of Acrobat’s security model. There is currently no way for third-party plug-ins to be “certified” by Adobe. There's a checkbox in the product user interface that can be used to ensure that only certified plug-ins load. This means that other plug-ins won't load. Certified plug-ins are reserved for Adobe only—no third parties.

Adobe Reader-enabled Plug-ins

Adobe Reader only accepts Reader-enabled plug-ins. Adobe Reader-enabled plug-ins can only access a limited set of APIs. For more information, see “Understanding Your Target Application” on page 18.

For information on enabling plug-ins for Adobe Reader, see the document, Adobe Reader-enabled Plug-ins.

SDK Platforms

The Acrobat 7 SDK is supported on the following platforms:

<table>
<thead>
<tr>
<th>Windows</th>
<th>Macintosh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 2000 with SP2</td>
<td>Mac OS X v.10.2.8, 10.3</td>
</tr>
<tr>
<td>Windows XP Home or Professional Edition</td>
<td></td>
</tr>
<tr>
<td>Visual Studio 2003, including C++, VB.NET and C#</td>
<td>CodeWarrior 9.2</td>
</tr>
</tbody>
</table>

The SDK is also available for several UNIX systems. However, release of the UNIX SDK to correspond with a new release of Acrobat usually lags release of the Windows and Macintosh SDKs. See the Adobe Solutions Network Web site for availability.
Acrobat 7 Configurations

Adobe Acrobat 7.0 consists of a family of products for creating, modifying, indexing, searching, displaying, and manipulating PDF (Portable Document Format) files. For further information about the Acrobat family of products, see the Acrobat Main Page.

- Adobe Reader for viewing, navigating, and printing PDF documents. Adobe Reader is free software that lets anyone view and print Adobe PDF files on all major computer platforms, as well as fill in and submit Adobe PDF forms. Adobe has distributed more than 500 million copies of the software worldwide.

- Acrobat for adding navigational links, annotations, and security options, in addition to the functionality provided by Adobe Reader. Acrobat 7.0 is offered in two configurations:
  - Acrobat 7.0 Standard — Offers all of the Adobe PDF creation and distribution tools in Acrobat Elements® (see below) and also allows users to more effectively manage document review cycles. Acrobat Standard’s intuitive review and commenting tools help streamline collaboration and reduce the costs and inefficiencies of paper-based review processes.
  - Acrobat 7.0 Professional — Contains all of the functionality of Acrobat 7.0 Standard and Acrobat Elements (see below). In addition, it offers the most advanced control over document creation, exchange, review, and output. It is intended for users who need to perform sophisticated tasks, such as preflighting print files, distributing layered CAD drawings, and creating interactive forms.

- Acrobat Elements for organizations requiring widespread PDF creation capability without the need to install the full Acrobat/Distiller product on every desktop. Acrobat Elements allows organizations to extend their investments in Microsoft® Office® by standardizing on Adobe PDF for document distribution. Acrobat Elements is available through licensing with minimum orders of 1,000 seats. There is no public API for Acrobat Elements.

Not all APIs work for all configurations, but all APIs work for Acrobat Professional. The Acrobat and PDF Library API Overview provides a table that specifies which APIs work with which products. In general, all APIs work with Acrobat Professional, nearly all work with Acrobat Standard, and many work with Adobe Reader.

Under Adobe Reader, when a rights-enabled PDF file is opened, a flag is set that allows a plug-in to use APIs that become enabled as a result of loading the rights-enabled PDF.
New API Overview

There are over 100 new APIs in Acrobat 7, and many existing APIs have been modified. The total number of APIs available to plug-in developers is now approximately 2200. Some APIs—including new and existing—will only work with certain configurations of Acrobat 7 and not with others.

Acrobat 7 supports viewing PDF files within a browser on Mac OS X. This means that browser-based collaboration and other browser-based features previously available only on Windows are now available on Mac OS X as well.

New APIs and Their Implications for Existing Plug-ins

New APIs in Acrobat 7 fall into the following general categories (see the last section of the Acrobat and PDF Library API Reference, “API Changes,” for details):

- Support for 64-bit file sizes
- New graphics objects: PDEImageFlate, PDEImageJPX
- New Cos-level APIs used to create strings instead of using ASAtoms
- New APIs to support thread safety
- New APIs to support opening multiple windows on the same document
- New member in PDDocSaveParams (numSubFilesToCompact) for use with new auto-save feature
- New callback AVAnnotHandlerGetAccessibilityStringProc for accessible annotations

Understanding Your Target Application

Both Acrobat and Adobe Reader accept plug-ins. Adobe Reader is designed predominantly for viewing and printing PDF documents. Acrobat Professional provides the ability to create PDF files, and offers advanced control over document exchange, review and output. Acrobat Standard also provides the ability to create PDF files and exchange and review comments.

PDF documents that are rights-enabled using Adobe LiveCycle™ Reader Extensions are able to access specific functionality in Adobe Reader. For more information, see “Rights-enabled Documents in Adobe Reader” on page 19.

Adobe Reader only accepts Reader-enabled plug-ins. For information on enabling plug-ins for Adobe Reader, see the document, Adobe Reader-enabled Plug-ins.

You may want your Reader-enabled plug-in to access APIs that are available when the plug-in is running with Acrobat but not when running with Adobe Reader. Use the ASGetConfiguration method to check whether Acrobat or Adobe Reader is running,
and call these APIs only if your plug-in is running with Acrobat. 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 Acrobat or Adobe Reader” on page 26 for information on 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.

**Rights-enabled Documents in Adobe Reader**

PDF files can be enabled for additional functionality using Adobe LiveCycle Reader Extensions. When a PDF document is rights-enabled, additional APIs become available for plug-in development.

The following features can be enabled in a PDF document by Adobe LiveCycle Reader Extensions:

- Allow basic form fill-in
- Allow form submit standalone
- Allow dynamic form fields (add and delete form fields)
- Allow dynamic form pages (spawn template pages)
- Allow digital signature
- Allow 2D barcode decoding
- Allow comments to be added, deleted, modified, copied, imported, exported, uploaded, downloaded, and summarized

For more details, see the Acrobat and PDF Library API Overview.
Which APIs Work With Which Products?

As has always been the case, Adobe Reader uses a limited set of APIs. In Acrobat 7, Acrobat Standard offers a subset of the APIs available in Acrobat Professional. All APIs work on Acrobat Professional. In addition, certain APIs can be enabled in Adobe Reader using additional usage rights. For information about which APIs are enabled with each additional usage right, see “Rights-enabled Documents in Adobe Reader” on page 19.

The following APIs work only with Acrobat Professional:

- AVDocPrintSeparations
- PDPageMakeSeparations
- AVPageViewGetNumVisibleInks
- AVPageViewGetPixelInformationAtPoint
- AVPageViewGetVisibleInks
- AVPageViewSetInkPreview
- AVPageViewSetVisibleInks
- PDPageEnumInks

In addition, the printer drawing flags from AVExpT.h are ignored in Acrobat Standard but recognized/used in Acrobat Professional (affects AVDocPrintPagesWithParams on Acrobat Standard):

```c
/** emit crop marks */
#define kAVCropMarks 0x00000001
/** emit trim marks */
#define kAVTrimMarks 0x00000002
/** emit bleed marks */
#define kAVBleedMarks 0x00000004
/** emit Reg marks */
#define kAVRegMarks 0x00000008
/** emit ColorBar marks */
#define kAVColorBarMarks 0x00000010
/** emit PageInfo marks */
#define kAVPageInfoMarks 0x00000020
/** emit Eastern style marks (default is Western-style) */
#define kAVEasternStyleMarks 0x00000040
```

The following table shows which APIs work in Adobe Reader and which work with Acrobat Professional and Standard. To identify which APIs are available in Reader, look at each HFT’s header file. Check xxxProcs.h and see whether an API is defined as UNPROC (UPROC, USPROC) or XNPROC (XPROC, XSPROC). If it uses U or X, then it is not available in
Adobe Reader. Alternatively, if you define `READER_PLUGIN` as 1, then only those APIs that are available in Adobe Reader are available.

<table>
<thead>
<tr>
<th>HFT Name</th>
<th>Configuration</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcroView</td>
<td>Acrobat/Adobe Reader</td>
<td>Acrobat</td>
</tr>
<tr>
<td>AcroViewSweetPea</td>
<td>Acrobat/Adobe Reader</td>
<td>Acrobat</td>
</tr>
<tr>
<td>Cos</td>
<td>Acrobat/Adobe Reader</td>
<td>Acrobat</td>
</tr>
<tr>
<td>PDSysFont</td>
<td>Acrobat</td>
<td>Acrobat</td>
</tr>
<tr>
<td>PDFEditRead</td>
<td>Acrobat</td>
<td>Acrobat</td>
</tr>
<tr>
<td>PDFEditWrite</td>
<td>Acrobat</td>
<td>Acrobat</td>
</tr>
<tr>
<td>PagePDEContent</td>
<td>Acrobat</td>
<td>Acrobat</td>
</tr>
<tr>
<td>AcroColor</td>
<td>Acrobat</td>
<td>Acrobat</td>
</tr>
<tr>
<td>PDSWriter</td>
<td>Acrobat</td>
<td>Acrobat</td>
</tr>
<tr>
<td>PDMetaData</td>
<td>Acrobat</td>
<td>Acrobat</td>
</tr>
<tr>
<td>PDMModel</td>
<td>Acrobat/Adobe Reader</td>
<td>Acrobat</td>
</tr>
<tr>
<td>ASEExtra</td>
<td>Acrobat/Adobe Reader</td>
<td>Acrobat</td>
</tr>
<tr>
<td>PDSRead</td>
<td>Acrobat/Adobe Reader</td>
<td>Acrobat</td>
</tr>
<tr>
<td>AcroSupport</td>
<td>Acrobat/Adobe Reader</td>
<td>Acrobat</td>
</tr>
<tr>
<td>Core</td>
<td>Acrobat/Adobe Reader</td>
<td>Acrobat</td>
</tr>
<tr>
<td>Forms</td>
<td>Acrobat/Adobe Reader</td>
<td>AcroForm</td>
</tr>
<tr>
<td>TTS</td>
<td>Acrobat/Adobe Reader</td>
<td>AcroForm</td>
</tr>
<tr>
<td>Catalog</td>
<td>Acrobat (Professional only)</td>
<td>Catalog</td>
</tr>
<tr>
<td>PDFConsultant</td>
<td>Acrobat</td>
<td>Checkers</td>
</tr>
<tr>
<td>DigSigHFT</td>
<td>Acrobat/Adobe Reader</td>
<td>DigSig</td>
</tr>
<tr>
<td>AcroHLS</td>
<td>Acrobat/Adobe Reader</td>
<td>HLS</td>
</tr>
<tr>
<td>PubSecHFT</td>
<td>Acrobat/Adobe Reader</td>
<td>PPKLite</td>
</tr>
<tr>
<td>Spell</td>
<td>Acrobat</td>
<td>Spelling</td>
</tr>
<tr>
<td>Search</td>
<td>Acrobat/Adobe Reader</td>
<td>Search</td>
</tr>
<tr>
<td>WebLink</td>
<td>Acrobat/Adobe Reader</td>
<td>WebLink</td>
</tr>
</tbody>
</table>
Getting Started

Which APIs Work With Which Products?
Plug-in Overview

The chapter discusses a number of topics basic to plug-in development, including:

- Host Function Tables
- Interaction Between Plug-ins and Acrobat or Adobe Reader
- Callbacks
- Notifications
- Handling Events
- Plug-in Prefixes
- Changing the Acrobat or Adobe Reader User Interface
- Acquiring and Releasing Objects
- Memory Allocation
- Using Modal and Modeless Dialogs
- PDF File Integrity
- Security
- Debugging
- Page View Layers
- Drawing
- Reducing Conflicts Among Plug-ins

Host Function Tables

Host Function Tables (HFTs) are the mechanism through which plug-ins call methods in Acrobat or Adobe Reader, 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 (see “Replacing Built-In Methods in HFTs” on page 103). 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 `ASExtensionMgrGetHFT` method to import each HFT they intend to use. A plug-in requests an HFT by its name and version number. This importing takes place during plug-in initialization, described in “Importing HFTs and Registering for Notifications” on page 28.

When a plug-in calls a method in Acrobat, Adobe Reader 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 Acrobat, Adobe Reader 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 Acrobat or Adobe Reader (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 can also 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 Chapter 13, “Exporting and Importing Host Function Tables”.

Rules to Follow

- Ensure that your plug-in behaves reasonably in the absence of any HFTs it imports. If a missing HFT is required for your plug-in to run, indicate a loading failure by returning `false` from your plug-in's `PluginImportReplaceAndRegister` function. If the HFT is optional, be sure that your plug-in imported it successfully before using it.

- Plug-ins that use new HFTs introduced with Acrobat 7.0 will not run on Acrobat versions prior to 7.0. An attempt to load these HFTs forces the plug-in to fail depending on flags in `PIMain.c` of the form `PI_HFT_OPTIONAL`. By default these flags are undefined, so if your plug-in attempts to load `HFT` and cannot, initialization will fail. If you define `PI_HFT_OPTIONAL` with some value (preferably 1) and the load is not successful, initialization will continue.

- The code in `PIRequir.h` controls which version of an HFT loads. If you don't use any of the functions in an HFT, you should change the appropriate `#define` to 0. The HFT version number that is stated is the `minimum` version required. Thus, if you want to run on Acrobat 6.0, none of the HFT version numbers should be higher than 0x00060000 (because Acrobat 6.0 cannot supply an HFT with all the functions available in Acrobat 6.0) unless that HFT is optional for the plug-in. Put another way, if a plug-in is built to run on Acrobat 6 but uses Acrobat 7 features when they are available, the `minimum` version required is 0x00060000. If the plug-in asks for 0x00060000 and receives 0x00070000, it will work successfully.
If you are writing plug-ins to be run on Adobe Reader, you must:
- `#define READER_PLUGIN` to 1 in your project (`PIRequir.h` needs it)
- `#include "PIRequir.h"

Doing so will make available all HFTs that your Adobe Reader plug-in is allowed to access.

Use the `ASCallbackCreateProto` macro to create callback functions wherever possible, rather than `ASCallbackCreate`. If your plug-in needs the callback later (to unregister it, to destroy it, to pass it to Acrobat or Adobe Reader again, or for any other reason), you should keep the value in a global variable rather than calling `ASCallbackCreate` or `ASCallbackCreateProto` a second time for the same function.

If you no longer need a callback, destroy it using `ASCallbackDestroy`. For example, if your plug-in enumerates menu items once, it should destroy the callback procedure used in the enumeration after it has executed.

Your plug-in should call the appropriate methods to generate notifications. For example, it should call `PDPageNotifyContentsDidChange` if it changes page contents. Your plug-in probably will not break if it does not do this, but other plug-ins might.

---

## Interaction Between Plug-ins and Acrobat or Adobe Reader

This section describes the sequence of operations that Acrobat or Adobe Reader perform to initialize a plug-in, and the operations a plug-in should perform in each step of the sequence.

### Plug-in Loading and Initialization

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`. Macintosh plug-ins must have a file type and creator of `XTND` and `CARO`, respectively. On UNIX, plug-in names must end in `.API` and the plug-in path must be specified correctly in the `.acrorc` file.

When launched, Acrobat or Adobe Reader scan the `PLUG_INS` directory (in the same directory as the program executable) to locate and load plug-ins. In addition, Acrobat or Adobe Reader searches any folders contained inside these folders when looking for plug-ins. This search only goes one level deep.

**Note:** No plug-ins are loaded if the **Shift** key is held down while Acrobat or Adobe Reader launch. Also, if Acrobat or Adobe Reader are running in certified mode, no third-party plug-ins are loaded.
Acrobat and Adobe Reader display a progress message in the bottom line of the splash screen at start-up. As each plug-in is loaded, the progress message shows the plug-in's name. Note that in certified mode, only certified plug-ins are loaded; no third-party plug-ins are loaded (see “Certified Plug-ins” on page 16).

**NOTE:** If Acrobat 6.0 attempts to load a plug-in created with the Acrobat 7.0 SDK using Microsoft Visual Studio .NET 2003, the plug-in may fail to load without displaying a message, or the following message may appear: "This application has failed to start because MSVCR71D.dll was not found." If this occurs, check the runtime library setting on the **C/C++ Code Generation** project property page. If it is set to **Multi-threaded Debug DLL (/MD)** for your Release build, change the setting to **Multi-threaded (/MT)**.

If handshaking is aborted by either Acrobat, Adobe Reader or a plug-in, the user is notified by an alert dialog box displaying a brief explanation. The application continues loading other plug-ins.

**Rules to Follow**
- Do not put up a modal dialog box in your plug-in's initialization or do anything else that might interfere with the successful launching of Acrobat or Adobe Reader. The application may be started via an IAC event in which case there would be not be a user present to respond to your modal dialog box.
- Implement a real **PluginUnload** procedure, to perform any cleanup of memory allocated, objects acquired or menu items added. This routine is called if any of the initialization routines returns **false**. Under normal conditions, it isn't called until Acrobat or Adobe Reader quits; however, future versions of Acrobat or Adobe Reader may unload plug-ins dynamically.

**Handshaking and Initialization**

There are several ways that a plug-in can register one or more of its functions with Acrobat or Adobe Reader so that it can continue to interact with each 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 method, such as the method that opens files. For more information, see “Replacing Built-In Methods in HFTs” on page 103.
- Registering to receive Interapplication Communication (IAC) messages that other applications send to Acrobat or Adobe Reader.

Acrobat and Adobe Reader perform 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.

**Interaction Between Plug-ins and Acrobat or Adobe Reader**
The plug-in must implement the following handshaking function:

\[
\text{ACCB1 ASBool ACCB2 PIHandshake(ASUns32 handshakeVersion, void, *hsData)}
\]

During handshaking, the plug-in receives the \text{hsData} data structure (see \text{PIVersn.h}). Acrobat or Adobe Reader converts all function pointers that are passed in this data structure into callbacks using \text{ASCallbackCreateProto}. See “Callbacks” on page 29 for more information.

The plug-in must fill in its extension name and implement, at minimum, the initialization callback \text{PluginInit}.

The \text{DUCallbacks.h} header file declares all callbacks for your plug-in. Declarations are:

\[
\begin{align*}
\text{ACCB1 ASBool ACCB2 PluginExportHFTs(void);} \\
\text{ACCB1 ASBool ACCB2 PluginImportReplaceAndRegister(void);} \\
\text{ACCB1 ASBool ACCB2 PluginInit(void);} \\
\text{ACCB1 ASBool ACCB2 PluginUnload(void)};
\end{align*}
\]

All callbacks return \textbf{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 \textbf{false}.

\textbf{NOTE:} In addition to defining the callbacks listed above, \text{DUCallbacks.h} includes the the Adobe Dialog Manager header file \text{ADMUtilities.h}.

If either Acrobat, Adobe Reader or a plug-in aborts handshaking, then Acrobat or Adobe Reader displays an alert dialog box showing a brief explanation before loading other plug-ins.

**Exporting HFTs**

After Acrobat finishes handshaking with all the plug-ins, it calls each plug-in’s \text{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. For more information, see Chapter 13, “Exporting and Importing Host Function Tables”.

\textbf{NOTE:} This is the only time that a plug-in can export an HFT.

**Importing HFTs and Registering for Notifications**

After Acrobat or Adobe Reader completes calling each plug-in’s \text{PluginExportHFTs} callback procedure, it calls each plug-in’s \text{PluginImportReplaceAndRegister} callback procedure. In this procedure, plug-ins may perform three tasks:

- Plug-ins may import any special HFTs they use (the standard Acrobat HFTs are automatically imported). Plug-ins also may import HFTs any time after this while the plug-in is running.
- Plug-ins may register for notifications (see \text{AVAppRegisterNotification}). Plug-ins also may register and unregister for notifications any time after this while the plug-in is running.
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.

- Plug-ins may replace any of the Acrobat API’s replaceable methods (see “Replacing Built-In Methods in HFTs” on page 103). Any method replacement must be performed at this time.

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 Acrobat or Adobe Reader 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 Acrobat’s user interface by adding menu items, toolbar buttons, windows, and so on. It is also acceptable to modify Acrobat’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 Acrobat 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 it 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 Acrobat exits.

**Callbacks**

There are several situations in the Acrobat API where Acrobat or Adobe Reader calls 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
Plug-in Overview
Notifications

- In enumeration methods
- Dialogs (filter procs, and so forth)
- Notifications

Whenever your plug-in passes a function pointer to Acrobat, 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 by the Acrobat or Adobe Reader application. Use `ASCallbackDestroy` to dispose of a callback that is no longer needed.

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

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

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

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

---

**Notifications**

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

**NOTE:** See “Notifications” in the *Acrobat and PDF Library 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.

---

**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, Acrobat or Adobe Reader
handles special keys (Esc, Page Up, Page Down, …) or uses the key to select a tool from the
toolbar.

Plug-in Prefixes

You should also 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

Changing the Acrobat or Adobe Reader User Interface

This section describes the kinds of things a plug-in can do to change the Acrobat or Adobe
Reader 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). Acrobat and
Adobe Reader do 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, Acrobat or Adobe Reader automatically add this menu, as well as the About Plug-
ins and Plug-in Help menus. After Acrobat or Adobe Reader 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 Appendix B,
“Registering and Using Plug-in Prefixes.”
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.

Controlling the “About” Box and Splash Screen

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

About Adobe 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 Acrobat or Adobe Reader provides a standardized location for your plug-in help files. 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.

User Interface Guidelines

Plug-ins should follow these user interface guidelines:

● During potentially time-consuming operations, provide user feedback with features such as progress monitors, cancel procs, hourglass cursors, or status dialog boxes.

● If you are adding an authoring tool to a toolbar, make it ignore all annotation types except your own. This way a link or thread will not interfere with the use of your tool.

NOTE: Navigation and selection tools should not ignore annotations. In Acrobat, for example, the Hand, Zoom, and selection tools all follow links. If the user holds down the Shift key, these tools will ignore annotations. All other tools, however, will ignore annotations that are not of the type authored by the tool.

● Use AVToolBarIsRoomFor to check if there is room on the toolbar for your plug-in’s buttons before adding them. If there isn’t, don’t add them. Because space in the toolbar can be limited, plug-ins should generally add a menu item for each toolbar button they add. This provides a backup location for users to access items that do not fit in the
toolbar, and it has the additional benefit of allowing users to access toolbar buttons via shortcut keys (if the menu item has one).

---

**Acquiring and Releasing Objects**

Make sure that calls to `Acquire/Release` pairs match. Objects obtained by `Acquire` methods must ultimately be released. If they are not, Acrobat or Adobe Reader will raise an exception when it discovers a non-zero reference count. This may occur when Acrobat or Adobe Reader quits or when a document is closed.

---

**Memory Allocation**

- Use `ASmalloc` and `ASfree` instead of `malloc` and `free`, or you risk memory leaks.
- Make sure that `ASmalloc/ASfree` pairs match or you will have memory leaks.
- Make sure to use `ASmalloc` to allocate memory for methods that state that Acrobat or Adobe Reader free the memory for you.
- If using C++, the use of the `new` operator allocates memory using the same compiler allocator `malloc` would use. It is best to have your C++ classes derive from a base class which overrides `new`, such as the class `CSafeAlloc`, found in `SafeAlloc.h`.

---

**Using Modal and Modeless Dialogs**

You can use the Adobe Dialog Manager (ADM) as a cross-platform set of dialog interfaces for your plug-in. It enables a single code base to implement dialog behavior on the Macintosh and Windows platforms. ADM enables you to create and manage modal and modeless dialogs and supports a wide variety of control types that can be used with the dialogs.

For details on using ADM, see the *Adobe Dialog Manager Programmer’s Guide and Reference*.

---

**PDF File Integrity**

Your plug-in should manipulate PDF files using PD layer methods rather than Cos layer methods whenever possible. PD layer methods ensure that the resulting file is a valid PDF file, but Cos layer methods do not.
Security

Be sure to check file permissions before assuming your plug-in can manipulate PDF file content. PDDocPermRequest supports a finer set of permissions than the older PDDocGetPermissions. Security handlers can be written with new callbacks to respond to the finer granularity of permission queries. For details, see the Acrobat and PDF Library API Overview.

Debugging

- During development, plug-ins should #define DEBUG 1 globally to enable the parameter type checking that is performed by macros such as ASCallbackCreateProto and to enable the debug exception-handling macros.
- The Acrobat SDK contains a sample plug-in called DebugWin that can be used to display strings from a plug-in. Although this sample was designed to illustrate how to export an HFT from a plug-in, it is also very useful for displaying progress messages without interrupting the plug-in’s execution.
- AVSysBeep provides a simple way to add an audible indication that a certain point has been reached in a plug-in’s code. It can be used to determine whether or not execution follows a certain branch, or reaches a certain point. It is less useful if you have to test many points in your plug-in’s code.
- AVAlertNote displays a dialog box containing a brief message and an OK button. It can be used much like AVSysBeep, except that AVAlertNote stops execution until the OK button is pressed. In addition, it is much more useful than AVSysBeep when you need to test many points, since you can use the message to identify that a particular point has been reached.
- Creating a log file is very helpful when tracing large sections of code or checking values of a number of variables. Use C library calls such as printf or platform-specific code to create a log file containing whatever information is useful for the particular situation.
Plug-in Overview

Page View Layers

Acrobat and Adobe Reader’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 Acrobat and Adobe Reader.

<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 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 and PDF Library API Reference for a discussion of the various `DoClickProc` callbacks).

Annotation handlers provided by plug-ins can reside in any layer. (See the Acrobat and PDF Library API Overview 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, Acrobat’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, Acrobat’s built-in link annotation does not change its layer.)

**Note:** Acrobat and Adobe Reader do not poll `AVAnnotHandlerGetLayerProc` callbacks for changes in value, so be sure to invalidate the page rectangle of an annotation when its layer changes.

For more information, see Chapter 8, “Page Views and Contents”.

NOTE: Acrobat and Adobe Reader do not poll `AVAnnotHandlerGetLayerProc` callbacks for changes in value, so be sure to invalidate the page rectangle of an annotation when its layer changes.
Plug-in Overview
Drawing

**Drawing**

Minimize screen redrawing by using `AVPageViewBeginOperation` and `AVPageViewEndOperation` to bracket any sequence of view changes you may perform. For example, the sequence of changing to another page, scrolling, and zooming would normally redraw the screen three times. But, by calling `AVPageViewBeginOperation` before the sequence and `AVPageViewEndOperation` after it, only one redraw occurs.

**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 Acrobat or Adobe Reader'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.

**Naming**

Correctly name all items your plug-in adds to PDF files or to Acrobat and Adobe Reader. For details on naming rules, see Appendix B, “Registering and Using Plug-in Prefixes.” Failure to do so will probably not cause your plug-in to break immediately, but the results are unpredictable when your plug-in collides with the plug-in of another developer who also ignored the rules, or when PDF is updated and happens to use the same name you did, for a different purpose.

Because PDF uses spaces as token delimiters, it’s important to prevent spaces from creeping into tokens you intend to use as names in the PDF file. This can happen, for example, if you allow a user to type a name into the PDF file and your plug-in does not check the input before writing it to the file.
Basic Plug-in Components

This chapter provides an overview of how plug-ins are loaded, initialized, and unloaded. It shows sample code from files provided by the Acrobat SDK that help implement required functions. This code is the bare minimum needed for a functioning plug-in. Subsequent chapters begin to do real work.

Upon reading this chapter, you will
- Acquire a high-level perspective of plug-in loading and initialization.
- Know where to find the SDK template files to start writing your plug-in.
- Know what methods you are required to, and can optionally, implement in your plug-in.
- Learn how to implement callback methods.

For more information, see “Interaction Between Plug-ins and Acrobat or Adobe Reader” on page 26.

Overview

The Acrobat 7.0 SDK provides some basic components to help you write a plug-in. They can be found in the SDK’s PluginSupport folder:

- PIMain.c (in the Headers folder) provides your plug-in with functionality it needs to communicate with Adobe Reader or Acrobat during startup. “PIMain.c: A Closer Look” on page 44 describes what the functions in PIMain.c do. However, in general you do not need to modify PIMain.c.

- StarterInit.cpp (in the Starter sample) provides a template for plug-in code that you do need to write. This file works with PIMain.c to provide Adobe Reader and Acrobat with the information it needs to load and initialize your plug-in. It contains several routines which are initially empty; the exercises in this document add to them as appropriate.

**Note:** Many of the samples in the SDK contain a file called samplenameInit.cpp. These files have the same routines as StarterInit.cpp, but are filled out with the specific requirements for the sample.

- Header files contain convenient macros, types, and structures for accessing common resources and handling platform-specific issues.
A Plug-in’s Life Cycle

Acrobat uses the following sequence to load, initialize, activate, and unload plug-ins:

1. At startup, Adobe Reader or Acrobat scans its Plug-in directory for all files to load.
2. For each file, it attempts to load the file. If loading succeeds, it calls routines in PIMain.c which complete a handshaking process with the plug-in.
3. Adobe Reader or Acrobat calls a set of callback functions in your plug-in, in this order:
   - PluginExportHFTs
   - PluginImportReplaceAndRegister
   - PluginInit
   This sequence establishes the linkages between the plug-in and Acrobat or Adobe Reader, and between the plug-in and any other plug-ins. See the next section “StarterInit.cpp: A Closer Look” for the definitions of these callbacks.
4. Once all plug-ins are loaded, Acrobat or Adobe Reader continues its own loading and starts the user interface. It adds any plug-in-provided tools to the toolbar and menu items to the menu bar and starts the user session.
5. Whenever the user activates a plug-in tool or resource, Acrobat or Adobe Reader calls the plug-in to perform the associated action.
6. When the user ends the session, each plug-in’s PluginUnload callback method is called to free resources and memory before quitting.

StarterInit.cpp: A Closer Look

You can write the code for your plug-in using the StarterInit.cpp file as a template. StarterInit.cpp contains stub code for callbacks, and it contains the PIHandshake call.

Callbacks

StarterInit.cpp contains stub code for four callbacks. The first three are called by the Adobe Reader or Acrobat during initialization as part of its handshake with each plug-in it loads. If they return true, the plug-in is loaded; otherwise, the plug-in is not loaded.

- **PluginExportHFTs**: (Optional) Allows your plug-in to export functionality to other plug-ins. See Chapter 13, “Exporting and Importing Host Function Tables”, for more information.
- **PluginImportReplaceAndRegister**: (Optional) Allows your plug-in to import functionality from other plug-ins, replace methods in Adobe Reader or Acrobat, and register for notifications. See Chapter 12, “Replacement, Notifications, and Event
Basic Plug-in Components

StarterInit.cpp: A Closer Look

Registration”, and Chapter 13, “Exporting and Importing Host Function Tables”, for more information.

- **PluginInit**: (Required) Initializes your plug-in and sets up other callbacks as needed. It performs user interface modifications and other initializations.

  - **PluginUnload**: (Optional) Cleans up resources when Acrobat or Adobe Reader quits. This routine is optional, but it is good programming practice to use it to clean up allocated memory and destroy callbacks created by your plug-in.

Example 3.1 shows the initial stub code for StarterInit.cpp. Initially, these methods do nothing but return true.

**PluginInit** is the only callback that you must implement. You must also implement the **GetExtensionName** function, which is called from one of the routines in PIMain.c. You modify it to provide the name of your plug-in to Acrobat or Adobe Reader.

**Example 3.1 StarterInit.cpp**

```c
ACCB1 ASBool ACCB2 PluginExportHFTs(void)
{
    return true;
}

ACCB1 ASBool ACCB2 PluginImportReplaceAndRegister(void)
{
    return true;
}
/*
** PluginInit
** The main initialization routine.
** Return true to continue loading plug-in.
** Return false to cause plug-in loading to stop.
*/
ACCB1 ASBool ACCB2 PluginInit(void)
{
    return true;
}
ACCB1 ASBool ACCB2 PluginUnload(void)
{
    return true;
}
/*
** GetExtensionName
** Return the unique ASAtom associated with your plug-in.
*/
ASAtom GetExtensionName()
{
    return ASAtomFromString("ADBE_Starter");
    /* Change to your extension's name */
}
```
PIHandshake

In **PIHandshake**, the plug-in fills in fields of the handshake structure with the following information for Adobe Reader or Acrobat:

- The plug-in's extension name, which it gets from the `GetExtensionName` call:
  ```c
  hsData->extensionName = GetExtensionName();
  ``
- Pointers to the above four methods:
  ```c
  hsData->exportHFTsCallback = 
      ASCallbackCreateProto (PIExportHFTsProcType, &PluginExportHFTs);
  hsData->importReplaceAndRegisterCallback = 
      ASCallbackCreateProto (PIImportReplaceAndRegisterProcType, 
                              &PluginImportReplaceAndRegister);
  hsData->initCallback = 
      ASCallbackCreateProto (PIInitProcType, &PluginInit);
  hsData->unloadCallback = 
      ASCallbackCreateProto (PIUnloadProcType, &PluginUnload);
  ``

These methods are called for each plug-in in the order they are shown here.

---

Defining Callback Functions

In addition to the callbacks such as `PluginInit` that are called during initialization, there are many other situations where plug-ins use callbacks. For example, you provide callbacks to be called when menu items or toolbar buttons are clicked. You will be introduced to more callback types in later chapters of this document.

The general format for declaring a callback is as follows:

```
ACCB1 <return type> ACCB2 <function name> (<parameter list>)
```

For example, the declaration of `PluginInit` in **Example 3.1** conforms to this format:

```
ACCB1 ASBool ACCB2 PluginInit (void)
```

`ACCB1` and `ACCB2` are macros with platform-specific definitions. Their use helps make plug-ins more portable across platforms.

---

Using ASCallbackCreateProto and Other Macros

When you specify a callback method for any of the API data types, you must use the `ASCallbackCreateProto` macro. For example, `PIMain.c` uses `ASCallbackCreateProto` to create a pointer to the plug-in's `PluginInit` callback:

```c
/* Perform your plug-in’s initialization in here */
hsData->initCallback = ASCallbackCreateProto(PIInitProcType, &PluginInit);
```

`ASCallbackCreateProto` does compile-time type checking between the type of the method and the way the method is declared, eliminating a common source of plug-in problems. The macro calls the `ASCallbackCreate` method; however, do not call that
method directly. Similarly, you must not specify an actual pointer to the address of the method when setting up a callback:

hsData->initCallback = &PluginInit; /*Wrong! Use the macro*/

Either at unload time or sometime after the callback is no longer needed, your plug-in should call the **ASCallbackDestroy** method on all such function pointers.

You use similar macros, **ASCallbackCreateReplacement** and **ASCallbackCreateNotification**, when you create callbacks to replace API methods and to create notifications. See Chapter 12, “Replacement, Notifications, and Event Registration,” for examples.

The following example is a framework for creating and destroying a callback, which will be fleshed out in subsequent chapters.

**EXAMPLE 3.2 Using **ASCallbackCreateProto**

```c
AVExecuteProc MyMenuItemExeProcPtr;

ACCB1 void ACCB2 MyMenuItemExeProc (void* data)
{
    /* do something when menu item is selected. */
}

CreatePointer (void)
{
    MyMenuItemExeProcPtr=ASCallbackCreateProto (AVExecuteProc,
        &MyMenuItemExeProc);
}

DestroyPointer (void)
{
    ASCallbackDestroy (MyMenuItemExeProcPtr);
}
```

Note that in this case, a global variable is used to hold the callback pointer returned by **ASCallbackCreateProto** so **ASCallbackDestroy** can be used to destroy the pointer. Another way to handle the pointer returned by **ASCallbackCreateProto** is to pass it directly as a value to the next method that needs the pointer, as shown here:

```c
AVMenuItemSetExecuteProc (menuItem,
    ASCallbackCreateProto( AVExecuteProc, &DisplayAbout),
    NULL);
```

**Defining Callback Types**

Anytime you want to define a type with **typedef** or to declare an instance of a method pointer that is a callback, you need to use the **ACCBPROTO1** and **ACCBPROTO2** macros as shown here.

The SDK header files define types for the various callback methods. Here is an example of a typedef for **PluginInitProcType**:

```c
typedef ACCBPROTO1 ASBool (ACCBPROTO2 *PluginInitProcType) (void);
```
Providing An Extension Name

StarterInit.cpp also contains the following function, which allows your plug-in to pass its extension name to Acrobat or Adobe Reader during handshaking. You replace ADBE:Starter with the name of your plug-in.

```c
ASAtom GetExtensionName()
{
    return ASAtomFromString("ADBE:Starter");
}
```

You must specify a unique name that should include a four-letter prefix denoting your company's name. Please register the name with Adobe to ensure that plug-ins from other parties will not conflict with your plug-in. In the rare event that two or more plug-ins try to use the same name, Acrobat or Adobe Reader loads the first plug-in it encounters and ignores the rest.

PIMain.c: A Closer Look

This section contains details about functions in PIMain.c that are called by Acrobat or Adobe Reader during plug-in loading. You do not need to read this section to get started building a plug-in. These functions are provided by the SDK, and you do not need to change them. However, this information may come in handy during debugging.

PluginMain

The main function is called PlugInMain on Windows and main on the Macintosh. It is the first method Acrobat or Adobe Reader calls when loading a plug-in, and the only method in PIMain.c that your plug-in exports to Acrobat or Adobe Reader. Its prototypes are:

(Windows)

```c
ASBool CALLBACK PlugInMain(ASSInt32 appHandshakeVersion,
    ASSInt32 *handshakeVersion, PISetupSDKProcType* setupProc,
    void* windowsData)
```

(Macintosh)

```c
ACCB1 ASBool ACCB2 main(ASSInt32 appHandshakeVersion,
    ASSInt32 *handshakeVersion, PISetupSDKProcType *setupProc)
```

This function is the first of a sequence of methods that Acrobat or Adobe Reader calls. The plug-in passes Acrobat or Adobe Reader a pointer to each of the methods in a prescribed order. If at any point a method returns false, the application stops loading that plug-in.

**NOTE:** If your plug-in is unable to load, PlugInMain is a good place to set a breakpoint. If this method never gets called, verify your project settings.
The `setupProc` parameter allows the plug-in to provide Acrobat or Adobe Reader with a pointer to the `PISetupSDK` function, which is called next.

```c
*setupProc = PISetupSDK;
```

### PISetupSDK

```
ACCB1 ASBool ACCB2 PISetupSDK (ASUns32 handshakeVersion, void *sdkData)
```

Acrobat or Adobe Reader performs a “handshake” with each plug-in as the plug-in is opened and loaded. `PISetupSDK` checks the version of the handshake structure. In addition, this function:

- Obtains the `extensionID`. The plug-in can store this ID to identify itself to Acrobat or Adobe Reader in subsequent calls to Core API methods that require the ID.
- Initializes Acrobat or Adobe Reader’s Core HFT. With access to the Core API methods in Acrobat or Adobe Reader’s HFT, the plug-in can set up callbacks for every method it wants Acrobat or Adobe Reader to call at a designated point during its execution.
- Provides Acrobat or Adobe Reader with a pointer to the `PIHandshake` method, as shown here:

```c
data->handshakeCallback =
    ASCallbackCreateProto(PIHandshakeProcType, PIHandshake);
    return true;
```
Basic Plug-in Components

PIMain.c: A Closer Look
Menus and Menu Items

When you create a plug-in, you may allow users to access its functionality through a menu command. The Acrobat core API allows you to add new menus and menu items to the Acrobat or Adobe Reader menu bar.

In this chapter, you will learn how to:

● Create a new menu.
● Create menu items and add them to new or existing menus.
● Write callback functions to implement the menu items’ functionality.

NOTE: With the Acrobat API, you cannot use task bars in Acrobat or Adobe Reader.

For more information, see the following:

| Appendix B, “Registering and Using Plug-in Prefixes” |
| Acrobat and PDF Library Overview | The sections “AVMenu”, “AVMenuBar” and “AVMenuItem” |

Adding A Menu Item To an Existing Menu

The code described in this section (see Example 4.1 for the complete sample) modifies StarterInit.cpp to add a menu item called New Item to the File menu. When selected, the menu item displays an alert dialog box.

NOTE: Acrobat and Adobe Reader make it easy to add a menu item alphabetically since each sub-group under a main menu has a separator called endToolsSubGroup1. This separator has a flag that determines how Acrobat or Adobe Reader behaves: if the flag is set and a menu item is added directly before the separator, Acrobat or Adobe Reader places it in alphabetical order relative to the other menu items in the sub-group defined by the separator. So the best way to add an item to Advanced is to find endToolsSubGroup1 and add the item just before it. You can find example code in several of the sample plug-ins.

Creating Callbacks

Your plug-in can implement three types of callback methods for menu items. They are called by Acrobat or Adobe Reader at various times:
Menus and Menu Items

Adding A Menu Item To an Existing Menu

- **Execute**: (Required) Called when the user selects the menu item. Implements the functionality of the menu item.
- **Compute-enabled**: (Optional) Called when Acrobat or Adobe Reader need to determine whether the menu item should be enabled.
- **Compute-marked**: (Optional) Called when Acrobat or Adobe Reader need to determine whether the menu item should be checked.

**Execute Callback**

```c
ACCB1 void ACCB2 ExecProc (void* data)
{
    AVAlertNote ("The Item’s execute proc was called.");
}
```

This routine provides the functionality that is triggered when the menu item is chosen. In this case, it simply calls `AVAlertNote`, which displays a message window showing the text passed to it and an OK button.

In a more complex program, the execution callback might start and manage a dialog or a complex operation.

Note that it is declared using the `ACCB1` and `ACCB2` macros (see “Defining Callback Functions” on page 42).

The `data` parameter (for this and the other callbacks) can be used to maintain private data for the menu item (see Setting the callback functions for the menu item).

**Compute-Enabled Callback**

```c
ACCB1 ASBool ACCB2 ComputeEnabledProc(void* data)
{
    if (AVAppGetNumDocs() > 0)
        return true;
    else return false;
}
```

The `ComputeEnabledProc` should return true when you want the menu item enabled. A common condition is to enable a menu item only when a document is open, as shown here by the call to `AVAppGetNumDocs`.

If your plug-in does not provide this callback, the menu item is enabled by default.

**Compute-Marked Callback**

```c
ACCB1 ASBool ACCB2 ComputeMarkedProc(void* data)
{
    ASBool expressionorcondition = true;
    if (expressionorcondition)
        return true;
    else return false;
}
```

The `ComputeMarkedProc` should return true when you want the menu item to have a checkmark next to it, and false otherwise.
Creating and Adding the Menu Item

The following code is part of the plug-in’s PluginInit callback (described in Chapter 3, “Basic Plug-in Components”). It adds a new menu item to an existing menu.

Creating pointers to the callbacks

First, the code declares global pointers of procedure types defined in the Core API:

```c
AVExecuteProc ExecProcPtr;
AVComputeEnabledProc CompEnabledProcPtr;
AVComputeMarkedProc CompMarkedProcPtr;
```

Next, it uses the ASCallbackCreateProto macro to convert the routines to Acrobat callbacks and assign them to the globals declared above.

```c
ExecProcPtr = ASCallbackCreateProto (AVExecuteProc, &ExecProc);
CompEnabledProcPtr = ASCallbackCreateProto (AVComputeEnabledProc, &ComputeEnabledProc);
CompMarkedProcPtr = ASCallbackCreateProto (AVComputeMarkedProc, &ComputeMarkedProc);
```

Acquiring the File menu

The code acquires an existing menu, the File menu. (When an object in the Acrobat Core API is acquired, it must be released when finished.)

```c
FileMenu = AVMenubarAcquireMenuByName (TheMenubar, "File");
if (FileMenu)
{
    ...
}
```

The subsequent steps execute only if the File menu is not NULL. It is up to your plug-in to check return values, so that it does not pass NULL to API functions when NULL is not valid.

Creating the new menu item

The example code creates a new menu item to put into the File menu:

```c
NewItem = AVMenuItemNew ("New Item", "MyExtn:New Item", NULL, false, NO_SHORTCUT, 0, NULL, gExtensionID);
if (NewItem == NULL)
{
    AAlertNote ("Unable to create a needed Menu Item, not loading.");
    return false;
}
```

The first two parameters for AVMenuItemNew are title and name. The title is the text of the item as it appears in the menu, while the name is a platform-independent identifier for the item. (The same distinction between names and titles also applies to menus.)

The title in the example is New Item, the name is MyExtn:New Item. It is good practice to precede the item's name with the extension's name followed by a colon. Although not enforced by the API, this convention avoids name conflicts.
On Windows only, you can specify a keyboard shortcut by including an ampersand (&) character in the title. An underscore character (_) is placed under the character following the ampersand (char). This allows the user to press `alt-char` to select the item.

The other parameters specify the submenu, shortcut key, modifier keys, icon, and extension. See the *Acrobat and PDF Library API Reference* for more details.

**NOTE:** If you plan to add a submenu to a menu item, you must create the submenu before creating the menu item.

**Setting the callback functions for the menu item**

This code associates the new menu item with the pointers to the callback functions that were created previously:

```c
AVmenuItemSetExecuteProc (NewItem, ExecProcPtr, NULL);
AVmenuItemSetComputeEnabledProc (NewItem, CompEnabledProcPtr, NULL);
AVmenuItemSetComputeMarkedProc (NewItem, CompMarkedProcPtr, NULL);
```

The third parameter to each of these calls allows you to pass a pointer to an optional private data structure that you can maintain to store information related to this procedure. For example, you may have created a structure containing data the execute callback needs. You would provide a pointer to the structure as shown here:

```c
struct
{
    int a;
    int b;
} MyStruct;
AVmenuItemSetExecuteProc (NewItem, ExecProcPtr, &MyStruct);
```

**Adding the new item to the File menu**

Add the menu item to a menu using `AVMenuAddMenuItem`, then release the acquired menu using `AVMenuRelease`.

```c
AVMenuAddMenuItem (FileMenu, NewItem, 1);
AVMenuRelease (FileMenu);
```

Menu items are indexed starting with index value 0; therefore this code makes the item the second one in the menu.

**Finishing Up**

The following code is from the plug-in's `PluginUnload` callback. It cleans up by destroying the callbacks that were created in `PluginInit`.

```c
ACCB1 ASBool ACCB2 PluginUnload (void)
{
    ASCallbackDestroy (ExecProcPtr);
    ASCallbackDestroy (CompEnabledProcPtr);
    ASCallbackDestroy (CompMarkedProcPtr);
    return true;
}
```
The Complete Example

**EXAMPLE 4.1 Adding a menu item to the File menu**

```c
AVExecuteProc ExecProcPtr;
AVComputeEnabledProc CompEnabledProcPtr;
AVComputeMarkedProc CompMarkedProcPtr;

/* Display alert box with message */
ACCB1 void ACCB2 ExecProc (void* data)
{
    AVALertNote ("The Item’s execute proc was called.");
}

ACCB1 ASBool ACCB2 ComputeMarkedProc (void* data)
{
    ASBool expressionorcondition = true;
    if (expressionorcondition)
        return true;
    else return false;
}

ACCB1 ASBool ACCB2 ComputeEnabledProc (void* data)
{
    if (AVAppGetNumDocs() > 0)
        return true;
    else return false;
}

ACCB1 ASBool ACCB2 PluginInit (void)
{
    AVMenu FileMenu = NULL;
    AVMenuItem NewItem = NULL;
    AVMenubar TheMenubar = AVAppGetMenubar ()
    
    ExecProcPtr = ASCallbackCreateProto (AVExecuteProc, &ExecProc);
    CompEnabledProcPtr = ASCallbackCreateProto (AVComputeEnabledProc,
        &ComputeEnabledProc);
    CompMarkedProcPtr = ASCallbackCreateProto (AVComputeMarkedProc,
        &ComputeMarkedProc);
    
    FileMenu = AVMenuAcquireMenuByName (TheMenubar, "File");
    if (FileMenu)
    {
        NewItem = AVMenuItemNew ("New Item", "MyExtn:Newitem", NULL,
            false, NO_SHORTCUT, 0, NULL, gExtensionID);
        if (NewItem == NULL)
        {
            AVALertNote ("Unable to create a Menu Item,
                not loading.");
            return false;
        }
    }
}
```
Creating A New Menu And Adding A Menu Item

Example 4.2 modifies PluginInit to create a new menu and add a menu item to it, instead of acquiring an existing menu. It uses the same callback methods used in the previous example, with the additions and changes discussed here.

Creating the new menu

This code calls AVMenuNew to create a new menu with the title New Menu and name MyExtn:New Menu. The parameters are the same as the equivalent ones in AVMenuItemNew.

```c
AVMenu NewMenu = NULL;
NewMenu = AVMenuNew ("New Menu", "MyExtn:New Menu", gExtensionID);
```

The following steps assume the menu was successfully created.

Creating the new menu item and adding it to the menu

This code is very similar to the previous example, except that it adds the new item to the newly created menu.

```c
NewItem = AVMenuItemNew ("New Item", "MyExtn:New Item", NULL, false, NO_SHORTCUT, 0, NULL, gExtensionID);
if (NewItem == NULL)
{
    AVALertNote ("Unable to create a needed Menu Item, not loading.");
}
```
Menus and Menu Items

Creating A New Menu And Adding A Menu Item

```c
AVMenuRelease (NewMenu);
return false;
}
AVMenuAddMenuItem (NewMenu,NewItem, 0);

Adding the new menu to the menu bar

This code adds the new menu to the menu bar, then releases it.

```c
AVMenubarAddMenu (TheMenubar, NewMenu, APPEND_MENU);
AVMenuRelease (NewMenu);
```

`APPEND_MENU` is a constant meaning that the menu is added to the end of the menu bar. If you want your menu to appear after a particular Acrobat or Adobe Reader menu, acquire the menu that your plug-in's new menu will follow, get its index value using the method `AVMenubarGetMenuIndex`, and assign your menu an index value one greater than that. Menus, like menu items, use zero-based indexing.

The Complete Example

EXAMPLE 4.2 Creating a new menu and adding a menu item to it

```c
ACCB1 ASBool ACCB2 PluginInit (void)
{
    AVMenu NewMenu = NULL;
    AVMenuItem NewItem = NULL;
    AVMenubar TheMenubar = AVAppGetMenubar ()

    ExecProcPtr = ASCallbackCreateProto (AVExecuteProc, &ExecProc);
    CompEnabledProcPtr = ASCallbackCreateProto (AVComputeEnabledProc, &ComputeEnabledProc);
    CompMarkedProcPtr = ASCallbackCreateProto (AVComputeMarkedProc, &ComputeMarkedProc);
    NewMenu = AVMenuNew ("New Menu", "MyExtn:New Menu", gExtensionID);

    if (NewMenu)
    {
        NewItem = AVMenuItemNew ("New Item", "MyExtn:New Item", NULL, false, NO_SHORTCUT, 0, NULL, gExtensionID);
        if (NewItem == NULL)
        {
            AVAlertNote ("Unable to create a needed Menu Item, not loading.");
            AVMenuRelease (NewMenu);
            return false;
        }
        AVMenuItemSetExecuteProc (NewItem, ExecProcPtr, NULL);
        AVMenuItemSetComputeEnabledProc (NewItem, CompEnabledProcPtr, NULL);
        AVMenuItemSetComputeMarkedProc (NewItem, CompMarkedProcPtr, NULL);
    }
```
AVMenuAddMenuItem (NewMenu, NewItem, 0);
AVMenuAddMenu (TheMenubar, NewMenu, APPEND_MENU);
AVMenuRelease (FileMenu);
return true;
else return false;
}
This chapter introduces the concept of exception handling for plug-ins.

For more information, see the following documents:

<table>
<thead>
<tr>
<th>Acrobat and PDF Library API</th>
<th>Exception Handling Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acrobat and PDF Library API</td>
<td>The sections “AVMenu,” “AVMenuBar,” and “AVMenuItem”</td>
</tr>
<tr>
<td>Overview</td>
<td>The chapter “Handling Errors”</td>
</tr>
<tr>
<td>Acrobat and PDF Library API</td>
<td>ASAtom</td>
</tr>
<tr>
<td>Reference</td>
<td></td>
</tr>
</tbody>
</table>

**Exception Handling Overview**

Core API methods generally do not return error codes; instead, they raise exceptions when errors occur. Acrobat and Adobe Reader provide a default exception handler, but this handler is not able to back gracefully out of an unfinished operation.

Plug-ins can add their own exception handlers to trap and handle various exceptions. Each time an exception occurs, your exception handler can perform any necessary cleanup, such as releasing memory.

Error handling in Acrobat is handled with macros (defined in CorCalls.h). Much like the throw/catch exception handling mechanism in C++, the DURING, HANDLER, and END_HANDLER macros define the scope of the exception handler and what should happen if an exception is raised. These macros are implemented using the standard C library functions setjmp() and longjmp(), which allow for consistent exception handling in C and C++.

Acrobat version 4.0 and higher have redefined the exception handling macros to detect errors in macro use at compile time during Debug builds. For details, see the comments in CorCalls.h.

For more information on Acrobat’s exception handling, see the Acrobat and PDF Library API Overview.
Exception Handling Macros

The following example illustrates macros that the core API provides for the purpose of handling errors: DURING, HANDLER, and END_HANDLER, illustrated here:

\[
\text{DURING} \\
\text{Code that may raise an exception} \\
\text{HANDLER} \\
\text{Code to handle the exception} \\
\text{END_HANDLER} \\
\text{Code after exception raising code}
\]

The code block between the DURING and HANDLER macros is executed. If your plug-in or Acrobat or Adobe Reader raises an exception at any point in the DURING block, execution proceeds with the code between HANDLER and END_HANDLER. If no exception occurs in the DURING . . . HANDLER block, the code after END_HANDLER executes.

To determine if a method can raise an exception, see the method description in the Acrobat and PDF Library API Reference.

Returning From a DURING...HANDLER Block

To return from a method within a DURING . . . HANDLER block, do not use a return statement. Instead, use the macros E_RETURN(x) or E_RTRN_VOID. E_RETURN(x) returns a value and E_RTRN_VOID does not return a value.

The E_RETURN(x) macro must not call a function that might raise an exception, because the macro 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, but instead calls the next handler up the stack. For example:

\[
\text{E\_RETURN(mynfn())}
\]

This is dangerous if there is any possibility that myfn could raise an exception. If you need to call such a function, it is best to do it this way:

\[
\text{result = myfn();} \\
\text{E\_RETURN(result);}
\]

Now, if myfn raises an exception, your handler will be executed.
Using ASRaise

Although many core API methods raise exceptions, some methods do return error codes or NULL if something goes wrong. Inside a `DURING...HANDLER` block, if your plug-in detects one of these cases, it can call the `ASRaise` method, which generates an exception. This has the effect of causing the `HANDLER...END_HANDLER` code to execute, just as if the original method raised an exception.

See Chapter 14, “Cos Objects and Methods“ for several examples of using ASRaise.

ShowException: An Error Display Method

Example 5.1 defines the `ShowException` method, which is used in some of the exception handler examples in this document. `ShowException` displays a message followed by a string for the error code. The first parameter is the string you want to display; the second is the error code. To obtain this error code, you can use the `ERRORCODE` macro provided in the Acrobat header files.

**EXAMPLE 5.1 ShowException**

```c
/* Shows a message along with the string for the exception
 * msg contains a string to be prepended to the error string for err. */
void ShowException (char* msg, ASInt32 err)
{
    char Msg[400];
    char buf[301];

    ASGetErrorString (err, buf, 300);
    strcpy (Msg, msg);
    strcat (Msg, " Exception info: ");
    strcat (Msg, buf);
    AVAlertNote (Msg);
}
```

For more information on handling exceptions, see the chapter “Handling Errors,” in the Acrobat and PDF Library API Overview.
**General Rules to Follow**

- Use `DURING/HANDLER` blocks around any code that can raise an exception. If the code is in a plug-in and the exception is not caught, Acrobat or Adobe Reader catches the exception and raise a dialog box that may or may not be helpful to the user. In general, Acrobat Viewer (AV) layer methods do not raise exceptions but will display a dialog box.

  **NOTE:** You should assume that, unless a method is explicitly documented as not raising an exception, it may raise any exception (not just those documented in the Acrobat and PDF Library API Reference).

- Do not use `return;` or `return(x);` within `DURING/HANDLER` clauses.

  Inside a `DURING/HANDLER` clause, you must use one of the `E_RETURN` macros in place of `return` so that the frame pushed onto the exception stack by `DURING` is popped before returning from the function. Using `return` inside a `DURING` clause returns from the current function without popping the values that were pushed onto the exception stack by the `DURING` macro. In a worst case scenario, doing this in a loop can overrun the exception stack array and result in a general protection fault (GPF). With the new macro definitions, a compile-time error such as one of the following errors occurs in debug builds:

  ```
  error C2115: '=' : incompatible types
  error C2679: binary '=' : no operator defined that takes a right-hand operand of type 'const int' (or there is no acceptable conversion)
  ```

- The macros `E_RTRN_VOID` and `E_RETURN(x)` should not be used within `HANDLER/END HANDLER` blocks. Simply use `return` instead.

  The macros pop the most recent frame off the exception stack and return, so using them outside a `DURING/HANDLER` block (where no frame has been pushed onto the stack) leaves the exception stack out of sync with program execution. The next exception raised causes the wrong environment to be restored, which can lead to memory leaks.

  With the new macro definitions, an error such as the following occurs at compile time for debug builds:

  ```
  error C2224: left of '.E_RETURNOutsideDURINGBlock' must have struct/union type
  ```

- Release resources if you end up in a `HANDLER` block.

- Declare variables that may change value inside `DURING` blocks as volatile. If you do not, the `setjmp/longjmp` mechanism used in Acrobat’s exception handling probably will not preserve their values if the compiler decides to use registers to store them.

- If you nest `DURING` handlers, do not return from the inner one.

- Do not forget to release resources, as necessary, in an exception handler. For example, if you acquire a menu in preparation for assigning a filename to it, and opening the file raises an exception, do not forget to release the menu in your exception handler.

- Check PD layer objects with the appropriate `isValid` method if there is a chance that Acrobat or Adobe Reader (or another plug-in) has made the object invalid since you last used it.
- Use `ASGetConfiguration` to guarantee you have an Acrobat or Adobe Reader configuration that supports the functionality your plug-in requires. Check for functionality instead of product name (that is, check `CanEdit`, not `Product`).
- Check return values.
- Initialize variables before using them.
- Set structure members to 0 before using them.
- Check function parameters before using them.
- Check variables before dividing them to avoid divide-by-zero errors.
In this chapter, your plug-in will add a tool button to a toolbar. In the process, it introduces two important concepts: **ASAtom** and exception handling.

For more information, see the following documents:

| Appendix B, “Registering and Using Plug-in Prefixes” | The chapter “Registering and Using Plug-in Prefixes” |
| Acrobat and PDF Library API Overview | The sections “AVMenu,” “AVMenuBar,” and “AVMenuItem” |
| Acrobat and PDF Library API Reference | **ASAtom** |

**Toolbar And Buttons**

Acrobat and Adobe Reader have one or more toolbars, each of which contains buttons. In the core API, a toolbar is represented by an **AVToolbar** object, and a button by an **AVToolButton** object.

An **AVToolButton** has a name, an icon, and a flag to indicate if it is a separator. A separator button separates groups of one or more buttons.

You can define two callbacks associated with a button:

- An **execute callback**, which Acrobat or Adobe Reader calls when the user clicks the button.
- A **compute-enabled callback**, which are called whenever it updates the toolbar.

**Note:** Separators do not have execute or compute-enabled callbacks.

**Example 6.1** shows how to add a button to a toolbar. First, we discuss some concepts that are used in the example, **ASAtoms** and exception handling.
ASAtoms

The Acrobat core API frequently uses objects called ASAtoms in place of character strings, for faster string access and comparison. For example, Example 6.1 creates a new button by calling AVToolButtonNew, whose first parameter is an ASAtom for the name of the button. The ASAtom must be created first, using ASAtomFromString:

```c
ASAtom MyButtonAtom;
AVToolButton MyButton;
MyButtonAtom = ASAtomFromString ("MyButton");
MyButton = AVToolButtonNew (MyButtonAtom, GetToolIcon(id), FALSE, FALSE);
```

To get the character string from an ASAtom use ASAtomGetString.

Acrobat and Adobe Reader maintain a list of ASAtoms that have unique numbers for each string. If the string is already in the list when your plug-in calls ASAtomFromString, the number for that string is returned. Once your plug-in creates an ASAtom for a string, the string cannot be removed from Acrobat or Adobe Reader’s ASAtom table.

Adding A Button To the Toolbar

Example 6.1 shows the code to add a button to a toolbar. AVToolButtonNew creates the button. The first parameter is an ASAtom corresponding to the button’s name. The second parameter is an icon (of type AVIcon) representing the appearance of the button.

**NOTE:** The sample as written here will not compile, because you need the code for the GetToolIcon function, which is platform-specific. Refer to the SDK samples (such as BasicUI) for concrete examples of using AVToolButtonNew.

This code adds the button to the toolbar.

```c
oldButton = AVToolBarGetButtonByName(ToolBar, ASAtomFromString("FirstPage"));
AVToolBarAddButton(ToolBar, MyButton, FALSE, oldButton);
```

The new button is positioned relative to an existing button, in this case the FirstPage button, assigned to oldButton. The third parameter is FALSE, placing the new button after the old button. TRUE would place it before the old button. If the existing button is NULL, passing TRUE places the new button at the beginning of the toolbar and FALSE places it at the end.

See the Acrobat and PDF Library API Reference for more information about the methods used here.

Error Handling in the Example

In Example 6.1 below, note the use of the exception-handling macros:

DURING
MyButton = AVToolButtonNew (ASAtomFromString ("MyExtn:MyButton"),
    GetToolIcon(id), FALSE, FALSE);

HANDLER
    AVAlertNote ("An exception was raised, not loading.");
    ASCallbackDestroy (ExecProcPtr);
    ASCallbackDestroy (CompEnabledProcPtr);
    return false;
END_HANDLER

If an error occurs, AVToolButtonNew raises an exception (rather than returning NULL). The HANDLER...END_HANDLER block is then executed, and does the following:

1. Displays an error message. This may or may not be useful, depending on the situation.
2. Performs appropriate cleanup operations. This is really the main purpose of an exception handler: to make sure that if something goes wrong, that a known state can be restored.
   In this example, it destroys the callbacks that were created earlier, since there is no longer any need for them.
3. Returns false from the function. This return value from the initialization callback tells Acrobat or Adobe Reader not to load the plug-in.

The Complete Example

**Example 6.1  Adding a button to a toolbar**

```c
AVExecuteProc ExecProcPtr;
AVComputeEnabledProc CompEnabledProcPtr;
ACCB1 void ACCB2 ExecProc (void* data)
{
    AVAlertNote ("The Button’s execute proc was called.");
}

ACCB1 ASBool ACCB2 ComputeEnabledProc (void* data)
{
    ASBool expressionorcondition = true;
    if (expressionorcondition)
        return true;
    else return false;
}

ACCB1 ASBool ACCB2 PluginInit (void)
{
    AVToolBar ToolBar = AVAppGetToolBar ();
    AVToolButton MyButton = NULL;
    AVToolButton oldButton = NULL;
    ExecProcPtr = NULL;
    CompEnabledProcPtr = NULL;
```
Toolbars and Buttons

Adding A Button To the Toolbar

```
ExecProcPtr = ASCallbackCreateProto (AVExecuteProc, &ExecProc);
CompEnabledProcPtr = ASCallbackCreateProto (AVComputeEnabledProc, &ComputeEnabledProc);
DURING
    MyButton = AVToolButtonNew (ASAtomFromString("MyExtn:MyButton"), GetToolIcon(id), FALSE, FALSE);
HANDLER
    AVAlertNote ("An exception was raised, not loading.");
    ASCallbackDestroy (ExecProcPtr);
    ASCallbackDestroy (CompEnabledProcPtr);
    return false;
END_HANDLER

AVToolButtonSetExecuteProc (MyButton, ExecProcPtr, NULL);
AVToolButtonSetComputeEnabledProc (MyButton, CompEnabledProcPtr);
oldButton = AVToolBarGetButtonByName(ToolBar, ASAtomFromString("FirstPage"));
AVToolBarAddButton(ToolBar, MyButton, FALSE, oldButton);
return true;
```

```
ACCB1 ASBool ACCB2 PluginUnload (void)
{
    ASCallbackDestroy (ExecProcPtr);
    ASCallbackDestroy (CompEnabledProcPtr);
    return true;
}
```
This chapter illustrates how a plug-in can perform basic operations on documents, such as opening, closing, reading and writing.

This chapter discusses:

- The different layers of the Acrobat core API (AV, PD, Cos and AS)
- Some important methods you use in working with documents, from the AV and PD layers, and also bridge methods that span both layers.
- Cross-platform issues that arise when working with files and documents.

**NOTE:** This chapter (and subsequent chapters) contains sample code that illustrates certain points but may not represent a standalone plug-in.

For more information, see the following:

| Acrobat and PDF Library API Overview | The sections “AVMenu”, “AVMenuBar”, and “AVMenuItem” |
| Acrobat and PDF Library API Reference | ASFile |
| PDF Reference, fifth edition, version 1.6 | Chapter 2, “Overview” |
|                                      | Chapter 3 , “Syntax,” through Section 3.8.4, “Rectangles” |

**Documents**

In the Acrobat Core API, there are several types of objects that can represent documents and files:

- An **AVDoc** object (from the AcroView or **AV layer**) represents a document as a window in Acrobat or Adobe Reader. A single **AVDoc** object exists for each displayed document. Operations on **AVDoc** objects are usually visual modifications to the document’s view.

- A **PDDoc** object (from the Portable Document or **PD layer**) represents a document as a PDF file. You work with **PDDoc** objects to make changes to a document's contents. Through **PDDoc** objects, you can access other PDF components, such as **PDPage** page objects (discussed in Chapter 8, “Page Views and Contents”“

- A **CosDoc** object (from the **Cos layer**). The Cos layer is a lower-level way of representing the objects in a PDF document. For details, see Chapter 14, “Cos Objects and Methods”“
● An **ASFile** object (from the Acrobat Support or **AS layer**) represents a document as an open file. **ASFile** (and **ASFileSys**) methods are used to help deal with cross-platform file-handling issues. See the *Acrobat and PDF Library API Overview* for more information.

---

### Opening a PDF File For Viewing

To open a PDF file for viewing, use the **AVDocOpenFromFile** method:

```c
AVDoc AVDocOpenFromFile (ASPathName pathName, ASFileSys fileSys, char* tempTitle);
```

- **pathName** is an **ASPathName**, which is a platform-independent path name for the file. Before calling this routine, you must create the **ASPathName**. There are various methods of creating an **ASPathName**; **Example 7.1** shows a way to create it on the Windows and Macintosh platforms.
- **fileSys** is an **ASFileSys**, which represents a file system. Typically you pass the return value from **ASGetDefaultFileSys** that represents the target platform's file system. It is also possible to create your own file system semantics and pass a different value.
- **tempTitle** is a title to be used for a temporary file.

The following code opens a PDF file for viewing:

```c
AVDoc MyDoc = AVDocOpenFromFile (ThePathName, ASGetDefaultFileSys (), NULL);
```

### Creating a Platform-Independent Path Name

This example shows one way to create an **ASPathName** on the Windows and Macintosh platforms. It uses the **ASFileSysCreatePathName** method to convert a platform-specific pathname to an **ASPathName**. This method is defined as follows:

```c
ASPathName ASFileSysCreatePathName (const ASFileSys fileSys, ASAtom pathSpecType, const void* pathSpec, const void* mustBeZero);
```

- **fileSys** is the **ASFileSystem** to use; pass **NULL** to use the default file system.
- **pathSpec** is the **ASFileSpec** to use; pass **NULL** to use the default file system.
- **pathSpecType** is an **ASAtom** indicating what type of data **pathSpec** is. This example uses C strings to specify the file names. For example, a Windows **pathSpec** would look like "C:\MyFolder\MyFile", and for Mac OS it would look like "MyDisk:MyFolder:MyFile".
The last parameter is unused. See the *Acrobat and PDF Library API Reference* for more information about this method.

**EXAMPLE 7.1  Creating a platform-independent path name**

```c
// Returns the ASPathName; returns NULL if errors occur.
ASPathName MakeCrossPlatformASPathName (char* platformPathName)
{
    ASPathName ThePathName = NULL;
    ThePathName = ASFileSysCreatePathName (NULL,
        ASAtomFromString("Cstring"), platformPathName, 0);
    return ThePathName;
}
```

**Accessing AVDoc Objects Already Opened**

To access AVDoc objects that are already opened by the user, your plug-in can:

- Call the `AVAppGetActiveDoc` method to obtain the AVDoc object associated with the active document.
- Call the `AVAppEnumDocs` method to enumerate all the open documents.

Once you have an AVDoc object, you can perform operations on it such as:

- Print the document with `AVDocDoPrint`.
- Save the document with `AVDocDoSaveAs`.

**Bridging Core API Layers**

If you want to operate on the contents of a PDF file, and not just manipulate its representation in the user interface, you must use a PDDoc object. To get a PDDoc object corresponding to an AVDoc object, you have already obtained, call `AVDocGetPDDoc`. `AVDocGetPDDoc` is referred to as a *bridge method* because it allows your plug-in access to PDDoc objects from the AV layer. There are several bridge methods which connect the different Core API layers (AV, PD, and Cos).

AVDoc objects always have an underlying PDDoc object. Operations on an AVDoc object usually result in visible modifications to the document’s view. If you just want your plug-in to make changes to a document’s contents, it can open a PDDoc object directly (that is, without going through an AVDoc object to get the PDDoc object) and use PD layer methods, as shown in the following section.
Opening a PDDoc Object

At times, you may want to open a PDF file “behind the scenes”; that is, access the contents of the file without having it be visible in Acrobat or Adobe Reader. Example 7.2 shows how to do that using the `PDDocOpen` method:

```c
PDDoc PDDocOpen (ASPathName fileName, ASFileSys fileSys,
    PDAuthProc authProc, ASBool doRepair);
```

`authProc` is a pointer to an **authorization callback** method. If your plug-in creates an encrypted document, it must provide an **authorization callback** method for opening it. In the authorization callback, your plug-in should obtain whatever information is needed to determine whether the user is authorized to open the file, then call `PDDocAuthorize` (which returns the permissions that the authentication data enables).

If the file is encrypted and `authProc` is `NULL` or returns `false`, then `PDDocOpen` raises the error `pdErrPassword`.

See the Acrobat and PDF Library API Reference for more information about this method.

**Example 7.2   Opening a PDDoc Object**

```c
ASPathName ThePathName;
void OpenAPDDoc (ASPathName thePathName)
{
    PDDoc myPDDoc = NULL;
    PDAuthProc myAuthProcPtr = NULL;
    DURING
    myAuthProcPtr = ASCallbackCreateProto (PDAuthProc,
        &authProc);
    myPDDoc = PDDocOpen (thePathName, ASGetDefaultFileSys (),
        myAuthProcPtr, false);
    ASCallbackDestroy (myAuthProcPtr);
    if (myPDDoc == NULL)
        ASRaise (ERR_GENERAL);
    HANDLER
        Code to deal with not being able to open the file ...
    END_HANDLER
}
/* Authorization proc required by PDDocOpen--does nothing. */
ASBool authProc (PDDoc TheDoc)
{
    return true;
}
```
Accessing Non-PDF Files

The following example shows how to open a file, write data to it, and then read it back in, using ASFile methods. Typically, you would only use this approach when opening non-PDF files (for example, you might want to use a text file to keep track of some information for your plug-in).

In the example, the CreateDataFile function creates a file and writes data to it. ReadDataFile opens the file and reads from it. ExeProc calls both functions.

The example uses the MakeCrossPlatformASPathName function from Example 7.1. In addition, it uses the following core API methods:

- AVAlertNote displays an error message if something goes wrong.
- ASFileSysOpenFile opens a file using the modes specified.
- ASFileWrite writes data to the file.
- ASFileClose closes the file.
- ASFileRead reads data from the file.
- ASFileGetEOF gets the current size of a file.

See the Acrobat and PDF Library API Reference for more details on these methods.

EXAMPLE 7.3 Opening, writing to, and reading from a file

```c
char* gDataBuf = "This is some data in the file."

// Returns false if error, true otherwise.
ASBool CreateDataFile (char* pathname)
{
    ASPathName path = NULL;
    ASFile TheFile = NULL;
    ASInt32 err = 0;
    ASInt32 numBytes = 0;
    ASInt32 mode = ASFILE_WRITE | ASFILE_CREATE;

    path = MakeCrossPlatformASPathName (pathname);
    if (path == NULL)
    {
        AVAlertNote ("Unable to gain access to data file.");
        return false;
    }

    err = ASFileSysOpenFile (ASGetDefaultFileSys (), path, mode, &TheFile);
    if (err != 0) // returns 0 is no error
    {
        AVAlertNote ("Unable to open data file for writing.");
        return false;
    }

    // Write data to file
    numBytes = strlen(gDataBuf);
    if (ASFileWrite (TheFile, gDataBuf, numBytes) == numBytes)
    {
        AVAlertNote ("Data written to file.");
    }
    else
    {
        AVAlertNote ("Unable to write to file.");
        return false;
    }

    // Close the file
    if (ASFileClose (TheFile) != AS_TRUE)
    {
        AVAlertNote ("Unable to close file.");
        return false;
    }

    return true;
}
```

numBytes = ASFileWrite (TheFile, gDataBuf, strlen (gDataBuf));
if (numBytes != strlen (gDataBuf))
{
    ASFileClose (TheFile);
    AVAlertNote ("Number of bytes written was not the expected number.");
    return false;
}
ASFileClose (TheFile);
return true;

void ReadDataFile (char* pathname)
{
    ASPATHName path = NULL;
    ASFile TheFile = NULL;
    ASInt32 err = 0;
    ASInt32  mode = ASFILE_READ;
    char Data [500];
    char buf [500];

    path = MakeCrossPlatformASPathName (pathname);
    if (path == NULL)
    {
        AVAlertNote ("Unable to gain access to data file.");
        return;
    }

    err = ASFileSysOpenFile (ASGetDefaultFileSys (), path, mode, &TheFile);
    if (err != 0) // returns 0 if no error
    {
        AVAlertNote ("Unable to open data file for reading.");
        return;
    }

    err = ASFileRead (TheFile, Data, ASFileGetEOF (TheFile));
    if (err != strlen (gDataBuf))
        AVAlertNote ("Number of bytes read was not the expected amount of bytes.");

    // NULL terminate the string.
    Data[ASFileGetEOF (TheFile)] = '\0'
    ASFileClose (TheFile);
    strcpy (buf, "Data read was: ");
    strcat (buf, Data);
    AVAlertNote (buf);
    return;
}

ACCB1 void ACCB2 ExeProc (void* data)
{
    if (CreateDataFile ("c:\somestuf.txt") == false);
    return;
    ReadDataFile ("c:\somestuf.txt");
This chapter shows how to access the modifiable contents of a PDF file. In the process, it discusses:

● Several objects that are used (**AVPageView**, **PDPage**, and **PDEContent**) and their methods.

● Two coordinate systems used by Acrobat or Adobe Reader (user space and device space), and how to convert between them.

● How PDFEdit methods differ from PD layer methods.

For more information, see the following:

<table>
<thead>
<tr>
<th>Acrobat and PDF Library API</th>
<th>The section “Coordinate Systems”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>PDF Reference, fifth edition, version 1.6</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2, “Overview”</td>
</tr>
<tr>
<td>Chapter 3, “Syntax,” through Section 3.8.4, “Rectangles”</td>
</tr>
<tr>
<td>Chapter 4, “Graphics”</td>
</tr>
</tbody>
</table>

---

**Page Views**

The *page view* is the area of the Acrobat or Adobe Reader window that displays the visible contents of a document page. In the Core API, it is represented by the **AVPageView** object. To obtain an **AVPageView** object, use the **AVDocGetPageView** method.

Using **AVPageView** methods, you can perform many operations related to the page view, such as:

● Display a page

● Select a zoom factor

● Scroll the displayed page

● Highlight one or more words

● Control screen redrawing

● Traverse the view stack that records where users have been in a document

**Note:** To control the size of the page view, your plug-in can use the **AVWindowSetFrame** and **AVDocSetSplitterPosition** methods.
Displaying a Page in a Document

The following function could be used to display a given page of a document. It takes as its parameters an AVDoc object (obtained previously) and an integer representing the page number to display.

**EXAMPLE 8.1  Displaying a page in a document**

```c
void ViewPage (AVDoc avDoc, ASInt32 pageNum)
{
    ASInt32 numPages;
    PDDoc pdDoc;
    AVPageView pageView;

    pdDoc = AVDocGetPDDoc (avDoc);
    numPages = PDDocGetNumPages (pdDoc);
    if (pageNum < numPages)
    {
        pageView = AVDocGetPageView (avDoc);
        AVPageViewGoto (pageView, pageNum);
        AVPageViewDrawNow (pageView);
    }
}
```

To be on the safe side, it checks to make sure that the page number is less than the number of pages in the document. To do this, it calls `PDDocGetNumPages` to find out how many pages are in the PDF document. However, since it must pass a PDDoc object to `PDDocGetNumPages`, it first uses the bridge method `AVDocGetPDDoc` to get the PDDoc object corresponding to the AVDoc object.

**Note:** The API uses zero-based indexing for pages, so the first page in a document is numbered zero. Because the highest page in the document is actually `pageNum - 1`, `pageNum` must be less than `numPages`.

After making sure it has a valid page (you might want to do something else if the page is not valid), the function calls `AVDocGetPageView` to get the AVPageView object for the page. Then it calls `AVPageViewGoto`, which goes to the page. This call (like others that affect views) does not automatically redraw the screen, because Acrobat or Adobe Reader may not have had a chance to process everything that could change the page's appearance. The `AVPageViewDrawNow` call makes sure that the view is updated and redrawn immediately.

**Note:** You can invalidate rectangles in the AVPageView to ensure they are redrawn, by calling `AVPageViewInvalidateRect`, then `AVPageViewDrawNow`. 
Coordinate Systems

There are two types of coordinate systems used by core API methods. Sometimes it is necessary to convert between them, depending on which methods you use.

- **User space** is the coordinate system used within PDF files. It specifies coordinates for most objects in the PD layer of the core API. In user space, you use the `ASFixedRect` type to represent a rectangle.

  ![User space coordinate system](image)

- **Device space** specifies coordinates in screen pixels. It is used in the AV layer of the API to specify screen coordinates of windows. In device space, you use the `AVRect` type.

  ![Device space coordinate system](image)

See Chapter 4 of the *PDF Reference, fifth edition, version 1.6* for more information about these coordinate systems.

The `AVPageViewRectToDevice` method can transform a rectangle’s coordinates from user space to device space. For example, your plug-in might use PD methods to find the coordinates of a rectangle in the PDF document. To display an outline around the
rectangle, it first must convert the rectangle’s user coordinates to screen (device) coordinates.

---

**PDF Pages**

Pages in a PDF file are represented by the `PDPage` object. Pages have several properties, including: annotations, contents, resources, bounding box, media box, crop box, rotation state, and Cos objects.

Numerous `PDPage` methods are available for your plug-in to examine objects, add new objects, remove objects, and modify existing objects in a page.

To access a `PDPage` object, your plug-in must first obtain the appropriate `PDDoc` object, either directly or by calling the bridge method `AVDocGetPDDoc`. It can then call the `PDDocAcquirePage` method to acquire the page. See “Acquiring and releasing objects” on page 77 for details on acquiring objects.

---

**Accessing Page Contents with PDFEdit**

To access the contents of PDF pages, you can use PD layer methods. In addition, the core API provides the `PDFEdit` API to provide easy access to the contents of a PDF page. PD layer and PDFEdit methods are useful for different reasons.

PDFEdit methods are useful in dealing with items on a page such as image, path, graphic, and text objects. This API treats the contents of a page as a list of objects whose values and attributes can be modified. `PDEContent` is a container type of object that can contain several types of `PDEElements` including `PDEText`, `PDEPath`, and `PDEImage`.

Each PDFEdit object encapsulates all the relevant information about itself. A `PDEText` object, for example, contains text and its attributes such as font and position on the page. It can access single characters or multiple character strings, but not words.

To access words, you need to use PD layer methods (see Chapter 9, “Finding Words in a Document”). In addition, PD layer methods are useful when your plug-in is looking for extra information about a page, such as annotations.

---

**Accessing PDE Objects**

Example 8.1 finds all the elements on the current page and outlines them in red. It uses `PDPageView`, `PDPage`, and `PDEContent` methods to access `PDEObjects`. The following sections describe the code in this example in detail.
Initialization

The PluginInit callback in this example is similar to the ones in the previous chapters. It sets up a menu item in the Window menu called Outline PDE Elements, and it sets the execute callback for the menu item to the OutlinePDEElements routine.

```c
windowMenu = AVMenuBarAcquireMenuByName(menubar, "Window");
menuItem = AVMenuItemNew("Outline PDE Elements",
            "ADBE:Outline PDE Elements",
            NULL, false, NO_SHORTCUT, 0, NULL, gExtensionID);
AVMenuItemSetExecuteProc(menuItem, ASCallbackCreateProto(AVExecuteProc, &OutlinePDEElements), NULL);
```

The next sections describe the OutlinePDEElements function.

Accessing a Page’s Content

To access the page’s content, the function calls the following methods:

```c
avDoc = AVAppGetActiveDoc();
pageView = AVDocGetPageView (avDoc);
pageNum = AVPageViewGetPageNum(pageView);
pdDoc = AVDocGetPDDoc(avDoc); /* bridge method to PD doc */
pdPage = PDDocAcquirePage(pdDoc, pageNum); /* acquire current page */
pdeContent = PDPageAcquirePDEContent(pdPage, gExtensionID);
```

First, it gets an AVDoc object representing the active document. From this AVDoc object it can obtain the AVPageView object with AVDocGetPageView. Then it gets the page number for the current page with AVPageViewGetPageNum.

To access the contents of the page, it needs PD layer calls, so AVDocGetPDDoc gets the PDDoc object corresponding to the AVDoc object.

Acquiring and releasing objects

The function uses PDDocAcquirePage to get the PDPage object for the current page. In general, API method names that contain the word acquire or create increment a reference counter. The reference counter keeps track of the objects to let other plug-ins know when objects are in use and when they can be deleted safely. When you no longer need an object, it should be released with the corresponding release method (in this case PDPageRelease). When in doubt, check the Acrobat and PDF Library API Reference to determine whether objects use reference counters.

To obtain the PDEContent object, the function uses the PDPageAcquirePDEContent method. It can then get, add, or remove elements using PDEContent methods. After it is done with the page, it calls PDPageReleasePDEContent.

**NOTE:** After a plug-in modifies elements, it can convert the PDEContent object back to page contents and resources with PDPageSetPDEContent. This example does not use this method.
Setting up a color

The following code calls `AVPageViewSetColor` to set the color to be used for subsequent drawing. The second parameter is the color, represented by a `PDColorValue` object, which is set up in advance to be the color red.

```c
/* define red */
red.space = PDDeviceRGB;
red.value[0] = ASInt32ToFixed(1);
red.value[1] = 0;
red.value[2] = 0;
AVPageViewSetColor(pageView, &red);
```

Accessing page elements

Next, the code accesses each element of the page content. It uses `PDEContentGetNumElems` to determine how many elements there are and loops through them. (You could also do this by enumerating the objects with the `PDEEnumElements` method.)

```c
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 */
```

Testing For text

After getting an element, the routine checks if the element is of type text. It will use `PDETextGetBBox` to get the bounding box of a text element, and `PDEElementGetBBox` for other types.

Note that your plug-in can cast `PDEElements` to `PDEObjects` as shown here, so that it can use `PDEObject` methods.

```c
if (PDEObjectGetType((PDEObject) pdeElement) == kPDEText)
{ ...
```

A `PDEText` object is a container of text runs. A text run is a `PDEType` and can be a single character or multiple characters having the same attributes in a PDF file. If the function finds `PDEText`, it loops through the text runs and gets the bounding box of each. Casting here works because `PDEText` is one type of `PDEElement`.

```c
numRuns = PDETextGetNumRuns ((PDEText) pdeElement);
for (i = 0; i < numRuns; i++)
{ /* get the bounding box of the text run */
    PDETextGetBBox ((PDEText) pdeElement, kPDETextRun, i, &bbox);
    /* convert from user space bbox to device space rect */
    AVPageViewRectToDevice (pageView, &bbox, &rect);
    AVPageViewDrawRectOutline (pageView, &rect, 1, NULL, 0);
} ```
The routine must convert coordinates from user space to device space
\texttt{(AVPageViewRectToDevice)} so it can draw a box around the element
\texttt{(AVPageViewDrawRectOutline)}. When \texttt{OutlinePDEElements} is done processing
the entire page, it releases the \texttt{PDPage} and the \texttt{PDEContent}.

\begin{verbatim}
    PDPageRelease(pdPage);
    PDPageReleasePDEContent(pdPage, gExtensionID);
\end{verbatim}

The Complete Example

\textit{Example 8.1 Accessing PDE objects}

ACCB1 void ACCB2 OutlinePDEElements (void* data)
{
    PDEContent pdeContent;
    AVDoc avDoc;
    AVPageView pageView;
    PDDoc pdDoc;
    PDPage pdPage;
    PDEElement pdeElement;
    PDColorValueRec red;
    ASInt32 numElems, pageNum;
    int j;

    avDoc = AVAppGetActiveDoc();
    pageView = AVDocGetPageView (avDoc);
    pageNum = AVPageViewGetPageNum(pageView)
    pdDoc = AVDocGetPDDoc(avDoc); /* bridge method to PD doc */
    pdPage = PDDocAcquirePage(pdDoc, pageNum); /* acquire first document page */
    pdeContent = PDPageAcquirePDEContent(pdPage, gExtensionID);

    /* define red */
    red.space = PDDeviceRGB;
    red.value[0] = ASInt32ToFixed(1);
    red.value[1] = 0;
    red.value[2] = 0;
    /*
    ** set the drawing color to be used for subsequent
    ** drawing by AVPageViewDrawRectOutline
    */
    AVPageViewSetColor(pageView, &red);
    numElems = PDEContentGetNumElems(pdeContent);
    for (j = 0; j < numElems; j++) /* loop through elements to find text */
    {
        ASFixedRect bbox;
        AVRect rect;
        pdeElement = PDEContentGetElem(pdeContent, j);
        /* determine if the object is of type text */
        if (PDEObjectGetType((PDEObject) pdeElement) == kPDEText)
```c
{  
    int numRuns;
    int i;
    numRuns = PDETextGetNumRuns ((PDEText) pdeElement);
    for (i = 0; i < numRuns; i++)
    {   
        /* get the bounding box of the text run */
        PDETextGetBBox ((PDEText) pdeElement, kPDETextRun, i, &bbox);
        /* convert from user space bbox to device space rect */
        AVPageViewRectToDevice (pageView, &bbox, &rect);
        AVPageViewDrawRectOutline (pageView, &rect, 1, NULL, 0);
    }
}
else
{
    PDEElementGetBBox (pdeElement, &bbox);
    AVPageViewRectToDevice (pageView, &bbox, &rect);
    AVPageViewDrawRectOutline (pageView, &rect, 1, NULL, 0);
}
PDPageRelease(pdPage);
PDPageReleasePDEContent (pdPage, gExtensionID);
}

ACCB1 ASBool ACCB2 PluginInit(void)
{
    /* add a menu item to the Window menu */
    AVMenubar menubar = AVAppGetMenubar();
    AVMenu windowMenu;
    AVMenuItem menuItem;
    /* error checking */
    if (!menubar)
    return false;
    /* See the Acrobat and PDF Library API Reference for a list of the menu names */
    windowMenu = AVMenubarAcquireMenuByName(menubar, "Window");
    /* append it to the Window menu */
    menuItem = AVMenuletNewItem("Outline PDE Elements", "ADBE:InternalName", NULL,
                            false, NO_SHORTCUT, 0, NULL, gExtensionID);
    AVMenuItemSetExecuteProc(menuItem, ASCallbackCreateProto(AVExecuteProc,
                            &OutlinePDEElements), NULL);
    AVMenuAddMenuitem(windowMenu, menuItem, APPEND_MENUITEM);
    AVMenuRelease (windowMenu);
    return true;
}
```
Finding Words in a Document

This chapter describes how your plug-in can find words in a PDF file.

For more information, see the following:

<table>
<thead>
<tr>
<th>Acrobat and PDF Library Overview</th>
<th>The sections “Coordinate Systems”, “PDWord”, and “PDWordFinder”</th>
</tr>
</thead>
</table>
| PDF Reference, fifth edition, version 1.6 | Section 5.2, “Text State Parameters and Operators”  
|                                    | Section 5.3, “Text Objects” |

Finding Words

As mentioned in Chapter 8, “Page Views and Contents”, the PD layer provides specific methods for accessing words on a page (unlike PDFEdit).

There are no guaranteed methods for finding words in every possible situation. Two word-finding indicators are:

- Presence of non-alphanumeric characters such as dashes.
- Offsets between characters. (While character offsets are well-defined quantities in a PDF file, word numbers are calculated by Acrobat or Adobe Reader’s word finder algorithm and, therefore, may change as the algorithm is improved in future versions.)

PDWord

A PDWord object represents a word in a PDF file. Each word contains a sequence of characters in one or more PDStyles. PDStyle methods enable your plug-in to get the fonts, font sizes, and colors used in a word, while PDWord methods determine font data and character types (such as lowercase, ligature, and digit) based on text matrix changes in the page contents.

See Sections 5.2 and 5.3 in the PDF Reference, fifth edition, version 1.6 for details on PDF text operators.
Finding Words in a Document

Example of Finding and Displaying Words

PDWordFinder

A PDWordFinder provides access to the PDWords of a PDPage. 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

You create a PDWordFinder using PDDocCreateWordFinderEx, and can specify options such as the type of encoding into which character codes will be translated. There are two primary ways of using a PDWordFinder:

- With the PDWordFinderEnumWords method, find words one at a time from a page or a document (see Example 9.1). Your plug-in can provide a callback function that can be called for each word found.
- With the PDWordFinderAcquireWordList method, build a table containing all the words in the document in the order in which they are encountered in the PDF file or according to their location on the page.

To alter the PDWords or their placement on the page, you must use PDFEdit methods.

Example of Finding and Displaying Words

Example 9.1 displays the words in a PDF document.

Initialization

The initialization function in Example 9.1 sets up a menu item execute procedure (DisplayWords) to display the words it finds. See Chapter 4, “Menus and Menu Items”, for details on setting up menus and menu items.

It also sets up a configuration record, PDWordFinderConfigRec, that is used by PDCreateWordFinderEx.

Creating the PDWordFinder

The code sets currentPDDoc by getting the active document (the frontmost document in Acrobat or Adobe Reader), then calling the bridge method AVDocGetPDDoc to get the PDDoc object. It then gets the current page number (to pass later to the word enumerator) by calling AVDocGetPageView and AVPageViewGetPageNum.

```c
AVDoc currentAVDoc = AVAppGetActiveDoc();
PDDoc currentPDDoc = AVDocGetPDDoc(currentAVDoc);

AVPageView currentPageView = AVDocGetPageView (currentAVDoc);
ASInt32 pageNum = AVPageViewGetPageNum (currentPageView);
```
To create the word finder, DisplayWords calls PDDocCreateWordFinderEx. It passes currentPDDoc as well as other parameters indicating defaults for the encoding information, encoding vector, ligature table, and word-finding algorithm.

    PDWordFinder wordFinder = PDDocCreateWordFinderEx(currentPDDoc,
                WF_LATEST_VERSION, toUnicode, pConfig);

**Enumerating the Words**

The example calls PDWordFinderEnumWords to enumerate the words in the current page, providing a routine that is called for each word found.

    PDWordFinderEnumWords(wordFinder, pageNum, ASCallbackCreateProto
                (PDWordProc, &wordEnumerator), NULL);

The routine applied to each word is wordEnumerator, which does the following:

- Removes punctuation characters from the word with PDWordFilterWord. The encoding information passed to PDDocCreateWordFinderEx determines which characters are removed.
- Gets the word as a string with PDWordGetString.
- Displays the string in an alert with AVAlertConfirm. Finally, AVAlertConfirm displays the word. If the user clicks OK the plug-in code displays the next word, and so on, until all words for the document page have been displayed or the user clicks Cancel (in which case wordEnumerator returns false).

    PDWordFilterWord(wInfo, stringBuffer, 99, &wordLength);
    stringBuffer[wordLength] = 0;
    /* Do this to get raw word breaks */
    PDWordGetString(wInfo, stringBuffer, 99);
    return AVAlertConfirm(stringBuffer);
The Complete Example

**Example 9.1 Using WordFinder to display words in a PDF document**

```c
/* Enumeration function for each word */
ACCB1 ASBool ACCB2 wordEnumerator(PDWordFinder wObj, PDWord wInfo, 
    ASInt32 pgNum, void *clientData)
{
    char stringBuffer[100];
    ASInt16 wordLength;
    /* Do this to remove punctuation */
    PDWordFilterWord(wInfo, stringBuffer, 99, &wordLength);
    stringBuffer[wordLength] = 0;
    /* Do this to get raw word breaks */
    PDWordGetString(wInfo, stringBuffer, 99);
    return AVAlertConfirm(stringBuffer);
}
ACCB1 void ACCB2 DisplayWords(void *data)
{
    AVDoc currentAVDoc = AVAppGetActiveDoc();
    PDDoc currentPDDoc = AVDocGetPDDoc(currentAVDoc);
    AVPageView currentPageView = AVDocGetPageView(currentAVDoc);
    ASInt32 pageNum = AVPageViewGetPageNum(currentPageView);

    /* setting up Custom Configurations for the WordFinder */
    PDWordFinderConfigRec pConfig;
    memset(&pConfig, 0, sizeof(PDWordFinderConfigRec));
    pConfig.recSize = sizeof(PDWordFinderConfigRec);
    pConfig.ignoreCharGaps = true;
    pConfig.ignoreLineGaps = true;
    pConfig.noAnnots = true;
    pConfig.noEncodingGuess = true; // leave non-Roman single-byte font alone

    /* Create PDWordFinder */
    PDWordFinder wordFinder = PDDocCreateWordFinderEx(currentPDDoc, 
        WF_LATEST_VERSION, toUnicode, pConfig);

    PDWordFinderEnumWords(wordFinder, pageNum, 
        ASCallbackCreateProto(PDWordProc, &wordEnumerator), NULL);

    PDWordFinderDestroy(wordFinder);
}
ACCB1 ASBool ACCB2 PluginInit(void)
{
    /* add a menu item to the Window menu */
    AVMenubar menubar = AVAppGetMenubar();
}
AVMenu windowMenu = AVMenubarAcquireMenuByName(menubar, "Window");

AVMenuItem menuItem = AVMenuItemNew("Display Words", "ADBE:DisplayWords", NULL,
    false, NO_SHORTCUT, 0, NULL, gExtensionID);

AVMenuItemSetExecuteProc (menuItem,
    ASCallbackCreateProto (AVExecuteProc, &DisplayWords), NULL);
AVMenuAddMenuItem(windowMenu, menuItem, APPEND_MENUITEM);

/* release to clean up */
AVMenuRelease (windowMenu);
return true;
Finding Words in a Document

Example of Finding and Displaying Words
This chapter introduces PDF annotations. It provides examples of how your plug-in can:

- Find the annotations in a document.
- Get and set the contents of a text annotation.
- Create a new annotation.

For more information, see the following:

<table>
<thead>
<tr>
<th>Acrobat and PDF Library API Overview</th>
<th>The sections “PDLinkAnnot” and “PDTextAnnot”</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDF Reference, fifth edition, version 1.6</td>
<td>Section 8.4, “Annotations”</td>
</tr>
</tbody>
</table>

## About Annotations

The Acrobat Core API provides methods for working with annotations in PDF files. Annotations are represented by the `PDAnnot` object. `PDAnnot` methods can get and set various properties of annotations, such as color, date, title, location, and subtype.

There are several types of annotations, identified by their subtype. Each subtype may have additional properties beyond the basic ones. The subtype for text annotations (also called notes) is “Text”; the subtype for link annotations is “Link”. These two types of annotations are represented by separate API objects, `PDTextAnnot` and `PDLinkAnnot`.

There are several other types of annotations, and it is also possible for a plug-in to create custom annotation types. You may not be able to access all the properties of other annotation types through PD layer methods; in that event, you can use methods in the Cos layer of the core API (see Chapter 14, “Cos Objects and Methods”).

For a list of the standard annotation types PDF supports, see Section 8.4.5, “Annotation Types,” in the `PDF Reference, fifth edition, version 1.6`.

## Finding Annotations

Example 10.1 shows how to find all the annotations in a document and display an alert showing the annotation’s subtype. It defines three functions, to iterate through the pages, to find the annotations on those pages, and to act on each annotation.
Iterating Through the Pages

The first function, VisitAllAnnots, iterates through all the pages in a document. It finds the number of pages in the document using PDDocGetNumPages, then for each page, it does the following:

- Acquires the PDPage object. This is necessary in order to access annotations that are on the page.
- Calls the DoPage function (explained below).
- Releases the PDPage object with PDPagerelease.

Note the use of the ShowException function and the ERRORCODE macro, explained in “ShowException: An Error Display Method" on page 57.

Finding the Annotations on a Page

The DoPage function calls PDPagenumerAnnots to get the number of annotations on the page, then does the following for each one:

- Calls PDAnnotIsValid to determine if the annotation is valid (has not been deleted).
- If valid, it calls ActOnAnnot (described below) to operate on the annotation.

Acting on the Annotation

The ActOnAnnot function finds the annotation’s subtype and displays it. The function:

- Calls PDAnnotGetSubtype, which returns an ASAtom corresponding to the type. (see “ASAtoms” on page 62).
- Converts the atom to a string with ASAtomGetString.
- Displays the string with AVAlertNote.

The Complete Example

EXAMPLE 10.1 Finding annotations

/* Visit all the annotations in a document. */
void VisitAllAnnots (PDDoc pd)
{
    PDPage p;
    ASInt32 i;
    for (i = 0; i < PDDocGetNumPages (pd); i ++) {
        DURING
            p = NULL;
        p = PDDocAcquirePage (pd, i);
        HANDLER
            ShowException ("Trying to get a page.", ERRORCODE);
        END_HANDLER
}
if (p) {
    DoPage (p);
    PDPageRelease (p);
}

/* Visit all the annotations on a page. */
void DoPage (PDPage p)
{
    PDAnnot annot;
    ASInt32 i;
    for (i = 0; i < PDPageGetNumAnnots (p); i++) {
        annot = PDPageGetAnnot (p, i);
        if (PDAnnotIsValid (annot))
            ActOnAnnot (annot);
    }
}

/* Display an alert showing an annotation’s subtype. */
void ActOnAnnot (PDAnnot annot)
{
    char* ptr;
    char buf [ 200];
    ASAtom atom;

    DURING
        atom = PDAnnotGetSubtype (annot);
        ptr = (char * ) ASAtomGetString (atom);
    HANDLER
        ShowException ("Trying to get an annotation’s subtype.",
                      ERRORCODE);
    END_HANDLER
    sprintf (buf, "Annotation’s subtype is %s", ptr);
    AVAlertNote (buf);
}
Annotations

Getting and Setting the Contents of a Text Annotation

This example shows how to get and set the contents of a text annotation.

Checking for Text Annotations

The function `GetAndSetTextContents` takes a `PDAnnot` parameter. The first thing it does is determine whether this annotation is a text annotation by checking its subtype (as in the previous example).

If the annotation is not text, an alert is displayed. If it is text, the function uses the `CastToPDTextAnnot` macro to convert the `PDAnnot` to type `PDTextAnnot`. This allows it to use methods that work specifically with `PDTextAnnot`s.

**NOTE:** You can convert from a `PDTextAnnot` back to a `PDAnnot` with `CastToPDAnnot`. You also can convert from a `PDAnnot` or `PDTextAnnot` to a link annotation using `CastToPDLinkAnnot`.

Exception Handling

This example uses two `DURING/HANDLER/END_HANDLER` blocks. The first one contains the code to get the text, and the second contains the code to set the text. If an error occurs, the `ShowException` handler displays a message (see “ShowException: An Error Display Method” on page 57).

If an exception occurs during memory allocation, `E_RTRN_VOID` is used to return from the function, because it does not return a value. See “Returning From a DURING...HANDLER Block” on page 56 for details.

Getting and Setting the Annotation Text

To get the text of an annotation, you use the `PDTextAnnotGetContents` method, defined as follows:

```c
ASInt32 PDTextAnnotGetContents (PDTextAnnot aTextAnnot, char* buffer, ASInt32 bufSize);
```

`aTextAnnot` is the annotation object, `buffer` is a pointer to a buffer to hold the text, and `bufSize` is the maximum number of characters that the buffer can hold. The return value is the number of characters written into the buffer, or if zero was passed for `bufSize`, the full length of the text.

The size of the text is unknown, so the function calls `PDTextAnnotGetContents` twice:

- The first time it passes zero for the buffer size and `NULL` for the buffer address. Therefore, no buffer is allocated, but the text length is returned and stored in `BufSize`. The call to `ASmalloc` then uses `BufSize` to allocate the buffer.
NOTE: The allocated memory needs to be deallocated using **ASfree**. This is done after the first **DURING/HANDLER/END_HANDLER** block, because if the second call to **PDTextAnnotGetContents** raised an error, the memory would never be deallocated.

- The second time it passes **BufSize** for the length, so the method returns the full text in the buffer whose starting address is **buf**.

Finally, the function calls **PDTextAnnotSetContents** to set the text of the annotation.

### The Complete Example

**EXAMPLE 10.2  Getting and setting text annotation contents**

```c
void GetAndSetTextContents ( PDAnnot theannot )
{
    ASInt32 BufSize;
    char* buf;
    char setbuf [ 200 ];
    PDTextAnnot annot;

    /* Check that it’s a text annotation. */
    if ( PDAnnotGetSubtype ( theannot ) != ASAtomFromString ( "Text" ) ) {
        AValertNote ( "Not a text annotation." );
        return;
    }
    annot = CastToPDTextAnnot ( theannot );

    DURING
    /* Get the number of bytes in the text annotation. */
    BufSize = PDTextAnnotGetContents ( annot, NULL, 0 ) +1;
    buf = ( char * ) ASmalloc ( ( os_size_t ) BufSize );
    if ( buf == NULL )
        E_RTRN_VOID
    PDTextAnnotGetContents ( annot, buf, BufSize );
    /* Act on the buffer’s contents here. */
    .
    .
    HANDLER
    ShowException ( "Trying to get annotation’s contents.",
                     ERRORCODE );
    END_HANDLER

    if ( buf == NULL )
        ASfree ( buf );

    sprintf ( setbuf, "This is the new text for the annotation." );
    DURING
    PDTextAnnotSetContents ( annot, setbuf, strlen ( setbuf ) );
    HANDLER
}
```
Creating a New Text Annotation

The examples so far show how to work on existing annotations. Example 10.3 creates a new text annotation. It does the following things:

● Creates a rectangle to specify the annotation’s location.
● Calls PDPageCreateAnnot to create the annotation, specifying the type (“Text”) and the location.
● Opens the annotation with PDTextAnnotSetOpen and sets its contents with PDTextAnnotSetContents.
● Adds the annotation to the page with PDPageAddAnnot.

EXAMPLE 10.3  Creating a new text annotation

PDTextAnnot CreateNewAnnot (PDPage p)
{
    PDAnnot annot;
    ASFixedRect fr;
    PDTextAnnot textannot;
    char* ptr = "This is initial text."

    /*
     ** Position the annotation .5 inches down from top of
     ** page, .5 inches from left of page, and make it 100
     ** by 100 points.
     */
    fr.left = ASInt32ToFixed (36);
    fr.top = ASInt32ToFixed (792-36);
    fr.right = ASInt32ToFixed (136);
    fr.bottom = ASInt32ToFixed (792-136);

    DURING
        annot = PDPageCreateAnnot (p, ASAtomFromString ("Text"),
                                   &fr);
    HANDLER
        ShowException ("Trying to create a new annotation.",
                       ERRORCODE);
    END_HANDLER

    textannot = CastToPDTextAnnot (annot);
Creating a New Text Annotation

DURING

    PDTextAnnotSetOpen (textannot, true);
    PDTextAnnotSetContents (textannot, ptr, strlen (ptr));
    PDPageAddAnnot(p, -2, textannot);

HANDLER

    ShowException ("Trying to set annot properties.", ERRORCODE);

END_HANDLER

Bookmarks

PDF files can display a document outline consisting of bookmarks. This chapter shows how a plug-in can work with bookmarks. It provides examples that show how to:

- Get the first bookmark in the bookmark tree.
- Search for a bookmark.
- Create a new bookmark and place it in the bookmark tree.

For more information, see the following:

*PDF Reference, fifth edition, version 1.6*

Section 8.2.2, “Document Outline”

Section G.5, “Outline Tree Example”

**About Bookmarks**

Bookmarks can be created interactively by the user through the Acrobat or Adobe Reader user interface, or they can be generated programmatically.

In PDF files, bookmarks are represented as outline objects. Each bookmark has properties including:

- A title (the text that appears on the screen).
- An action that specifies what happens when a user clicks on the bookmark. The typical action for a user-created bookmark is to move to another location in the current document, although other actions can be specified.

**Bookmark Organization**

Bookmarks are organized in a tree in which each bookmark 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 open if its children are visible on screen, and closed if they are not.
Bookmark Methods

The core API has several methods to operate on bookmarks, which are represented by the **PDBookmark** object. A plug-in can:

- Get and set various attributes of a bookmark (such as its title or action or whether it is open)
- Search for a bookmark
- Add new siblings, children or subtrees

Finding Bookmarks

The following sections illustrate ways to find bookmarks.

The Root Bookmark

Every document has a **root bookmark**. The root bookmark does not represent a physical bookmark, but is the root from which all bookmarks in the tree are descended.

The following code illustrates how to get a document’s first bookmark. First it calls **PDDocGetBookmarkRoot** to get the bookmark root, then calls **PDBookmarkGetFirstChild** to get the first child of the root. If there are no bookmarks, **PDBookmarkGetFirstChild** returns **NULL**.

**EXAMPLE 11.1  Getting a document’s first bookmark.**

```c
PDBookmark GetFirstBookmark (PDDoc d)
{
  PDBookmark theroot, thebookmark;
  theroot = PDDocGetBookmarkRoot (d);
  thebookmark = PDBookmarkGetFirstChild (theroot);
  return thebookmark;
}
```

Visiting All Bookmarks

The following code shows how to visit all the bookmarks in a tree. It is a recursive function that takes a single parameter, a **PDBookmark** object. If the root bookmark is passed, the function visits all the bookmarks in the document. If any other bookmark is passed, the function visits the subtree that has that bookmark as its root.

First it calls **PDBookmarkIsValid**, which just makes sure the bookmark has not been deleted. (The root bookmark can never be deleted.) Note the comment in the code indicating that this would be a good place to actually do something with the bookmark (such as setting its destination).
Next, the call to `PDBookmarkHasChildren` determines whether there are any bookmarks under the current one. If there are not, the function returns. If there are, it calls `PDBookmarkGetFirstChild` to get the first child. Then, in the `while` loop, it recursively calls `VisitAllBookmarks` (that is, the routine is calling itself) until there are no more first children. Then it finds the next sibling with `PDBookmarkGetNext`, and the process continues.

**EXAMPLE 11.2 Visiting all bookmarks**

```c
/* Recursively go through bookmark tree to visit each bookmark. */
void VisitAllBookmarks (PDBookmark aBookmark)
{
    PDBookmark treeBookmark;
    DURING
        if (!PDBookmarkIsValid (aBookmark))
            E_RTRN_VOID
        /*
           ** This is where you would act on the current bookmark.
           ** Now visit that bookmark’s children if it has any children
           */
        if (PDBookmarkHasChildren (aBookmark))
        {
            /* Get the first child of the bookmark */
            TreeBookmark = PDBookmarkGetFirstChild (aBookmark);
            while (PDBookmarkIsValid (treeBookmark))
            {
                VisitAllBookmarks (treeBookmark);
                treeBookmark = PDBookmarkGetNext (treeBookmark);
            }
        }
    HANDLER
        ...
    END_HANDLER
}
```

**Searching for A Bookmark by Title**

The following example searches for an existing bookmark, given its title.

This function calls `GetFirstBookmark` (from Example 11.1) to find the document’s first bookmark as a starting point for the search. It then calls `PDBookmarkGetByTitle` to find the first bookmark whose title matches the `title`.

`PDBookmarkGetByTitle` is defined as follows:

```c
PDBookmark PDBookmarkGetByTitle (PDBookmark aBookmark, char* aName,
                                  ASInt32 nameLen, ASInt32 maxdepth);
```

`aBookmark` is the root of the subtree to search. In this case, the function uses the first bookmark in the document as the root to search.
**Finding Bookmarks**

*Name* is the name, and *nameLen* is the length of the name.

*maxdepth* is the number of subtree levels to search. The function here passes -1, which means to search the entire subtree. 0 means look only at a `Bookmark`, not any of its children. 1 means look at a `Bookmark` and its children, but not grandchildren.

**Example 11.3 Searching for a bookmark by title**

```c
void SearchForBookmark (PDDoc myDoc)
{
    PDBookmark firstBookmark;
    PDBookmark theBookmark;
    char* theTitle = "Find this bookmark";

    DURING
      theRoot = GetFirstBookmark (myDoc);
      theBookmark = PDBookmarkGetByTitle (firstBookmark, theTitle, strlen (TheTitle), -1);
      if (PDBookmarkIsValid (theBookmark)) {
          /* Act on the bookmark */
      }
      else {
          /* Could not find the bookmark you wanted */
      }
    HANDLER
    ...
    END_HANDLER
}
```
Creating New Bookmarks

The following example creates two new bookmarks and places them in the tree. It uses PDBookmarkAddNewChild to add a new bookmark that is a child of the root bookmark, then uses PDBookmarkAddNewSibling to add another bookmark at the same level.

EXAMPLE 11.4 Creating a bookmark and placing it in the tree

```c
void MakeNewBookmark (PDDoc myDoc)
{
    PDBookmark theRoot;
    PDBookmark newBk;

    DURING
        theRoot = PDDocGetBookmarkRoot (myDoc);
        newBk = PDBookmarkAddNewChild (theRoot, "New bookmark");
        if (PDBookmarkIsValid (newBk)) {
            /* Act on new bookmark */
            ...
            PDBookmark newSib;
            newSib = PDBookmarkAddNewSibling (newBk, "New sibling");
            if (PDBookmarkIsValid (newSib)) {
                /* Sibling is ok */
                ...
            } else {
                /* Sibling is invalid */
                ...
            }
        } else {
            /* Creation failed */
        }
    HANDLER
        AVAlertNote("Could not create all bookmarks.");
    END_HANDLER
}
```
PDF Objects in the Bookmark Tree

To get the CosObj object representing a particular bookmark, use PDBookmarkGetCosObj. To convert from the CosObj object back to a PDBookmark object, use PDBookmarkFromCosObj.

This section provides PDF examples of the Cos objects that represent the following elements of a simple bookmark tree:

Root
  First visible bookmark
  First child
  First sibling

Root
When a document has bookmarks, its Catalog dictionary has an Outlines key whose value is an indirect reference to the root bookmark

3 0 obj
<<
/Pages 5 0 R
/Type /Catalog
/Outlines 6 0 R
>>

First Visible Bookmark
Here is the PDF for an example root bookmark. Your plug-in can get this by calling PDDocGetBookmarkRoot.

6 0 obj
<<
/Count 3
/Type /Outlines
/First 7 0 R
/Last 9 0 R
>>
endobj
**First Child**

Indirect object 7 is the first child of the root bookmark. It is the first visible bookmark. The value of the `Dest` key describes the action that occurs when the bookmark is clicked.

```latex
7 0 obj
<<
/Title (First visible bookmark)
/Parent 6 0 R
/Dest [ 4 0 R /FitR -122 137 735 797 ]
/Next 9 0 R
/First 8 0 R
/Last 8 0 R
/Count 1
>>

This is the first child of the above bookmark:

```latex
8 0 obj
<<
/Title (First Child)
/Dest [ 4 0 R /FitR -122 137 735 797 ]
/Parent 7 0 R
>>
endobj
```

**First Sibling**

This is the first sibling of indirect object 7 above.

```latex
9 0 obj
<<
/Title (First sibling)
/Prev 7 0 R
/Parent 6 0 R
/Dest [ 4 0 R /FitR -122 137 735 797 ]
>>
endobj
```
This chapter discusses the following topics:

- Replacing API methods with your plug-in's own methods
- Receiving notification of events that occur
- Registering events with Acrobat or Adobe Reader

For more information, see the following:

<table>
<thead>
<tr>
<th>Acrobat and PDF Library API Overview</th>
<th>The sections “Replacing Methods” and “Notifications”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2, “Plug-in Overview”</td>
<td>The section “Host Function Tables”</td>
</tr>
</tbody>
</table>

### Replacing Built-In Methods in HFTs

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 Acrobat or Adobe Reader, or to override file opening behavior. For a list of all the replaceable Acrobat and Adobe Reader methods, see the Acrobat and PDF Library API Overview.

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 Acrobat and Adobe Reader’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 Acrobat or Adobe Reader’s `AVAlert` method, it should not prevent other plug-ins from also replacing `AVAlert`.

All plug-ins, and Acrobat or Adobe Reader, share a single copy of each HFT. As a result, when a plug-in replaces a method’s implementation, all other plug-ins and Acrobat or Adobe Reader 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 Acrobat or Adobe Reader.
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 the following figure.

**HFT Entry Replacement**

Replacing a method basically involves:

- Using the macro `ASCallbackCreateReplacement` to create the callback pointer.
- Using the `REPLACE` macro to replace the desired method.

**NOTE:** When you replace a method, you replace it for all plug-ins—not just yours.

**Replacing AVAppCanQuit**

The following example shows how to replace the API method `AVAppCanQuit`. The arguments and return value for `MyAVAppCanQuit` are identical to those of `AVAppCanQuit`. Replaceable methods must be replaced with methods that have the same arguments and return type.

The first statement in Example 12.1 initializes a global pointer `gMyAVAppCanQuitPtr` to your plug-in’s replacement method. You can use this pointer if your plug-in needs it to call the original method it is replacing. For instance, a plug-in may want to add its own functionality and then call the original method that it replaced. To call the original method, use the macro `CALL_REPLACED_PROC`, passing the pointer to your replacement method. See the Acrobat and PDF Library API Reference for more information on this macro.
EXAMPLE 12.1 Replacing AVAppCanQuit

```c
void* gMyAVAppCanQuitPtr = NULL;
/*
 ** Function that tells app whether it’s OK to quit.
 ** When quitting, only allow exit when all docs are closed.
 */
ACCB1 ASBool ACCB2 MyAVAppCanQuit (void)
{
    if (AVAppGetNumDocs () == 0)
        return true;
    else
        return false;
}

void ReplaceAVAppCanQuit ()
{
    DURING
        /* Create the callback */
        gMyAVAppCanQuitPtr = 
            ASCallbackCreateReplacement (AVAppCanQuitSEL, 
                &MyAVAppCanQuit);

        REPLACE (gAcroViewHFT, AVAppCanQuitSEL, gMyAVAppCanQuitPtr);
    HANDLER
        ShowException ("Trying to replace AVAppCanQuit", ERRORCODE);
    ENDHANDLER
```
Notifications

You can register for a notification when you want your plug-in to be notified of a certain event's occurrence. See "Notifications" in the Acrobat and PDF Library API Reference for an alphabetical list of, and details on, all the notification methods.

To register for a notification, use AVAppRegisterNotification and provide a callback function with the appropriate arguments. Acrobat and Adobe Reader call the callback method when the event occurs.

The arguments for each notification provide your plug-in with information of interest for that particular notification.

Registering For a Notification

Example 12.2 provides the framework for registering for the AVAppDidInitialize notification.

In this example, PluginInit uses the ASCallbackCreateNotification macro to create the callback for MyDidInit. MyDidInit is called when the AVAppDidInitialize notification event takes place (when the application has initialized). Then it calls AVAppRegisterNotification, passing the name of the notification (AVAppDidInitialize) with "NSEL" appended, and the MyDidInit callback. When Acrobat or Adobe Reader has initialized, the MyDidInit is called.

During registration, your plug-in can also specify data that is passed to its notification function when it is called back. In this example, PluginInit simply passes NULL for client data in the call to AVAppRegisterNotification.
EXAMPLE 12.2 Registering for a notification

```c
ACCB1 void ACCB2 MyDidInit (void* data)
{
    /*
    ** The app has initialized fully and you can now start
    ** processing documents.
    */
    ...
}
ACCB1 ASBool ACCB2 PluginInit (void)
{
    DURING
        AVAppRegisterNotification(AVAppDidInitializeNSEL,
                                    gExtensionID,
                                    ASCallbackCreateNotification(AVAppDidInitialize,
                                    &MyDidInit), NULL);
    HANDLER
        ShowException("Trying to set up AVAppDidInitialize
                        notification.", ERRORCODE);
        return false;
    END_HANDLER
    return true;
}
```

Registering For Events

Besides the notifications, there are other events for which your plug-in can register an interest, as shown in Table 12.1.

**Table 12.1: Events for which your plug-in can register**

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IdleProc</td>
<td>Called when Acrobat or Adobe Reader is idle</td>
</tr>
<tr>
<td>PageViewDrawing</td>
<td>Called after a PageView is drawn</td>
</tr>
<tr>
<td>PageViewClicks</td>
<td>Called when the mouse is clicked in the AVPageView</td>
</tr>
<tr>
<td>PageViewAdjustCursor</td>
<td>Called to change the cursor when the mouse moves</td>
</tr>
</tbody>
</table>

For each of these events, there is an AVAppRegisterXXX method to tell Acrobat or Adobe Reader your plug-in wants to be called back for the event. A corresponding AVAppUnregisterXXX method tells the application that your plug-in no longer wants to be called.
Registering For AVPageView Click Events

Example 12.1 shows how to register for AVPageView click events.

**EXAMPLE 12.1 Registering for AVPageView click events**

```c
AVPageViewClickProc gMyProcPtr = NULL;

/*
** Return true to process the click or false to pass the click
** to the application for processing.
*/
ACCB1 ASBool ACCB2 MyClickProc (AVPageView pv, ASInt16 x, ASInt16 y,
      ASInt16 Flags, ASInt16 ClickNo, void* data)
{
    AVSysBeep (0);
    return false;
}

void RegisterClickProc (void)
{
    DURING
        gMyProcPtr = ASCallbackCreateProto (AVPageViewClickProc,
          &MyClickProc);
    HANDLER
        ShowException ("Creating callback.", ERRORCODE);
        return;
    END_HANDLER

    DURING
        AVAppRegisterForPageViewClicks (gMyProcPtr, NULL);
    HANDLER
        ShowException ("Registering for page view clicks.",
          ERRORCODE);
    END_HANDLER
}
Exporting and Importing Host Function Tables

A Host Function Table (HFT) is a table containing pointers to methods. Acrobat and Adobe Reader have HFTs containing pointers to all the core API methods. In addition, a plug-in may create its own HFT to export its methods to other plug-ins.

This chapter illustrates how your plug-in can create an HFT and export it, as well as importing other HFTs.

For more information, see the following:

<table>
<thead>
<tr>
<th>Acrobat and PDF Library API Overview</th>
<th>The sections “Replacing Methods” and “Notifications”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2, “Plug-in Overview”</td>
<td>The section “Host Function Tables”</td>
</tr>
</tbody>
</table>

Exporting an HFT

The code in this section shows how to export an HFT consisting of three simple methods: **BeepOnce**, **BeepTwice**, and **BeepNTimes**. Example 13.1 shows the full listing of the header file *MyHFT.h* and Example 13.2 shows the code file *MyHFT.c*. The following sections describe what is going on.

The Exported Methods

This code from *MyHFT.c* is the implementation of the methods that will be exported.

```c
ACCB1 void ACCB2 BeepOnceImplementation ()
{
    AVSysBeep (0);
    AVAlertNote ("In BeepOnceImplementation function.");
}

ACCB1 void ACCB2 BeepTwiceImplementation ()
{
    AVSysBeep (0);
    AVSysBeep (0);
    AVAlertNote ("In BeepTwiceImplementation function.");
}

ACCB1 void ACCB2 BeepNTimesImplementation (ASInt32 numtimes)
{
    ASInt32 i;
    for (i=0; i < numtimes; i++)
        AVSysBeep (0);
}```
Accessing the Methods

When you call a method in an HFT, or when another plug-in calls a method in your HFT, the methods are accessed through a function pointer. The header files are set up to make this transparent to the user; however, this makes the headers a bit difficult to understand. The header for the example, MyHFT.h, contains the following declarations:

- This enumeration defines the constants that allow indexing into the HFT.
  ```
  enum
  {
    DUMMYBLANKSELECTOR, /* 0 */
    BeepOnceSEL,  /* 1 */
    BeepTwiceSEL,  /* 2 */
    BeepNTimesSEL,  /* 3 */
    NUMSELECTORSPlusOne /* 4 */
  };
  ```
  The indexes are called selectors, hence the SEL at the end of the method names. **BeepOnce** is at index 1; **BeepTwice**, at index 2; and **BeepNTimes**, at index 3.

- This global variable references the HFT for your plug-in:
  ```
  extern HFT gMyHFT;
  ```

- This indicates the number of indexes in the HFT:
  ```
  #define NUMSELECTORS (NUMSELECTORSPlusOne - 1)
  ```

HFT Method Definitions

**Define function prototypes**

The following statement from MyHFT.h defines a function pointer for the **BeepNTimes** method.

```
typedef ACCBPROTO1 void (ACCBPROTO2 *BeepNTimesSELPROTO)(ASInt32 numtimes);
```
In general, to declare function prototypes in your HFT, use the following format:

```c
typedef ACCBPROTO1 return_type (ACCBPROTO2 *function_nameSELPROTO) (parameters);
```

where

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return_type</td>
<td>is the return type of the HFT method</td>
</tr>
<tr>
<td>function_name</td>
<td>is the name of the HFT method</td>
</tr>
<tr>
<td>parameters</td>
<td>are the HFT method’s parameters with their types</td>
</tr>
</tbody>
</table>

**Define a name by which to reference each function**

The next statement in the example defines a name to reference the function:

```c
#define BeepNTimes (*((BeepNTimesSELPROTO)(gMyHFT[BeepNTimesSEL])))
```

This macro defines the symbol `BeepNTimes`, which is what users of your HFT will call.

- `gMyHFT[BeepNTimesSEL]` is the function pointer obtained by indexing into the HFT.
- `BeepNTimesSELPROTO` casts the pointer to the right type.

The end result is that users of the method can simply call, for example:

```c
BeepNTimes(3);
```

This is why your implementation of `BeepNTimes` must have a different name (such as `BeepNTimesImplementation`). If the names were the same, there would be a conflict between the `#define` and the implemented method.

In general, to define methods in your HFT, use the following format:

```c
#define method_name (*((method_nameSELPROTO) (HFTname [method_nameSEL])))
```

where

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method_name</td>
<td>is the name of the HFT method</td>
</tr>
<tr>
<td>HFTname</td>
<td>is the name of the HFT variable</td>
</tr>
</tbody>
</table>
The Callbacks

Recall from Chapter 3, “Basic Plug-in Components”, that there are four callbacks a plug-in can provide as part of its handshaking. These callbacks allow your plug-in to:

- Export an HFT (PluginExportHFTs)
- Import an HFT (PluginImportReplaceAndRegister)
- Initialize itself (PluginInit)
- Unload itself (PluginUnload)

The initialize and unload callbacks have already been discussed in Chapter 3. This chapter discusses PluginExportHFTs and PluginImportReplaceAndRegister.

In Example 3.1, the PluginExportHFTs callback in Starter.cpp did nothing. The following example shows a real implementation.

```c
ACCB1 ASBool ACCB2 PluginExportHFTs (void)
{
    gMyHFT = NULL;
    gMyHFTServer = NULL;

    DURING
        provideMyHFTCallback = ASCallbackCreateProto
            (HFTServerProvideHFTProc, &ProvideMyHFT);
        /* Create an HFT server */
        gMyHFTServer = HFTServerNew("MyHFT", provideMyHFTCallback,
            NULL, NULL);
    HANDLER
        gSomethingWentWrong = 1;
        return false;
    END_HANDLER
    return true;
}
```

This is the first handshaking callback that Acrobat or Adobe Reader calls. It calls HFTServerNew to create a new HFT server, specifying a callback that will create the HFT. ProvideMyHFT is the routine that will actually create the HFT; you must use ASCallbackCreateProto to create the callback to pass to HFTServerNew.

In addition, the following code is added to the initialization callback to check whether any problems occurred while creating the HFT server.

```c
ACCB1 ASBool ACCB2 PluginInit(void)
{
    ...
    if (gMyHFTServer && !gSomethingWentWrong)
        return true;
    else
        return false;
} /* MyHFTInit */
```
The following routine, **ProvideMyHFT**, is the callback that actually creates the HFT. It gets called by Acrobat or Adobe Reader when another plug-in tries to import the HFT (see “Importing an HFT” on page 118) by calling `ASExtensionMgrGetHFT`.

```c
ACCB1 HFT ACCB2 ProvideMyHFT (HFTServer server, ASUns32 version,
                    void *rock)
{
   if (version != 1)
      return NULL;
   if (!gMyHFT)
      CreateMyHFT();
   return gMyHFT;
}/* ProvideMyHFT */
```

**ProvideMyHFT** checks whether the HFT has been created already (if `gMyHFT` is not zero) and if not, calls `CreateMyHFT` to create the HFT.

```c
*/
** Create a new HFT of NUMSELECTORS entries.
** Then put the functions into the table via HFTReplaceEntry.
*/
void CreateMyHFT(void)
{
   DURING
      gMyHFT = HFTNew(gMyHFTServer, NUMSELECTORS);
      if (gMyHFT == NULL) {
         gSomethingWentWrong = 1;
         E_RTRN_VOID;
      }
      /*
      ** When creating a new HFT, replace the entries in the HFT
      ** with the functions that you want to be called.
      */
      HFTReplaceEntry (gMyHFT, BeepOnceSEL,
                       ASCallbackCreateReplacement(BeepOnceSEL,
                                                   &BeepOnceImplementation), 0);
      HFTReplaceEntry (gMyHFT, BeepTwiceSEL,
                       ASCallbackCreateReplacement(BeepTwiceSEL,
                                                   &BeepTwiceImplementation), 0);
      HFTReplaceEntry (gMyHFT, BeepNTimesSEL,
                       ASCallbackCreateReplacement(BeepNTimesSEL,
                                                   &BeepNTimesImplementation), 0);
   HANDLER
      gSomethingWentWrong = 1;
      return;
   END_HANDLER
} /* CreateMyHFT */
```

**CreateMyHFT** creates the HFT by calling `HFTNew` and passing the server that was created and the number of entries in the HFT. Then it calls `HFTReplaceEntry` once for each entry, passing the address of the method that implements the entry.
The Header File

Example 13.1 shows the MyHFT.h header file, which defines the prototypes for the methods the plug-in will export.

Example 13.1  Defining prototypes for exportable methods

/* MYHFT.H */
#include "corcalls.h"
#include "avcalls.h"
#include "coscalls.h"
#include "pdcalls.h"
#include "ascalls.h"

enum
{
    DUMMYBLANKSELECTOR,
    BeepOnceSEL,
    BeepTwiceSEL,
    BeepNTimesSEL,
    NUMSELECTORSPlusOne
};

extern HFT gMyHFT;

#define NUMSELECTORS (NUMSELECTORSPlusOne - 1)
typedef ACCBPROTO1 void (ACCBPROTO2 *BeepOnceSELPROTO)(void);
#define BeepOnce (*((BeepOnceSELPROTO)(gMyHFT[BeepOnceSEL])))

typedef ACCBPROTO1 void (ACCBPROTO2 *BeepTwiceSELPROTO)(void);
#define BeepTwice (*((BeepTwiceSELPROTO)(gMyHFT[BeepTwiceSEL])))

typedef ACCBPROTO1 void (ACCBPROTO2 *BeepNTimesSELPROTO)(ASInt32 numtimes);
#define BeepNTimes (*((BeepNTimesSELPROTO)(gMyHFT[BeepNTimesSEL])))

/* End of MyHFT.h */
Exporting and Importing Host Function Tables

Exporting an HFT

The Source File

Example 13.2 shows the code for MyHFT.c, which implements the exportable methods and exports the HFT.

**Example 13.2 Implementing the exportable methods**

```c
/* MyHFT.c */
#include "corcalls.h"
#include "avcalls.h"
#include "coscalls.h"
#include "pdcalls.h"
#include "ascalls.h"
#include "myhft.h"

HFTServer gMyHFTServer;
HFT gMyHFT;
HFTServerProvideHFTProc provideMyHFTCallback;
ASInt32 gSomethingWentWrong = 0;

/*
 ** The implementation for the BeepOnce() function. Note it
 ** has a different name than the #define for the function
 ** in MyHFT.h
 */
ACCB1 void ACCB2 BeepOnceImplementation ()
{
    AVSysBeep (0);
    AVAlertNote ("In BeepOnceImplementation function.");
}

/* The implementation for the BeepTwice() function. Note it has a
 ** different name than the #define for the function in MyHFT.h
 */
ACCB1 void ACCB2 BeepTwiceImplementation ()
{
    AVSysBeep (0);
    AVSysBeep (0);
    AVAlertNote ("In BeepTwiceImplementation function.");
}

/* The implementation for the BeepNTimes() function. Note it has a
 ** different name than the #define for the function in MyHFT.h
 */
ACCB1 void ACCB2 BeepNTimesImplementation (ASInt32 numtimes)
{
    ASInt32 i;
    for (i=0; i < numtimes; i++)
        AVSysBeep (0);
    AVAlertNote ("In BeepNTimesImplementation function.");
}
```
/*
** Create a new HFT of NUMSELECTORS entries.
** Then put the functions into the table via HFTReplaceEntry.
*/
void CreateMyHFT(void)
{
    DURING
        gMyHFT = HFTNew(gMyHFTServer, NUMSELECTORS);
        if (gMyHFT == NULL) {
            gSomethingWentWrong = 1;
            E_RTRN_VOID;
        }
    /*
    ** When creating a new HFT, replace the entries in the HFT
    ** with the functions that you want to be called.
    */
    HFTReplaceEntry (gMyHFT, BeepOnceSEL,
                    ASCallbackCreateReplacement(BeepOnceSEL, &BeepOnceImplementation), 0);
    HFTReplaceEntry (gMyHFT, BeepTwiceSEL,
                    ASCallbackCreateReplacement(BeepTwiceSEL, &BeepTwiceImplementation), 0);
    HFTReplaceEntry (gMyHFT, BeepNTimesSEL,
                    ASCallbackCreateReplacement(BeepNTimesSEL, &BeepNTimesImplementation), 0);
    HANDLER
        gSomethingWentWrong = 1;
        return;
    END_HANDLER
} /* CreateMyHFT */

/*
** Called from the application to return the HFT being provided.
*/
ACCB1 HFT ACCB2 ProvideMyHFT (HFTServer server, ASUns32 version, void *rock)
{
    if (version != 1)
        return NULL;
    if (!gMyHFT)
        CreateMyHFT();
    return gMyHFT;
} /* ProvideMyHFT */

/*
** Called by viewer to set up for exporting an HFT. This method
** creates a new HFT server and provides a callback that
** provides the HFT.
*/
ACCB1 ASBool ACCB2 PluginExportHFTs (void)
{
    gMyHFT = NULL;
gMyHFTServer = NULL;

DURING
provideMyHFTCallback =
    ASCallbackCreateProto(HFTServerProvideHFTProc,
        &ProvideMyHFT);
/* Create an HFT server */
gMyHFTServer =
    HFTServerNew("MyHFT", provideMyHFTCallback,
        NULL, NULL);

HANDLER
    gSomethingWentWrong = 1;
    return false;
END_HANDLER

    return true;
} /* ExportMyHFT */

/*
** The init proc to return true if the HFTServer is OK
** and nothing went wrong.
*/
ACCB1 ASBool ACCB2 PluginInit(void)
{
    if (gMyHFTServer && !gSomethingWentWrong)
        return true;
    else
        return false;
} /* PluginInit */
Importing an HFT

This example shows how a plug-in imports an HFT.

**EXAMPLE 13.3  Importing an HFT**

```c
#include "CorCalls.h"
#include "AVCalls.h"
#include "CosCalls.h"
#include "pdcalls.h"
#include "AsCalls.h"
#include "myhft.h"

/* Make use of the HFT methods */
ACCB1 ASBool ACCB2 PluginInit (void)
{
    BeepOnce ();
    BeepTwice ();
    BeepNTimes (5);
    return true;
} /* MyInit */

/*
** This method gets the HFT from the AS extension manager.
** Set ImportMyHFT as the function for the PluginImportReplaceAnd-
** RegisterCallback field in PIHandshake in PIMain.c.
*/
ACCB1 ASBool ACCB2 PluginImportReplaceAndRegister (void)
{
    gMyHFT = ASExtensionMgrGetHFT(ASAtomFromString("MyHFT"), 1);
    return (gMyHFT != NULL);
} /* ImportMyHFT */

/*
** Get the HFT from the ASExtensionMgr if a plug-in has exported it.
*/
gMyHFT = ASExtensionMgrGetHFT(ASAtomFromString("MyHFT"), 1);
return (gMyHFT != NULL);

**NOTE:** For exporting and importing to work, both plug-ins must be present in Acrobat or Adobe Reader's Plug-ins directory. If the exporting plug-in is not present, when the importing plug-in attempts to get the HFT, the HFT will not be there and the importing plug-in cannot use the methods.
Cos Objects and Methods

A PDF file is structured as a tree of low-level objects, called Cos objects. Cos objects underlie all document components, such as bookmarks, pages, fonts, and annotations. The core API has a set of methods (the Cos layer) to operate directly on objects at this level.

As you have seen in this document, there are several PD layer and PDFEdit methods that you can use to modify the contents of a PDF file. There are certain cases, however, where you may want to do something that is not specifically provided for in the API. In such cases, it may be necessary to use Cos layer methods.

For example, Chapter 10, “Annotations,” showed how to set properties of text annotations using PDTextAnnot methods. Some newer types of annotations may have properties that cannot be accessed directly by PD-layer methods. In such cases, you must directly access the PDF dictionary representing the annotation, by using Cos layer methods. Care is required, because unlike using the AV and PD layer methods, using Cos methods can lead to the production of an invalid PDF file.

This chapter describes how to use Cos-level objects in your plug-ins.

**NOTE:** This chapter is more complex than the other chapters in this document, and does not have as much explanatory text.

More more information, see the following:

<table>
<thead>
<tr>
<th>Acrobat and PDF Library API Overview</th>
<th>The chapter “Cos Layer”</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDF Reference, fifth edition, version 1.6</td>
<td>Chapter 2, “Overview”</td>
</tr>
<tr>
<td></td>
<td>Chapter 3,”Syntax,” through Section 3.8.4, “Rectangles”</td>
</tr>
<tr>
<td></td>
<td>Additional chapters, as necessary</td>
</tr>
</tbody>
</table>
Cos Objects and Methods
Cos Objects in PDF Files

PDF files contain several types of Cos objects. In addition to basic data types such as integer, fixed, and Boolean values, and the null object, they also contain the following types described in this section:

- Array
- Dictionary
- Name
- String
- Stream

These objects are similar to those in the PostScript language. You can destroy existing objects, create new objects, or modify the data in objects in PDF files.

This chapter describes these Cos objects and provides examples of how to modify PDF files at the Cos-object level.

Direct and Indirect Objects

Cos objects in PDF files can either be direct or indirect:

- A direct object is placed directly into another object (such as an array or dictionary). Direct objects cannot be shared between two or more dictionaries or arrays.
- An indirect object is an object that has been labeled so that it can be referenced by other objects, any number of times. Its syntax is shown here.

```
<object number> <generation number> obj
<direct object>
endobj
```

<object number> <generation number> is known as the object ID for an indirect object.

An object referencing another indirect object uses the following syntax:

```
<object number> <generation number> R
```

This reference is equivalent to the direct object represented by the indirect object.

This example shows indirect object 6, followed by a reference to it in indirect object 7.

```
6 0 obj
(This is a string)
endobj

7 0 obj
[ 6 0 R ] %An array with one element that is indirect object 6
endobj
```
If you were to retrieve the zeroth element in the array represented by indirect object 7, you would get the Cos object that represents this string:

(This is a string)

On the other hand, in the following definition of indirect object 8, the elements of the array are all direct objects (the integer objects, 1, 2, and 3).

8 0 obj
[1 2 3]
endobj

Cos Object Types

This section describes the different types of Cos objects. The following sections provide code samples showing how to work with them.

There are two API objects in the Cos layer:

- **CosDoc**, which represents an entire PDF file.
- **CosObj**, which represents all the individual object types described in this section.

There are various methods to create the different types of Cos objects mentioned in this section, as well as getting and setting their values.

Strings

A string is a sequence of characters enclosed in parentheses. For example:

(This is a string)

Arrays

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 following array has 7 elements: 3 integers, a string, a Boolean, a dictionary (containing one key-value pair), and an indirect object reference.

[ 1 2 3 (This is a string) true << /Key (The value) >> 6 0 R ]

Names

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
Dictionaries

A 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.

You can add new key-value pairs, modify existing key-value pairs, or delete existing key-value pairs in a dictionary.

This is an example of a dictionary.

```
<< /Name (Bubba) /Age 27 /AnArray [1 2 3] >>
```

The value associated with the key **Name** is the string *(Bubba)*. The value for the key **Age** is 27. And the value for the key **AnArray** is an array with the contents 1, 2, and 3.

Streams

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 created by a plug-in. A stream consists of these elements, which are listed in their relative order in the stream object, starting at the beginning:

- A dictionary describing the stream's character sequence
- The keyword **stream**
- One or more lines of characters
- The terminating keyword **endstream**

Streams can be encoded in various ways. See the *PDF Reference, fifth edition, version 1.6*, for details on streams.
Strings

Creating a New String

To create a new string, you can use the framework shown in Example 14.1.

**EXAMPLE 14.1 Creating a new string**

```c
void MakeNewString (PDDoc d)
{
    CosObj StrObj;
    CosDoc cd = PDDocGetCosDoc (d);
    char* mystr = "New String";
    DURING
    StrObj = CosNewString (cd, false, mystr, strlen (mystr));
    HANDLER
    AVAlertNote ("Oops.");
    END_HANDLER
}
/* MakeNewString */
```

Getting the String From the Cos Object

The following code fragment gets the string back from the Cos object.

```c
char* TheStr;
ASInt32 length;

TheStr = CosStringValue (StrObj, length);
```

Arrays

Creating a Simple Direct Array

This example creates a simple direct array: [ 1 2 3 ]

**EXAMPLE 14.2 Creating the array**

```c
CosObj MakeAndFillArray (PDDoc d);
{
    CosObj ArrayObj, IntObj;
    CosDoc cd;
    ASInt32 i;
    DURING
    /* You need to get the CosDoc in order to manipulate CosObjects. */
    cd = PDDocGetCosDoc (d);
    /* Create a new 3-element array */
```
ArrayObj = CosNewArray (cd, false, 3); /* create direct object */
/* Now fill the array with the numbers 1, 2, and 3. */
for (i=1; i<=3; i++)
{
  /* Create a new CosObj representing the integer value. */
  IntObj = CosNewInteger (cd, false, i);
  /* Store the integer object in the array. */
  CosArrayPut (ArrayObj, i-1, IntObj);
}
HANDLER
  AVAlertNote ("Oops.");
  return CosNewNull (); /* return null object */
END_HANDLER
return ArrayObj;
} /* MakeAndFillArray */

The **CosNewArray** method creates an array that expands as needed; the initial number of entries is an argument.

**Retrieving Array Values**

**Example 14.3** provides the framework for retrieving the array values.

**EXAMPLE 14.3 Retrieving array values**

```c
void GetArrayValues (CosObj TheArray)
{
  CosObj IntObj;
  ASInt32 value, i, NumElements;
  DURING
    NumElements = CosArrayLength (TheArray);
    for (i=0; i < NumElements; i++)
    {
      /* Grab the CosObj representing the integer from the array */
      IntObj = CosArrayGet (TheArray, i);
      /* Convert the CosObj retrieved to its ASInt32 value. */
      value = CosIntegerValue (IntObj);
      /* Do whatever to the value. */
      .
      .
      .
    }
  HANDLER
    AVAlertNote ("Oops.");
  END_HANDLER
}
/* GetArrayValues */
```
Dictionaries

Creating a Simple Direct Dictionary

This example creates a simple direct dictionary: `<< /Key1 1 /Key2 2 >>`

**EXAMPLE 14.4 Creating the dictionary**

```c
CosObj MakeAndFillDictionary (PDDoc d)
{
    CosObj Dict, IntObj;
    CosDoc cd;

    DURING
        /* Get the CosDoc */
        cd = PDDocGetCosDoc (d);
        /* Make a new 2 entry dict. */
        Dict = CosNewDict (cd, false, 2);
        IntObj = CosNewInteger (cd, false, 1);
        /* Put into the dictionary the key value pair of /Key1 */
        CosDictPut (Dict, ASAtomFromString ("Key1"), IntObj);
        IntObj = CosNewInteger (cd, false, 2);
        /* Put into the dictionary the key value pair of /Key2 */
        CosDictPut (Dict, ASAtomFromString ("Key2"), IntObj);

    HANDLER
        return CosNewNull ();
    END_HANDLER
} /* MakeAndFillDictionary */
```

**CosNewDict** creates a new dictionary that expands as required.

Retrieving Values From the Dictionary

**Example 14.5** retrieves the values from the dictionary created above.

**EXAMPLE 14.5 Retrieving dictionary values**

```c
void GetDictValues (CosObj TheDict)
{
    CosObj IntObj;
    ASInt32 value;
    DURING
        IntObj = CosDictGet (TheDict, ASAtomFromString ("Key1"));
        value = CosIntegerValue (IntObj);

    /* do something with the value */
    .
    .
}
Cos Objects and Methods

Names

Querying the Dictionary For a Key

Frequently you need to inquire whether a particular key-value pair is in a dictionary. To do this, use `CosDictKnown` to query the dictionary for the key of interest. Suppose you have this dictionary, and the key `Optional` is an optional item:

```
<</Name (bubba) /Age 27 /Optional 123 >>
```

To test if the optional item is in the dictionary, you can use this framework:

```
if (CosDictKnown (MyDict, ASAtomFromString ("Optional") == true) {
   ...
}
```

---

Names

Creating a New Name Value

**Example 14.6** shows how to create a new name value.

**EXAMPLE 14.6 Creating a new name value**

```
void MakeNewName (PDDoc d)
{
   CosObj NameObj;
   CosDoc cd = PDDocGetCosDoc (d);
   NameObj = CosNewName (cd, false, ASAtomFromString ("MyName"));
}
```

To get the name value back, do this.

```
char* StringFromName (CosObj NameObj)
{
   ASAtom atom = CosNameValue (NameObj);
   char* str = ASAtomGetString (atom);
}
```
Getting an Annotation’s Subtype

This PDF excerpt is a dictionary representing an annotation. The value of the `Subtype` key is a name object, in this case indicating a link annotation.

```
<<
  /Type /Annot
  /Subtype /Link
  ...
>>
```

The following example shows how to get the subtype from an annotation, given the `Cos` object for the annotation dictionary.

**Example 14.7  Getting the annotation subtype name**

```cpp
void GetSubtypeFromDict (CosObj MyDict)
{
    CosObj name;
    ASAtom atom;

    name = CosDictGet (MyDict, ASAtomFromString("Subtype"));
    atom = CosNameValue (name);
    if (atom == ASAtomFromString("Link"))
    {
        /* You’ve got a link annot. */
        
    }
}
/* GetSubtypeFromDict */
```

### Streams

**Note:** You should study the arguments to `CosNewStream` carefully to understand the examples in this section. For details on `CosNewStream`, see the method description in the *Acrobat and PDF Library API Reference*.

### An Unencoded Page Content Stream

Here is an excerpt from a PDF file showing an unencoded page content stream.

```
8 0 obj
<< /Length 161 >>
stream
BT /F0 1 Tf 12 0 0 12 36 756 Tm 0 Tr 0 g 0 Tc 0 Tw
(This is some text on line 1) Tj
0 -1 TD
(This is line 2.) Tj
0 -1 TD
```

Creating a Stream Using a Memory Buffer

Streams are a bit harder to create than arrays or dictionaries. This section first creates an unencoded stream, then an LZW encoded stream. For these examples, the stream created contains the following data:

1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

Creating an Unencoded Stream

Example 14.8 shows how to create the unencoded stream.

**CosNewStream** accesses data by means of an **ASStm** object. An **ASStm** object is a data stream that you can create in various ways:

- Using a memory buffer, as the examples in this section demonstrate. To create the **ASStm** object for a memory buffer, your plug-in uses the **ASMemStmRdOpen** method, which takes a pointer to the buffer and the length of the buffer.
- Reading a file that contains the data from which your plug-in want to create the stream. Use the **ASFileStmRdOpen** method to do this. This method is illustrated by the example shown in “Creating a Stream From an Existing Data File” on page 133.”
- By using an arbitrary user-written procedure. Use the **ASProcStmRdOpen** method to create the **ASStm** object.
- Calling **CosStreamOpenStm** to create a non-seekable **ASStm** object that may be filtered or encrypted.

See the Acrobat and PDF Library API Reference for details on the above methods. Once your plug-in has called **CosNewStream** to created the new stream, it needs to close the **ASStm** object with **ASStmClose**.

The **MakeNewStream** method in Example 14.8 creates the read-only, seekable stream from a memory buffer using the **ASMemStmOpen** method.

**EXAMPLE 14.8  Creating a stream using a memory buffer**

```c
#define ERRORCODE ASGetExceptionErrorCode()
/* Create and return a stream’s attributes dictionary. */

CosObj CreateAttribsDict(CosDoc Doc, ASInt32 Len)
{
    CosObj Dict; /* Holds newly created dictionary */
    ASAtom Key; /* Key used to retrieve CosObj in dictionary */
    CosObj Value; /* Assigned, then added to dictionary */
    CosObj DecodeArray;
```

DURING
  Dict = CosNewDict(Doc, false, 10);
HANDLER
  ShowException("Trying to make new attributes dictionary.", ERRORCODE);
  return CosNewNull();
END_HANDLER

DURING
  Key = ASAtomFromString("Length");
  Value = CosNewInteger(Doc, false, Len);
  CosDictPut(Dict, Key, Value);
HANDLER
  ShowException("Trying to put length key into attributes dictionary.", ERRORCODE);
  CosObjDestroy(Dict);
  return CosNewNull();
END_HANDLER

return Dict;
} /* CreateAttribsDict */

/*
** MakeNewStream assumes that the stream is the value for the Contents key in
** the page’s dictionary. “Creating a Stream From an Existing Data File” on page 133
** and “Creating a New Document, New Page, and Page Content Stream” on page 135
** cover placement of page
** content streams in PDF files.
*/
void MakeNewStream (PDPage page, ASInt32 StmLen)
{
  CosDoc cd;
  ASStm OpenedStream = NULL;
  CosObj PageStrm;
  ASInt32 FileLen;
  char buf [500];
  CosObj EncodeParms = CosNewNull();

  DURING
    cd = PDDocGetCosDoc(PDPageGetDoc(page));
  HANDLER
    ShowException("Unable to get CosDoc.", ERRORCODE);
    return;
  END_HANDLER

  AttrDict = CreateAttribsDict(cd, StmLen);
  if (CosObjEqual(AttrDict, CosNewNull()) == true)
  {
    AVAlertNote("Not making stream, attributes dict was not created.");
    return;
  }
/* Put our data into a memory buffer. */
sprintf(buf,1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz ");

DURING
/*
** Acrobat has an internal limit of 4096 bytes for the second argument
** of ASMemStmRdOpen and ASFileStmRdOpen.
*/

OpenedStream = ASMemStmRdOpen (buf, 4096);

HANDLER
ShowException ("Trying to open stream from CosStream.",ERRORCODE);
CosObjDestroy (AttrDict);
return;
END_HANDLER
if (!OpenedStream)
{
    AVAlertNote ("Unable to open data stream to create content stream.");
    return;
}

DURING
PageStrm = CosNewStream(cd, true, OpenedStream, -1,
    false, /* StmDataIsNotDecoded */
    AttrDict, /* attributesDict */
    EncodeParms, -1);
ASStmClose (OpenedStream);

HANDLER
ShowException ("Trying to create new CosStream.", ERRORCODE);
CosObjDestroy (AttrDict);
ASStmClose (OpenedStream);
return;
END_HANDLER

PDPageAddCosContents (page, PageStrm);

/*
** Any time you modify a page’s content stream,
** PDPageNotifyContentsDidChange is called automatically
*/

} /* MakeNewStream */
Creating an LZW-Encoded Stream

Example 14.9 is very similar to Example 14.8 in that also uses ASMemStmRdOpen to create a readable, seekable stream from a memory buffer. However, in this case, the CreateAttribsDict method LZW-encodes the stream.

Like Example 14.8, Example 14.9 assumes that the stream is the value for the Contents key in the page’s dictionary. See “Creating a Stream From an Existing Data File” on page 133 for details on how you can place page content streams in PDF files.

**Example 14.9  Creating an LZW-encoded stream**

/* Create the attributes dictionary that describes the stream. 
** CreateAttribsDict creates a dictionary that looks like this. 
**  << 
** /Filter [ /LZWDecode ] 
** >> 
*/

CosObj CreateAttribsDict(CosDoc Doc, char* TheStr)
{
    CosObj Dict; /* Holds newly created dictionary */
    ASAtom Key; /* Key used to retrieve CosObj in dictionary */
    CosObj Value; /* Assigned; then added to dictionary */
    CosObj DecodeArray;
    DURING
        Dict = CosNewDict(Doc, false, 10);
        HANDLER
            ShowException ("Trying to make new attributes dictionary.", ERRORCODE);
            return CosNewNull ();
        END_HANDLER
    DURING
        DecodeArray = CosNewArray (Doc, false, 5);
        HANDLER
            ShowException ("Trying to make new filters array.", ERRORCODE);
            CosObjDestroy (Dict);
            return ();
        END_HANDLER
    DURING
        Value = CosNewName (Doc, false, ASA ("LZWDecode"));
        CosArrayPut (DecodeArray, 0, Value);
        CosDictPut (Dict, ASAtomFromString ("Filter"), DecodeArray);
        HANDLER
            ShowException ("Trying to place LZWDecode into filters dictionary.", ERRORCODE);
            CosObjDestroy (DecodeArray);
            CosObjDestroy (Dict);
            return CosNewNull ();
        END_HANDLER
        return Dict;
void MakeNewStream (PDPage page, Inst32 StmLen)
{
    CosDoc cd;
    ASStm OpenedStream = NULL;
    CosObj PageStrm;
    ASInt32 FileLen;
    char buf [500];
    CosObj EncodeParms = CosNewNull();

    DURING
        cd = PDDocGetCosDoc (PDPageGetDoc (page));
    HANDLER
        ShowException ("Unable to get CosDoc.", ERRORCODE);
        return;
    END_HANDLER

    AttrDict = CreateAttribsDict (cd, StmLen);
    if (CosObjEqual (AttrDict, CosNewNull ()) == true)
    {
        AVAlertNote ("Not making stream, attributes dict
                      was not created.");
        return;
    }
    /* Put our data into a memory buffer. */
    sprintf (buf,"1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz ");

    DURING
        OpenedStream = ASMemStmRdOpen (buf, 4096);
    HANDLER
        ShowException ("Trying to open memory stream to create CosStream.",
                       ERRORCODE);
        CosObjDestroy (AttrDict);
        return;
    END_HANDLER

    if (!OpenedStream)
    {
        AVAlertNote ("Unable to open data stream to create content stream.");
        return;
    }

    DURING
        PageStrm = CosNewStream(cd, true, OpenedStream, -1,
                                true, /* StmDataIsDecoded */
                                AttrDict, /* attributesDict */
                                EncodeParms, -1);
        ASStmClose (OpenedStream);
    HANDLER
        ShowException ("Trying to create new CosStream.",ERRORCODE);
        CosObjDestroy (AttrDict);
Creating a Stream From an Existing Data File

Example 14.10 makes a stream from an existing data file. Suppose you had a file at c:\mydata.txt. Here is how to create a Cos stream object from that data file, assuming you want LZW encoding.

The main differences between this data file example and Example 14.8 and Example 14.9 in “Creating a Stream Using a Memory Buffer” on page 128, are:

- You need to open an ASFile object from a data file.
- You must use ASFileStmRdOpen to generate the needed ASStm object.
- The exception handler does an ASFileClose on the opened ASFile object.

**EXAMPLE 14.10 Creating a stream from an existing data file**

```c
void MakeNewStream (PDPage page, ASInt32 StmLen) {
    CosDoc cd;
    ASStm OpenedStream = NULL;
    CosObj PageStm;
    ASFile TheFile = NULL;
    ASInt32 err = 0;
    CosObj EncodeParms = CosNewNull();
    DURING
        cd = PDDocGetCosDoc (PDPageGetDoc (page));
    HANDLER
        ShowException ("Unable to get CosDoc.", ERRORCODE);
        return;
    END_HANDLER

    AttrDict = CreateAttribsDict (cd, StmLen);
    if (CosObjEqual (AttrDict, CosNewNull ()) == true) {
        AVAlertNote ("Not making stream, attributes dict was not created.");
        return;
    }

    /* Open the needed ASFile for reading. */
    err = ASFileSysOpenFile (ASGetDefaultFileSys (),
```
ASPathFromPlatformPath("c:\\mydata.txt"), ASFILE_READ, &TheFile);
if (err)
{
    AVAlertNote ("Unable to open data file to create page stream.");
    return;
}

/*
** CosNewStream accesses your data via an ASStm. This method to
** create an ASStm uses a file that contains the data from which you want to
** create the stream. You need to use ASFileStmRdOpen.
**
OpenedStream = ASFileStmRdOpen (TheFile, 4096);

if (!OpenedStream)
{
    AVAlertNote ("Unable to open data stream to create content stream.");
    ASFileClose (TheFile);
    return;
}

DURING
    PageStrm = CosNewStream(cd, true, OpenedStream, -1,
        true, /* StmDataIsDecoded */
        AttrDict, /* attributesDict */
        EncodeParms, -1);
    ASStmClose (OpenedStream);
    ASFileClose (TheFile);
HANDLER
    ShowException ("Trying to create new CosStream.",ERRORCODE);
    CosObjDestroy (AttrDict);
    ASStmClose (OpenedStream);
    SFileClose (TheFile);
    return;
END_HANDLER

PDPageAddCosContents (page, PageStrm);

/*
** Any time you modify a page’s content stream,
** PDPageNotifyContentsDidChange is called automatically
*/
} /* MakeNewStream */
Creating a New Document, New Page, and Page Content Stream

This section creates a new document. Then, it creates a new page and creates its page content stream to display the text “Hello World.” First the page objects need to be created.

Dictionaries Containing Needed Page Object Information

This section shows what needs to be added to the page. The PDF is taken from a PDF file. The dictionaries that need to be created are described below.

This is the resource dictionary for the page.

```plaintext
4 0 obj
<<
/Font << /F0 5 0 R >>
/ProcSet 6 0 R
>>
endobj
```

This is the font descriptor.

```plaintext
5 0 obj
<<
/Type /Font
/Subtype /Type1
/Name /F0
/BaseFont /Courier
/Encoding /WinAnsiEncoding
>>
endobj
```

This is a procset resource.

```plaintext
6 0 obj
[
/PDF /Text
]
endobj
```

This is the page's dictionary.

```plaintext
7 0 obj
<<
/Type /Page
/MediaBox [ 0 0 612 792 ]
/Parent 2 0 R
/Resources 4 0 R
/Contents 8 0 R
>>
endobj
```
Creating the PDF Objects

Example 14.11 creates the objects above.

**NOTE:** The following sections partially describe the code in Example 14.11.

The `PluginInit` routine for this example sets up menu items, similarly to the examples in Chapter 4, “Menus and Menu Items”. It adds a **New Page** item to the **Tools** menu, which when called will trigger the `ExeProc` routine, which in turn calls `MakeTheFile`.

**MakeTheFile**

1. Defines the media box rectangle that it will use in creating the first page of the document.
   ```c
   FixedRect MedBox;
   MedBox.left = ASInt32ToFixed (0);
   MedBox.top = ASInt32ToFixed (792);
   MedBox.right = ASInt32ToFixed (612);
   MedBox.bottom = ASInt32ToFixed (0);
   ```

2. Defines the stream to set the text matrix and write out the text:
   ```c
   StreamBuf = (char*)"BT /F0 1 Tf 24 0 0 24 36 756 Tm 0 Tr 0 g 0 Tc 0 Tw \n   (Hello World) Tj ET";
   ```

3. Creates a new document using `PDDocCreate`. The only Cos object in the document is the Catalog. After the document is created, at least one page must be added before Acrobat or Adobe Reader can display the document.
   ```c
   NewDoc = PDDocCreate ();
   ```

4. Calls `CreateResourceDicts` (described in the next section). Note the call to `PDDocGetCosDoc`, which gets the `CosDoc` object corresponding to the `PDDoc` object. The `CosDoc` object is needed for some of the other Cos-level methods.
   ```c
   if (CreateResourceDicts (PDDocGetCosDoc (NewDoc)) == false)
     ASRaise (0);
   ```

5. Creates a page using `PDDocCreatePage`, passing the media box created previously.
   ```c
   NewPage = PDDocCreatePage (NewDoc, PDBeforeFirstPage, MedBox);
   ```

6. Calls `SetResourceForPage` and `AddStreamToPage`, described in the next sections.
   ```c
   result = SetResourceForPage (NewPage, PDDocGetCosDoc (NewDoc));
   ... 
   result = AddStreamToPage (NewPage, StreamBuf, StreamBufLen);
   ```

7. Releases the page.
   ```c
   PDPageRelease (NewPage);
   ```

8. Opens the document for viewing.
   ```c
   AVDocOpenFromPDDoc (NewDoc, NULL);
   ```
CreateResourceDicts

/*
** Create the font’s resources. Return true if all is well, else false.
** Creates the needed font and proc set dicts; then creates the
** global resource dict for the pages
*/

First, this routine creates two new dictionaries in the document. Passing true for the
second parameter means that it is an indirect object, and the third parameter is the initial
number of entries (which is just a hint; dictionaries grow dynamically as needed).

ResDict = CosNewDict (cd, true, 10);
...
FontDictObj = CosNewDict (cd, true, 5);

The dictionary FontDictObj will be the font descriptor. Next, the code adds the
following entries to it:

<<
/Type /Font
/Subtype /Type1
/Name /F0
/BaseFont /Courier
/Encoding /WinAnsiEncoding
>>

Since all the values in this dictionary are name objects (Font, Type1, etc.), the function uses
CosNewName to create them, and passes them to CosDictPut to create the key-value
pairs:

CosDictPut (FontDictObj, ASA ("Type"),
    CosNewName (cd, false, ASA ("Font")));
CosDictPut (FontDictObj, ASA ("Subtype"),
    CosNewName (cd, false, ASA ("Type1")));
...

NOTE: The macro ASA is defined as ASAtomFromString to save space. ASAtoms are
used to represent name objects.

FontDict = CosNewDict (cd, false, 2);

/* Add entries to the page’s resource dictionary.
** <<
** /Font << /F0 5 0 R >>
** /ProcSet 6 0 R
** >>
*/
/* Add /Font key-value pair to resource dictionary */
    CosDictPut (FontDict, ASA ("F0"), FontDictObj);
    CosDictPut (ResDict, ASA ("Font"), FontDict);
/* Create the following proc set resource array.
** [
** /PDF /Text

Acrobat SDK Plug-in Guide
** ] */

This code creates a new array and assigns two entries to it.

```
ArrayThing = CosNewArray (cd, true, 5);
CosArrayPut (ArrayThing, 0, CosNewName (cd, false, ASA("PDF")));
CosArrayPut (ArrayThing, 1, CosNewName (cd, false, ASA("Text")));
/*
** Place the proc set key-value pair into the page’s resource
dictionary.
*/
CosDictPut (ResDict, ASA("ProcSet"), ArrayThing);
```

**The Complete Example**

**EXAMPLE 14.11 Creating a new document, new page, and page content stream**

```c
// #define DOLZW 1

#include "ascalls.h"
#include "avcalls.h"
#include "avcalls.h"
#include "coscalls.h"
#include "pdctells.h"
#include "ascalls.h"
#include "corcalls.h"
#include "dos.h"
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
CosObj PageStrm; /* To hold newly created Cos stream */
CosObj AttrDict; /* As returned by CreateAttribsDict */
CosObj EncodeDict;
/*
** Used to specify what filters are used to encode stream if
** used for output.
*/
CosObj ResDict; /* Resource dictionary for the page */
CosObj FontDictObj;
CosObj FontDict;
CosObj ArrayThing;
#define ASA(a) ASAtomFromString (a)
```
/** Set the page’s resource key. Return true if all is well, **
else false. */
ASBool SetResourceForPage (PDPage page, CosDoc cd)
{
    CosObj PageCosObj;
    if (!page)
        return false;
    DURING
        PageCosObj = PDPageGetCosObj (page);
        if (CosDictKnown (PageCosObj, ASA ("Resources") == true))
            CosObjDestroy (CosDictGet (PageCosObj, ASA ("Resources")));
            CosDictPut (PageCosObj, ASA ("Resources"), ResDict);
    HANDLER
        return false;
    END_HANDLER
    return true;
} /* SetResourceForPage */

/*/  
** Create the font’s resources. Return true if all is well, else false.  
** Creates the needed font and proc set dicts; then creates the  
** global resource dict for the pages  
*/
ASBool CreateResourceDicts (CosDoc cd)
{
    DURING
        ResDict = CosNewDict (cd, true, 10);
    HANDLER
        ShowException ("Trying to create the resource dictionary.",
            ERRORCODE);
        return false;
    END_HANDLER

    DURING
        FontDictObj = CosNewDict (cd, true, 5);
    HANDLER
        ShowException ("Trying to create the font’s dictionary.",
            ERRORCODE);
        CosObjDestroy (ResDict);
        return false;
    END_HANDLER
/* Create this font descriptor dictionary. 
** << 
** /Type /Font 
** /Subtype /Type1 
** /Name /F0 
** /BaseFont /Courier 
** /Encoding /WinAnsiEncoding 
** >> 
*/

DURING
    CosDictPut (FontDictObj, ASA ("Type"),
        CosNewName (cd, false, ASA ("Font")));
    CosDictPut (FontDictObj, ASA ("Subtype"),
        CosNewName (cd, false, ASA ("Type1")));
    CosDictPut (FontDictObj, ASA ("Name"),
        CosNewName (cd, false, ASA ("F0")));
    CosDictPut (FontDictObj, ASA ("BaseFont"),
        CosNewName (cd, false, ASA ("Courier")));
    CosDictPut (FontDictObj, ASA ("Encoding"),
        CosNewName (cd, false, ASA ("WinAnsiEncoding")));

HANDLER
    ShowException ("Trying to add key-value pairs to the Font
descrriptor dictionary.", ERRORCODE);
    CosObjDestroy (FontDictObj);

    return false;

END_HANDLER

DURING
    FontDict = CosNewDict (cd, false, 2);

HANDLER
    ShowException ("Trying to create page’s resource dictionary.", ERRORCODE);
    CosObjDestroy (FontDictObj);

    return false;

END_HANDLER

/* Add entries to the page’s resource dictionary. 
** << 
** /Font << /F0 5 0 R >> 
** /ProcSet 6 0 R 
** >> 
*/

DURING
/* Add /Font key-value pair to resource dictionary */
    CosDictPut (FontDict, ASA ("F0"), FontDictObj);
    CosDictPut (ResDict, ASA ("Font"), FontDict);

HANDLER
    ShowException ("Trying to add key-value pairs to the page’s resource dict.",
        ERRORCODE);
    CosObjDestroy (FontDictObj);
    CosObjDestroy (FontDict);
CosObjDestroy (ResDict);
    return false;
END_HANDLER

/* Create the following proc set resource array. */
** [ */
** /PDF /Text */
DURING
    ArrayThing = CosNewArray (cd, true, 5);
HANDLER
    ShowException ("Trying to create proc set array.", ERRORCODE);
    CosObjDestroy (FontDictObj);
    CosObjDestroy (FontDict);
    CosObjDestroy (ResDict);
    return false;
END_HANDLER

DURING

    CosArrayPut (ArrayThing, 0, CosNewName (cd, false, ASA ("PDF")));
    CosArrayPut (ArrayThing, 1, CosNewName (cd, false, ASA ("Text")));
    /* */
    ** Place the proc set key-value pair into the page’s resource dictionary. */
    CosDictPut (ResDict, ASA ("ProcSet"), ArrayThing);
HANDLER
    CosObjDestroy (FontDictObj);
    CosObjDestroy (FontDict);
    CosObjDestroy (ResDict);
    CosObjDestroy (ArrayThing);
    return false;
END_HANDLER
    return true;
} /* CreateResourceDicts */

/**
 ** Create and return the stream’s attributes dictionary.
 ** You can control whether or not to do LZW encoding by the #define.
 ** If you want to LZW encode, you must uncomment the first line
 ** containing the #define.
 */

CosObj CreateAttribsDict(CosDoc Doc, ASInt32 Len)
{
    CosObj Dict; /* Holds newly created dictionary */
    ASAtom Key; /* Key used to retrieve CosObj in dictionary */
    CosObj Value; /* Assigned, then added to dictionary */
    CosObj DecodeArray;

    DURING
Dict = CosNewDict(Doc, false, 10);
HANDLER
    ShowException ("Trying to make new attributes dictionary.",
        ERRORCODE);
    return CosNewNull ();
END_HANDLER
#endif DOLZW
Key = ASAtomPromString("Length");
DURING
    Value = CosNewInteger(Doc, false, Len);
    CosDictPut(Dict, Key, Value);
HANDLER
    ShowException ("Trying to put length key into
        attributes dictionary.", ERRORCODE);
    CosObjDestroy (Dict);
    return CosNewNull ();
END_HANDLER
#else
DURING
    DecodeArray = CosNewArray (Doc, false, 5);
HANDLER
    ShowException ("Trying to make new filters array.", ERRORCODE);
    CosObjDestroy (Dict);
    return CosNewNull ();
END_HANDLER
DURING
    Value = CosNewName (Doc, false, ASA ("LZWDecode"));
    CosArrayPut (DecodeArray, false, Value);
    CosDictPut (Dict, ASA ("Filter"), DecodeArray);
HANDLER
    ShowException ("Trying to place LZWDecode in filters dict.", ERRORCODE);
    CosObjDestroy (DecodeArray);
    CosObjDestroy (Dict);
    return CosNewNull ();
END_HANDLER
#endif
return Dict;
} /* CreateAttribsDict */

/* Add stream to page. Return false if there were problems. */

ASBool AddStreamToPage (PDPage page, char* StreamBuf, ASInt32 StreamBufLen)
{
    CosDoc cd;
    ASStm Stm=NULL;
    CosObj PageStrm;
    CosObj EncodeParms = CosNewNull();
DURING
  cd = PDDocGetCosDoc (PDPageGetDoc (page));
HANDLER
  ShowException ("Unable to get CosDoc.", ERRORCODE);
  return false;
END_HANDLER
AttrDict = CreateAttribsDict (cd, StreamBufLen);
if (CosObjEqual (AttrDict, CosNewNull ()) == true)
{
  AVAlertNote ("Not making stream. Attribs dict not created.");
  return false;
}
Stm = ASMemStmRdOpen (StreamBuf, StreamBufLen);
if (!Stm)
{
  AVAlertNote ("Unable to open data stream to create content stream.");
  return false;
}

DURING
  PageStrm = CosNewStream (cd, true, Stm, -1,
  #ifdef DOLZW
    true, /* stmDataIsDecoded */
  #else /* StmDataIsNotDecoded */
    false,
  #endif
  AttrDict, /* attributesDict */
    EncodeParms,
  -1);
ASStmClose (Stm);
HANDLER
  ShowException ("Trying to create new CosStream.", ERRORCODE);
  CosObjDestroy (AttrDict);
  ASStmClose (Stm);
  return false;
END_HANDLER

PDPageAddCosContents (page, PageStrm);
return true;
} /* AddStreamToPage */

/* Create the file. */
void MakeTheFile (void)
{
  volatile PDDoc NewDoc = NULL;
  volatile PDPage NewPage;
  FixedRect MedBox;
  ASInt32 PageCount = 0;
  char* StreamBuf = NULL;
  int StreamBufLen = 0;
/* Set up the page’s media box. */
MedBox.left  = ASInt32ToFixed (0);
MedBox.top   = ASInt32ToFixed (792);
MedBox.right = ASInt32ToFixed (612);
MedBox.bottom = ASInt32ToFixed (0);

/* Here is the stream to set the text matrix and write out the text. */
StreamBuf = (char*)"BT /F0 1 Tf 24 0 24 36 756 Tm 0 Tr 0 g 0 Tc 0 Tw \n(Hello World) Tj ET";
StreamBufLen = strlen (StreamBuf);

DURING
  NewDoc = PDDocCreate ();
  if (NewDoc) {
    if (CreateResourceDicts (PDDocGetCosDoc (NewDoc)) == false)
      ASRaise (0);

    NewPage = PDDocCreatePage (NewDoc, PDBeforeFirstPage, MedBox);
    if (!NewPage) ASRaise (0);

    ASBool result;
    result = SetResourceForPage (NewPage, PDDocGetCosDoc (NewDoc));
    if (result == false) ASRaise (0);
    result = AddStreamToPage (NewPage, StreamBuf, StreamBufLen);
    if (result == false) ASRaise (0);
    PDPageRelease (NewPage);
  }

HANDLER
  AVAlertNote ("Problem creating document.");
  if (NewPage) PDPageRelease (NewPage);
  if (NewDoc) PDDocClose (NewDoc);
  return;
END_HANDLER

DURING
  AVDocOpenFromPDDoc (NewDoc, NULL);
HANDLER
  AVAlertNote ("Cannot open new document.");
END_HANDLER

} /* MakeTheFile */
/* Main program */
ACCB1 void ACCB2 ExeProc (void* s)
{
    MakeTheFile ();
}

ACCB1 ASBool ACCB2 PluginInit(void)
{
    AVMenuBar MenuBar= (AVMenuBar)NULL;
    AVMenu tools = (AVMenu)NULL, About = (AVMenu)NULL,
        SubMenu = (AVMenu)NULL;
    AVMenuItem Item = (AVMenuItem)NULL, AboutItem = (AVMenuItem)NULL,
        Item12Pt = (AVMenuItem)NULL;
    char* ptr=(char*)NULL;
    MenuBar = AVAppGetMenubar();
    tools = AVMenuBarAcquireMenuByName (MenuBar, "Tools");

    DURING
    if (tools)
    {
            (AVMenu)NULL, false, (char)NO_SHORTCUT, 0,
            (void*)0, gExtensionID);
        if (!Item)
            ASRaise (0);
        AVMenuItemSetExecuteProc (Item,
            ASCallbackCreateProto (AVExecuteProc, &ExeProc), NULL);
        AVMenuAddMenuItem (tools, Item,
            AVMenuGetNumMenuItems (tools));
        AVMenuRelease (tools);
    }

    HANDLER
    AVAlertNote ("Unable to set up menu item, "New Page"
        plug-in is not loaded.");
    return false;
    END_HANDLER
} /* PluginInit */
Reading a Stream's Data

Example 14.12 shows how to read a stream's data.

Assuming the code in Example 14.11 is used to create a document, this example reads the stream back in.

**Example 14.12 Reading a stream's file**

```c
void OkReadStream (CosObj Stream)
{
    ASStm Stm;
    char StreamBuf [1000];
    ASInt32 NumBytesToRead = 1000;

    DURING
        Stm = CosStreamOpenStm (Stream, cosOpenFiltered);
        /*
        ** Read in 1000 byte chunks. ASStmRead returns 0 when
        ** there is nothing more to read.
        */
        while (ASStmRead (StreamBuf, 1, NumBytesToRead, Stm) != 0)
        {
            // Act on data or store data somewhere.
        }
        ASStmClose (Stm);
    HANDLER
        AVAlertNote ("Oops.");
        ASStmClose (Stm);
    END_HANDLER
}
```

```c
void ReadStream (PDPage p)
{
    CosObj PageObj;
    CosObj StreamObj;

    DURING
        PageObj = PDPageGetCosObj (p);
        if (CosDictKnown (PageObj, ASAtomFromString ("Contents")) == true)
        {
            StreamObj = CosDictGet (PageObj, ASAtomFromString ("Contents"));
            OkReadStream (StreamObj);
        }
    HANDLER
        AVAlertNote ("Cannot read stream.");
    END_HANDLER
} /* OkReadStream */
```
Page Content Streams

Page content streams come in the various flavors shown below.

Here is a reference to a stream.

```
4 0 obj
<<
/Type /Page
/Parent 51 0 R
/Resources << /Font << /F0 6 0 R /F1 9 0 R >> /ProcSet 2 0 R >>
/Contents 7 0 R
>>
endobj
```

Contents is an array of content streams.

If you were writing a plug-in to modify or parse existing page content streams, you would need to account for the different ways to access the stream content to read it. Suppose you needed to write a plug-in that turns all black text and graphic objects to the color blue. You have to parse the page content streams of PDF files and replace

```
0 0 0 rg
```

with

```
0 0 1 rg
```
Other Plug-in Topics

This chapter gives general platform-independent information about other topics that are useful for Acrobat or Adobe Reader plug-in development, including:

- Adding Message Handling
- Creating Thumbnail Images

Adding Message Handling

Plug-ins can add their own Apple events and DDE messages to those supported by Acrobat and Adobe Reader. 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 or Adobe Reader's Apple event handling loop to handle Apple events. To do this, replace the API's `AVAppHandleAppleEvent` method (see "Replacing Built-In Methods in HFTs" on page 103). 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 Acrobat or Adobe Reader the opportunity to handle the Apple event.

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

Creating Thumbnail Images

You can use a direct or an indexed color space when creating thumbnail images using `PDDocCreateThumbs`.

If desired, the following code can be used to create an indexed color space's lookup table that is passed as the `lookupTable` parameter to `PDDocCreateThumbs`:

```c
/* IN: Pass an array of ASUns8 containing 256*3 entries. The entry’s values will be overwritten. 
OUT: The array filled with RGB entries. */

void MakeColorTable(ASUns8 *lookupTable)
{
    ASUns8 red, green, blue;
    int redCount, greenCount, blueCount;
    int i;
    ASUns8 *p = lookupTable;
```
/* Populate a 6x6x6 entry RGB cube */
red = 0;
for ( redCount = 7; --redCount; ){
green = 0;
for ( greenCount = 7; --greenCount; ){
blue = 0;
for ( blueCount = 7; --blueCount; ){
  *p++ = red;
  *p++ = green;
  *p++ = blue;
  blue += 0x33;
}
green += 0x33;
}red += 0x33;
}

/* Add 10 additional grays */
red = 0;
for ( redCount = 0; redCount < 10; redCount++ ){
  /* skip this one because it’s already in the 6x6x6 cube */
  if ( (redCount & 1) == 0 )
    red += 0x11;
    *p++ = red;
    *p++ = red;
    *p++ = red;
    red += 0x11;
}

/* Fill the rest of the table with white */
for ( i = 226; i < 256; i++ ) {
  *p++ = 255;
  *p++ = 255;
  *p++ = 255;
}
Upgrading Plug-ins From Acrobat 6 to Acrobat 7

This appendix provides information on how to upgrade your plug-in from Acrobat 6 to Acrobat 7. It includes the following sections:

- General Information
- Upgrading Plug-ins on Windows
- Upgrading Plug-ins on a Macintosh

General Information

This section provides general platform-independent guidelines for upgrading Acrobat or Adobe Reader plug-ins to Acrobat 7.

Backward Compatibility

For Windows, plug-ins written for a version of Acrobat previous to Acrobat 7 are, in most cases, binary compatible with Acrobat 7—that is, pre-Acrobat 7 plug-ins will usually work “as is” with Acrobat 7. For Macintosh, plug-ins developed in previous SDK versions must be rebuilt using Mach-O.

Developers must test existing plug-ins for compatibility with Acrobat 7. If they don’t work, they may need to be rebuilt with the new headers in the Acrobat 7 SDK.

The code in PIRequir.h controls which version of an HFT loads. The PIRequir.h states the minimum version of the HFT to load. Thus, if you want to run on Acrobat 6.0, none of the HFT version numbers should be higher than 0x00060000 (because Acrobat 6.0 cannot supply an HFT with all the functions available in Acrobat 7.0) unless that HFT is optional for the plug-in. Put another way, if a plug-in is built to run on Acrobat 6 but use Acrobat 7 features when they are available, the minimum version required is 0x00060000. If the plug-in asks for 0x00060000 and receives 0x00060000, it will work successfully. For more information on working with HFTs in Acrobat 7 plug-ins, see Chapter 13, “Exporting and Importing Host Function Tables”.

Adobe engineering has provided several important mechanisms to enable backward compatibility in Acrobat 7 plug-ins with earlier versions of Acrobat.

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 header files, AVExpTObsolete1.h and AVExpTObsolete2.h. These files contain types used in former versions of Acrobat or Adobe Reader. By defining ACRO_SDK_LEVEL to 0x00070000, 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 or Adobe Reader. 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 or Adobe Reader 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 or Adobe Reader. It is set to 0xFFFFFFFF by default, which is the value required for shipping Acrobat 7 plug-ins. However, the macro can be set to previous versions of Acrobat or Adobe Reader to simulate those execution environments. For example, it can be set to 0x00060000 to simulate Acrobat 6.0.

**NOTE:** While the simulation approximates earlier Acrobat or Adobe Reader execution environments, it is not the same as the real thing. A plug-in should still be tested against the actual products before shipping.

**Development Header Files Changes for Acrobat 7**

The directory structure of the header files has changed in Acrobat 7. The following subdirectories are now located under the Headers directory:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDK</td>
<td>Contains header files that are common to most plug-ins and are generally referenced from PIMain.c. See “Acrobat SDK Header Files” on page 177</td>
</tr>
<tr>
<td>API</td>
<td>Contains API header files specific for core and extended APIs. See “API Header Files” on page 178.</td>
</tr>
<tr>
<td>ADM</td>
<td>Contains header files for use with the Adobe Dialog Manager (ADM). See “ADM Header Files” on page 183.</td>
</tr>
</tbody>
</table>

See “Upgrading Plug-ins on Windows” on page 158 and “Upgrading Plug-ins on a Macintosh” on page 159 for specific changes required for your development platform.
Detecting Supported APIs on Acrobat Professional, Acrobat Standard, and Adobe Reader

Acrobat Professional supports the full set of APIs. If you try to use an API that is not supported on a particular Acrobat configuration, nothing will happen. The same HFT version numbers are used across configurations, so all APIs are callable on all configurations, but certain APIs simply won’t work on certain configurations.

To address this issue, the Acrobat and PDF Library API Overview has been enhanced to provide a table that specifies which APIs work with which products. Additionally, the Extended APIs provided by plug-ins won’t work if the Acrobat configuration does not support the use of those APIs. The HFTs will not load, so you need to check whether the HFT was successfully imported. See “Which APIs Work With Which Products?” on page 20 for details of which Extended APIs work with which configurations.

It is possible to determine in your code whether the HFT you are expecting is in fact the one that you are importing, and whether it imported at all: simply check for a NULL return value. For example, a NULL will be returned in the following call if AcroColorHFTNAME with the specified version is not available:

```c
    gAcroColorHFT = ASExtensionMgrGetHFT(ASAtomFromString(AcroColorHFTNAME), PI_ACROCOLOR_VERSION);
```

Plug-ins that use new HFTs introduced with Acrobat 7.0 will not run on Acrobat versions prior to 7.0. Whether or not an attempt to load these forces the plug-in to fail is controlled by flags in PIMain.c of the form PI_HFT_OPTIONAL. By default these flags are undefined, so if your plug-in attempts to load HFT and cannot, initialization will fail. If you define PI_HFT_OPTIONAL with some value (preferably 1) and the load is not successful, initialization will continue.

Use the ASGetConfiguration API to determine the configuration on which the plug-in is running. Use conditional logic in your code so that it makes calls only to APIs that are supported on that particular configuration. In any case, your code should check for NULL HFTs so that it does not call APIs that are not supported on the current configuration.

**NOTE:** Under Adobe Reader, when a rights-enabled PDF file is opened, a flag is set that allows a plug-in to use APIs that become enabled as a result of loading the rights-enabled PDF.

Familiarize yourself with the features available on the different configurations of Acrobat to ensure that you install plug-in menus and toolbars appropriately at initialization. Ensure that you make calls only to APIs supported on the platform detected.
Upgrading Plug-ins From Acrobat 6 to Acrobat 7

General Information

Updating the UI

Third party developers should add their plug-in's menu item to the Advanced menu. The Acrobat SDK menu is available as a submenu under the Advanced menu; it is recommended that third party developers take a similar approach.

Migrating to XML Forms

The XML Forms Architecture (XFA) is a forms architecture built around standards-based XML. With XML being an extensible markup language allowing developers to create and define their own XML tags, the XML Forms Architecture can be considered as a standards-based electronic forms framework, which uses XML tags to describe each part of an electronic form.

Up until recently, form designers wanting to publish their electronic forms in PDF format were required to:

1. Create the form layout (i.e. the static content) in a source application besides Adobe Acrobat or Adobe Reader (e.g. Adobe InDesign).
2. Convert the document to PDF.
3. Add the form field controls and form processing logic to the PDF document using the full version of Adobe Acrobat.

For any changes to be made to the form layout, the form designer had to revert back to the source application, make the changes, and convert to PDF again. With the latest release of Adobe LiveCycle Designer, it is now possible to design the form, add the form controls, and form processing logic in the one process, using one application.

Further benefits of using Adobe LiveCycle Designer for form design and publishing include:

- More intuitive user interface for form layout and design, resulting in increased efficiency for form authors.
- Easily bind forms to XML schemas, databases and web services without the need for extensive scripting knowledge. This helps the form author to quickly and easily integrate their forms with existing applications.
- Create dynamic forms that can increase and/or decrease form space based on events.
- Publish forms in either PDF or HTML format.

For more information on migrating from Acrobat Forms to XML Forms, see the Adobe LiveCycle Designer documentation provided with the Acrobat 7 SDK.
Adobe Dialog Manager in Acrobat 7

ADM in Acrobat 7 is almost identical to ADM in Acrobat 6. However, the header files are now in the ADM subfolder of the Headers folder instead of the ADMHeaders subfolder in Acrobat 6. You must change the path to the headers accordingly (see “Upgrading Plug-ins on Windows” on page 158 and “Upgrading Plug-ins on a Macintosh” on page 159).

Note that in the Acrobat 6 SDK, the IADM object-oriented utility interface was removed, though developers could still compile their plug-ins using the IADM headers from Acrobat 5. Use of the Acrobat 5 header files is no longer supported by Adobe. Developers who used this interface in the past should remove it from their plug-ins and replace it with direct calls to the ADM API.

Removing the IADM Interface From Your Plug-in

There are two kinds of work involved in changing an existing program using IADM to one making direct calls to ADM. One is to change C++ object-oriented IADM code to C-style ADM code. The other is to make the appropriate code architecture changes. Both of these are described below.

To change from C++ to C, the IADM code:

```
IADMItem item = GetItem (IDC_STARTPAGE);
item.Enable (!bAllPages);
item.SetUnits (kADMNoUnits);
item.SetMinIntValue (1);
item.SetMaxIntValue (iOption);
item.SetIntValue (ASCabGetInt(*cmdParams, kPIStartPage, 1));
```

can be converted to:

```
ADMItemRef item = sADMDialog->GetItem(dialogRef, IDC_STARTPAGE);
if(item) {
    sADMItem->Enable(item, !bAllPages);
    sADMItem->SetUnits (item, kADMNoUnits);
    sADMItem->SetMinIntValue (item, 1);
    sADMItem->SetMaxIntValue (item, iOption);
    sADMItem->SetIntValue (item, ASCabGetInt(*cmdParams, kPIStartPage, 1));
}
```

You need to go through related parts in your files to make the conversion.

You also need to make appropriate code architecture changes. The approach varies with the specific program, but a good approach requires you make only minimum changes in the whole program. Therefore, this work is more important and should be done before making the code changes mentioned above.

Basically, the IADM classes that you derived from IADM classes, such as CAcroADMDialog, BaseADMItem, etc., need to be modified or eliminated.
In simple cases, particularly if you use a modal dialog, you may eliminate the classes. For example, to replace the IADM code:

```c
CIsolateDataCmdDlg dlg;
dlg.SetUserData (&cmdParams);
iRetVal = dlg.Modal (sADMPluginRef, "ADBE:IsolateDataCmdDlg",
                 IDD_ISOLATE_CMD, kADMModalDialogStyle, kADMModalDialogStyle, 0);
if (iRetVal == IDOK)
    AVCommandSetParams (cmd, cmdParams);
```

you can write a non-class function to pop up the modal dialog box, then call the function where the dialog is needed.

```c
int IsolateDataCmdADMDlg(void* data)
{
    // initialize ADM
    ADMInitialize();
    // Display modal dialog to get user input
    int iRetVal = sADMDialog->Modal(sADMPluginRef, "ADBE:IsolateDataCmdDlg",
                                    IDD_ISOLATE_CMD, kADMModalDialogStyle, DialogInit, (ADMUserData)
                                    data, NULL);
    // Release ADM
    ADMRelease();
    return iRetVal;
}
```

```
// Call it somewhere when you need to pop up the dialog
iRetVal = IsolateDataCmdADMDlg(&cmdParams);
if (iRetVal == IDOK)
    AVCommandSetParams (cmd, cmdParams);
```

Also, you need to have ADM dialog init, notification, and other callback functions in the form of non-class functions. They should be converted from the member functions of the dialog class. In ADM you never need classes for ADM items such as buttons and text fields, so eliminate them and initialize the items in the dialog init function, and set the item notification callback function correspondingly.

In the case of using modeless dialogs, some public member functions of the dialog IADM class may be called in other places or files. You may want to keep the dialog IADM class and the public member function prototypes, so that you don’t break other program successions or files.

Here are some hints for the approach:

- Keep the dialog class, but do not inherit from `CAcroADMDialog`.
- Add a private class member `ADMDialogRef* m_pAdmDialog` to the class.
- Modify member function `Create()` inside the function, call `sADMDialog->Create()` with `this` (this class object) as userdata, and assign the returned dialog
to the class member `m_pAdmDialog` if the call succeeds. Then, the class member `m_pAdmDialog` can be used to access the ADM dialog throughout your dialog class.

- Eliminate all the classes for dialog items. Initialize the items in dialog init function, and set corresponding notification or other callback functions.

- When writing ADM dialog or item init, notification, or other callback functions, call the ADM function `GetUserdata()` to get the object of your dialog class; then you can use the class member functions or its data members inside those non-class functions.

- You can keep most of the class member functions. It is certainly very important to keep all public member function prototypes, but you may keep some private functions as well. You may write functions to get and set user data associated with the dialog class, and use them to transfer data to or from the ADM dialog.

- Make the code changes after the class architecture is completed.

Two Acrobat 5 SDK plug-in samples using IADM were converted to Acrobat 6 using ADM. These samples were included in the Acrobat 6 SDK. BatchCommand uses a modal dialog; ProgressMonitor uses a modeless dialog.

### Updates to ASCoord and AVRect

The `ASCoord` datatype is being deprecated in favor of more descriptive and accurate datatypes. For new plug-ins, you should use:

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVDevCoord</td>
<td>If the coordinate is an <code>AVPageView</code> device coordinate</td>
</tr>
<tr>
<td>AVWindowCoord</td>
<td>If the coordinate is an <code>AVWindow</code> coordinate</td>
</tr>
<tr>
<td>AVScreenCoord</td>
<td>If the coordinate is a screen coordinate, such as <code>AVWindowSetFrame</code></td>
</tr>
</tbody>
</table>

Though your existing plug-ins will continue to be backward compatible, you should also change them accordingly.

If you are currently using `ASCoord` for any coordinates other than those listed above, then you may be using `ASCoord` inappropriately and it should be changed.

You should also replace the `AVRect` datatype with the more descriptive `AVDevRect`, `AVWindowRect`, and `AVScreenRect` datatypes.
Upgrading Plug-ins on Windows

In general, Acrobat 6 and earlier plug-ins require no code modification to make them compatible with Acrobat 7 running on any Windows platform.

For supported platforms and development environments, see “SDK Platforms” on page 16.

The Acrobat 7 SDK is supported only on Visual Studio .NET 2003, so you must upgrade your plug-ins from previous versions of Visual Studio.

Use Visual Studio .NET 2003 to automatically convert your project from previous versions of Visual Studio. Once converted, you must also make the following changes to your plug-in's project:

- Update the relative path of the header files in the file AcroDspOptions.rsp. The Headers folder now contains three sub-folders: ADM, API and SDK (see “Development Header Files Changes for Acrobat 7” on page 152).

- Files that are located in the header files folders and are source files of the project need to have their paths updated. For each plug-in, you need to update the path for PIMain.c. For plug-ins that use ADM, you must update the path for ADMAcroSDK.cpp and ADMAcroSDK.h. To update these files, delete the source file from your project and then add it from the correct path.


- Turn on incremental linking for each plug-in being upgraded. In Visual Studio 2003, under Project->Properties->Linker->General, set the Enable Incremental Linking flag to Yes.

Once you have upgraded your project to Visual Studio .NET 2003, recompile your plug-in using the headers provided in the Acrobat 7 SDK.

Be sure that the AcroSDKPIDir environment variable is set correctly. Under My Computer->Control Panel->System->Advanced, set this environment variable to where you wish your compiled plug-ins to appear. Normally this is the following, but the location may vary depending on where Acrobat 7 is installed on your system:

C:\Program Files\Adobe\Acrobat 7.0\Acrobat\plug_ins\AcrobatSDK

To support the changes to the data types in the API, a “compatibility layer” has been provided to allow backward compatibility. See “Backward Compatibility” on page 151 for more information.

Reader-enabling Windows Plug-ins

To enable a Windows plug-in, use the Reader-enabling tool supplied with the Acrobat 7 SDK. See the document, Adobe Reader-enabled Plug-ins, for details.
Upgrading Plug-ins on a Macintosh

Beginning in Acrobat 7.0, all plug-ins developed on the Mac OS X must use the Mach-O runtime architecture. A plug-in can be built either as a bundle or a framework – each option has pros and cons for both developers and users.

**NOTE:** Plug-ins developed in previous SDK versions must be rebuilt using Mach-O.

This is a basic outline of the steps and settings needed to convert an existing Acrobat 6 plug-in project to a plug-in project for Acrobat 7. The process and settings may not be accurate or complete for every project. Please refer to Apple Developer documentation and Metrowerks Code Warrior documentation for full information on the Apple Mach-O architecture settings for Mach-O bundle packages. For additional details regarding Acrobat 7 plug-in projects settings, please review the settings in the samples included with the SDK.

The first thing is to convert your plug-in project from CodeWarrior 8.x to 9.2. CodeWarrior can do this for you automatically. After converting your project to CodeWarrior 9.2, make sure that all project files are present and accessible.

**Target settings**

1. The Linker must be set to **Mac OS X PowerPC Mach-O**.
2. Update the output directory, if needed.
3. Update the User Paths to include the Acrobat 7 SDK Headers.
4. Add `MSL_ALL_Macho-o_D.lib` for the debug configuration. Add `MSL_ALL_Macho-o.lib` for the release configuration.
5. Enable **Interpret foreign path separators** for Mach-O compatibility.
6. Ensure the **Build Extras** are C/C++ compatible. These settings are language dependent.
7. Set the **Project Type to Bundle Package** and update the bundle name to include a `.acroplugin` suffix. Creator is **CARO** and Type is **XTND**.
8. **Property List** version is 1.0.
9. To successfully compile with the SDK headers, under **Language Settings**, enable the following:
   - **Enable bool Support**
   - **Enums Always int**
10. Set **Linker Main Entry Point** to `_main`.
Project Settings

1. Add and position in the **Linker** panel the appropriate Mach-O supporting C/C++ standard libraries and precompiled headers to the project. Note that `bundle1.o` is required for bundle packages, rather than `crtl.o`.

2. Add **Carbon Framework** and **System Framework** via the **Frameworks** tab, and any additional frameworks as needed, MSL C/C++ libraries, and `/usr/lib` and `/usr/include` directories.

3. Update the package actions to Mach-O structure. For an example, see the Acrobat SDK 7 plug-in samples. You must include the `resourceVersion.xml` file under **MacOS** and under **en.lproj**.

4. Remove all old CFM libraries and files from the project.

Reader-enabling Macintosh Plug-ins

As noted above in “Upgrading Plug-ins on a Macintosh” on page 159, all Macintosh plug-ins must be rebuilt using Mach-O in order to run on Acrobat 7. Once rebuilt, you must also re-enable them for Adobe Reader.

To enable a Macintosh plug-in, use the Reader-enabling tool supplied with the Acrobat 7 SDK. See the document, *Adobe Reader-enabled Plug-ins*, for details.
Registering and Using Plug-in Prefixes

This chapter lists rules for naming certain parts of your plug-in, as well as data placed in PDF files. All plug-in developers must follow these rules to prevent their plug-ins from conflicting with other plug-ins or accidentally overwriting data in PDF files.

Obtaining a Developer Prefix

Adobe maintains a registry of four-character prefixes. Each plug-in developer must contact Adobe to obtain a prefix.

To obtain a prefix, contact Acrobat Developer Support to obtain a four-character prefix to be used in all plug-ins you or your company develop. Examples of prefixes are ADBE or Acro — both of which are used by Adobe. This chapter uses ADBE in the examples; you would replace it in your plug-in with the prefix assigned to you.

Please register Acrobat plug-in prefixes at Adobe Solutions Network site:
http://partners.adobe.com/asn

NOTE: Registering a prefix with Acrobat Developer Support ensures that no other developer is assigned the same prefix. It is up to you to ensure that names are unique among all plug-ins you or your company write (for example, you must ensure that two of your plug-ins do not both use ADBE_save as a menu item name.)

Using the Developer Prefix

Every plug-in must use the prefix to name its various elements as well as private data it writes into PDF files. The following sections describe and provide an example of each element that must use the prefix.

Plug-in Name

ExtensionName, used in plug-in handshaking, must be of the form Prefix_Name.

hsData->extensionName = ASAtomFromString("ADBE_SuperCrop");
Menus

Menu names must be of the form **Prefix:MenuName**.

```c
SuperCropMenu = AVMenuNew(SuperCrop, "ADBE:SuperCropMenu", gExtensionID);
```

Menu Items

Menu item names must be of the form **Prefix:MenuItem**.

```c
SelSuperCropTool = AVMenuItemNew(SuperCrop, "ADBE:SuperCropMItem", NULL, false, '9', AV_OPTION, SuperCropIcon, gExtensionID);
```

Tools

Tools must have a `GetType` method that returns a name of the form **Prefix:Tool**.

```c
static ACCB1 ASAtom ACCB2 SuperCropToolGetType(AVTool tool) {
    return ASAtomFromString("ADBE:SuperCropTool");
}
```

Toolbar Buttons

Toolbar buttons must have a name of the form **Prefix:ToolbarButton**.

```c
myButton = AVToolButtonNew(ASAtomFromString("ADBE:HelloWorld"), (void *) myBM, false, false);
```

Private Data

When adding private data to keys defined in the PDF specification, it must be added to the PDF file with a key name of the form **Prefix_PrivDataName**. When adding additional keys that are directly referenced from private keys, it is not necessary to use the developer prefix. In the following example, the keys named **First** and **Second** cannot be referenced from any object in the PDF file except the private key that uses an appropriate prefix, so there is no need to use a prefix for the latter two keys.

```c
/Prefix_PrivDataName << /First 2 /Second << /Third [ 2 3 ] >> >>
```

```c
anAnnot = PDPageAddNewAnnot(pdPage, annCnt - 1, ASAAtomFromString("ADBE_MarkupData"), &rcPage);
```
NOTE: Please contact Acrobat Developer Support if you add private data to your plug-in that you believe is useful to other plug-ins. Developer Support would like to know what types of private data plug-ins save in PDF files so that similar data can be standardized with the same name and data format. In this way, developers of different plug-ins using similar data can be assured that their plug-ins can interchange data. For more information regarding private data in PDF files, see the *PDF Reference, fifth edition, version 1.6*.

### Actions

Actions must have a **pdfName** of the form `Prefix_ActionName`.

```c
AVAppRegisterActionHandler(&BkmkHandler, NULL, "ADBE_HWAction", "HWAct");
```

### Annotations

Annotations must have a **GetType** method that returns a name of the form `Prefix_AnnotType`. If you create custom actions or annotations that would be useful to other developers, inform us of the appropriate annotation or action information so that other developers do not reinvent the same type.

```c
static ACCB1 ASAtom ACCB2 Annotation_GetType(  
    AVAnnotHandler AnnotHandler)
{
    return(ASAtomFromString("ADBE_MarkUpAnnot"));
}
```

### Security Handlers

Security handlers must have a **pdfName** of the form `Prefix_SecurityHandlerName`.

```c
PDRegisterCryptHandler(handler, "ADBE_Subscript", "Subscription");
```

### HFTs

When your plug-in exposes any HFTs of its own, it must use an HFT name of the form `Prefix_HFTName`.

```c
gDebuggWinHFTServer = HFTServerNew("ADBE_DebugWin",  
    provideDebugWinHFTCallback, NULL, NULL);
```
Registering and Using Plug-in Prefixes

Using the Developer Prefix
The Acrobat API is almost completely platform-independent. By using the memory allocation and file system APIs provided by Acrobat or Adobe Reader, many parts of a plug-in can be highly portable across platforms.

This chapter contains platform-specific development information for the Macintosh and Windows platforms and provides guidelines that can help make plug-ins more portable among the various supported platforms.

---

**Platform-dependent Data in the API**

The following are platform-specific data types that appear explicitly in the Acrobat API:

- **platformThing** — The data structure that represents a window.
- **platformPath** — The data structure that represents the path to a file.
- **platformEvent** — The data structure that represents mouse clicks, key presses, window activation, and so forth.
- Return codes from AS methods — Constants that indicate, for example, that a file could not be opened because it was not found.

The following are platform-specific data types that do not appear explicitly in the API, but are used by Acrobat, Adobe Reader or plug-ins.

- Cursors — Data structures representing a cursor.
- Toolbar button icons — Pixmaps that appear in the Acrobat or Adobe Reader toolbar.
- Menu item icons — Icons that some platforms allow you to display adjacent to a menu item.
- Menu items — Keep in mind that not all Acrobat or Adobe Reader implementations have the same menu items. For example, Acrobat and Adobe Reader on UNIX® do not have a Window/Tile menu item.
Portability Techniques

The following are techniques that can improve your plug-in's portability:

- Use predefined types instead of `short` and `long`.
- Use Acrobat API methods wherever possible instead of platform-specific APIs.
- Use `#if` around platform-specific code such as dialog boxes and use the predefined platform constants (`MAC_PLATFORM`, `UNIX_PLATFORM`, `WIN_PLATFORM`, and so forth) to test what platform you are compiling for.
- Place platform-specific code in separate files from the main portion of the plug-in, so that you can easily recognize and rewrite platform-dependent sections.
- Use the Adobe Dialog Manager (ADM) for dialogs. For details, see the *Adobe Dialog Manager Programmer's Guide and Reference*.

Macintosh Techniques

This section contains information necessary to implement plug-ins for the Macintosh using Mac OS X.

Developing a Macintosh Mach-O Plug-in

Beginning in Acrobat 7.0, all plug-ins developed on the Mac OS X must use the Mach-O runtime architecture. A plug-in can be built either as a bundle or a framework – each option has pros and cons for both developers and users.

**NOTE:** Plug-ins developed in previous SDK versions must be rebuilt using Mach-O.

Using the Correct Development Environment

Metrowerks CodeWarrior version 9.2 is the currently supported development environment for plug-in development on the Macintosh platform. Metrowerks CodeWarrior version 9.2 contains the correct headers and libraries as well as extensive documentation on making plug-ins (and applications) Mach-O and Carbon compliant.

Using the Samples

You are encouraged to use the *Starter* plug-in sample as a basis for developing your plug-ins. This sample has all of the appropriate project settings. The *Starter* sample does nothing other than build a loadable plug-in. In addition, other plug-ins that could be useful as a starting point for developing a plug-in are available. See the *Guide to SDK Samples* for a description of each of these samples.

Setting Up Your Plug-in’s Package

*Figure 15.1* shows the structure of the Starter plug-in package distributed with the Acrobat 7 SDK.
The package structure for your plug-in should resemble that of the Starter plug-in. The `Info.plist` file contains a list of properties used by the package. Adobe provides a file, `info.plc`, that creates a `info.plist` file. Refer to Code Warrior’s documentation for more information regarding Code Warrior’s property list compiler (PLC) for customizing the `.plc` file.

The `ResourceVersion.xml` file contains information that is used by Code Warrior to generate `info.plist` from `info.plc`. This file is supplied with the Acrobat 7 SDK and must be included in your plug-in’s package.

**Establishing Carbon/Property List Compliance**

*Carbonization.* Carbon is a set of programming interfaces that include header files and a library called `CarbonLib`. These interfaces can be used to build Acrobat 5 plug-ins (and applications) that run on Mac OS 8.1 and later. Acrobat 6 and 7 use the newer plists.

Carbon interfaces include most of the functions commonly used for developing on the Macintosh platform. By developing to these interfaces and linking with `CarbonLib`, you create a “carbonized” executable that can run on Mac OS X (in native mode) as well as Mac OS 8/9.

You *must* carbonize your plug-in if you intend that your plug-in use the Adobe Dialog Manager (ADM).
NOTE: To prevent problems with older style event handling, plug-ins need to replace calls to `WaitNextEvent` with calls to `RunCurrentEventQueue` or `ReceiveNextEvent`.

For More Information on Carbon Compliance. Metrowerks CodeWarrior version 6 and above contains the correct headers and libraries as well as extensive documentation on making plug-ins (and applications) Carbon compliant. For additional information, see http://developer.apple.com/documentation/Carbon/Conceptual/carbon_porting_guide

Using `GetGWorld` Over `SetPort`

With the move to carbonization and double buffering, you should use `GetGWorld` over the toolbox call `SetPort`. Using both calls in the same plug-in can cause the current port to get out of sync with the current device. Using `GetGWorld` only maintains the correct port and device settings.

In all cases, you should pass `GetMainDevice` unless you have a particular device in mind or you are restoring the `GWorld` to its original state. An example would be:

```c
ACCB void ACCB2 foo(AVPageView pageView)
{
    CGrafPtr oldGWorld, pagePort = NULL;
    GDHandle oldDevice;

    pagePort = (CGrafPtr)AVPageViewAcquireMachinePort(pageView);
    if (pagePort)
        GetGWorld(&oldGWorld, &oldDevice);
    SetGWorld(pagePort, GetMainDevice());
    // ...Draw to the port here
    SetGWorld(oldGWorld, oldDevice);
    AVPageViewReleaseMachinePort(pageView, pagePort);
}
```

Locating and Loading Plug-ins

When Acrobat or Adobe Reader launches, it scans the `Plug-ins` folder to locate and load plug-ins (including plug-ins represented by alias files in that folder). PowerPC plug-ins must have creator `CARO` and type `XTND`, and contain a `cfrg` resource with ID=0 (see Inside Macintosh: PowerPC System Software for more information on code fragment storage).

When Acrobat or Adobe Reader locates a plug-in, it scans the fragment descriptors in the `cfrg` resource. It loads that fragment into memory and jumps to the plug-in's entry point to begin handshaking.

NOTE: Testing has only been performed with PowerPC plug-ins that contain exactly one code fragment, comprising the entire data fork of the plug-in.
Memory

Acrobat or Adobe Reader’s memory allocator gets its memory from the system as opposed to the application’s memory partition.

“Memory Allocation” on page 34 discusses general guidelines about allocating memory. Using these guidelines is particularly important on the Macintosh to insure that memory is allocated from the system rather than the application partition. Otherwise your plug-in is very likely to cause Acrobat or Adobe Reader to run out of memory.

Resource File Considerations

Acrobat or Adobe Reader open a plug-in’s resource fork with read-only permissions. In addition, plug-ins cannot assume that their resource file is on top of the resource chain each time they are entered via an ASCallback. Plug-ins must explicitly move their resource file to the top of the resource chain before accessing resources in it. As a result, all code that directly or indirectly calls getResource must be modified. This can be accomplished either directly or by using the SafeResources routines in the Acrobat SDK.

Using SafeResources

The recommended way to access resources in the plug-in file is to use the functions declared in the header file SafeResources.h in the SDK. These functions are direct replacements for each Toolbox function that directly or indirectly calls getResource. The replacement functions automatically place the plug-in file on top of the resource chain before accessing the resource, and restore the old resource chain after accessing the resource.

Manipulating the Resource Chain Directly

If you choose to manipulate the resource chain directly, you must modify all code that directly or indirectly calls getResource. The list of such Toolbox calls can be determined from SafeResources.h, by removing the prefix Safe from the names of the calls. Before calling each such Toolbox function, you must put the plug-in’s resource file on top of the resource chain, and after such calls, you must restore the old resource chain. For example:

```
DialogPtr myDialog = GetNewDialog(23, NULL, (Ptr) -1);
```

must be rewritten as:

```
short oldResFile;
DialogPtr myDialog;

oldResFile = CurResFile();
UseResFile(gResFile);
myDialog = GetNewDialog(23, NULL, (Ptr) -1);
useResFile(oldResFile);
```

The global variable gResFile is automatically set up during handshaking and is declared in PICommom.h.
Macros

The following macros must be defined/set:

- **POWER_PC** must be defined
- **PLATFORM** must be defined as **MacPlatform.h**
- **PRODUCT** must be defined as **Plugin.h**

All of these macros are automatically defined correctly for the platform and development environment by the header file **PIPrefix.h**. You are encouraged to use this header file.

Macrowerks CodeWarrior users can accomplish this in any of the following ways:

- Use **PIPrefix.h** as your prefix file
- Include **PIHeaders.pch** in your project and use the precompiled header file **PIHeadersMWERKSPPC** as your prefix file. (Plug-in writers are encouraged to use the precompiled headers.) This will precompile the plug-in headers for you each time you completely rebuild your project and include **PIPrefix.h**.
- Use your own prefix file that includes **PIPrefix.h** as its first line.
- If you are developing an Adobe Reader plug-in, you also must define a macro to access HFTs available to Adobe Reader. For details, see “Rules to Follow” on page 25.

Macintosh-Only Methods

Plug-ins should *not* use the **ASPathFromPlatformPath** method on the Macintosh platform. Instead, they should call **ASFileSysCreatePathName**.

**NOTE:** For Acrobat 6, APIs were added to create **ASPathnames** from **FSSpec**, **FSRef**, **CFURL**, and **POSIX** pathnames. See the **Acrobat and PDF Library Reference** for details.

The **AVAppDidOrWillSwitchForDialog** method is only useful to plug-ins on the Macintosh platform.

Interapplication Communication

Plug-ins can add their own Apple events to those supported by Acrobat or Adobe Reader, by hooking into Acrobat or Adobe Reader’s Apple event handling loop. This is done by replacing the **AVAppHandleAppleEvent** method in the API. If the 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 Acrobat or Adobe Reader the opportunity to handle the Apple event.
Debugging

Source code debugging is straightforward with CodeWarrior. You can initiate a debugging session directly with CodeWarrior, or double-click the symbols files to launch the debugger. In the latter method, a dialog box displays asking for the location of the project’s executable; indicate the location of the plug-in. A debugging window appears; open the desired file and set breakpoints. Finally, launch Acrobat or Adobe Reader. You can also start the debugger after Acrobat or Adobe Reader is running.

Windows Techniques

This section contains information necessary to implement plug-ins under Windows.

Developing a Windows Plug-in

Use Visual C++ .NET.

The Acrobat 7 SDK introduces the use of the AcroSDKPIDir environment variable on Windows systems. Under My Computer->Control Panel->System->Advanced, set this environment variable to where you wish your compiled plug-ins to appear. Normally this is C:\Program Files\Adobe\Acrobat 7.0\Acrobat\plug_ins\AcrobatSDK, but the location may vary depending on where Acrobat 7 is installed on your system.

You are encouraged to use the plug-in samples BasicUI and Starter as a basis for developing plug-ins. These samples have all of the appropriate project settings. The Starter sample does nothing besides build a loadable plug-in while the BasicUI sample adds menu items. In addition, see the Guide to SDK Samples for other plug-ins that could be useful as a start for development.

Plug-in Implementation

Plug-ins are implemented 32-bit DLLs on the Windows® platform. With Acrobat 4.0 and higher, there are no 16-bit versions of Acrobat or Adobe Reader. Plug-in names must end in .API.

Locating and Loading Plug-Ins

When Acrobat or Adobe Reader launches, it scans the Plug_ins folder in the same directory as the Acrobat executable for DLLs with the extension .API. Acrobat or Adobe Reader also searches nested directories, allowing you to group plug-ins in folders. When it locates a file with the extension .API, it looks for the exported symbol PlugInMain. It loads the plug-in by calling LoadLibrary and then calls the function pointed to by the symbol PlugInMain.
LoadLibrary calls the DLLMain entry point of the plug-in with the parameter DLL_PROCESS_ATTACH passed. It’s possible for a developer to run some initialization code in DLLMain (such as allocating memory) before its PluginMain procedure is called by Acrobat or Adobe Reader. If you do this, it’s important to deallocate the memory when DLLMain is called with DLL_PROCESS_DETACH.

If you rely on PluginUnload for deallocation of memory, it could fail if Acrobat or Adobe Reader unloads the plug-in immediately without calling the plug-in’s handshaking callbacks. This would happen in the following situations:

1. If the plug-in is not Adobe-certified and the user has specified the Certified Plug-ins Only option (accessible through Edit->Preferences->General->Options->Startup).

2. If the plug-in is running under Acrobat Reader, but it is not Reader-enabled. This could potentially cause a crash when Acrobat or Adobe Reader quits.

Why a Plug-in Might Not Load

There are several reasons that a plug-in may not load successfully:

- You forgot to change the plug-in’s extension from .DLL to .API.
- There are too many plug-ins already loaded by either Acrobat or Adobe Reader. There is a limit to the number of plug-ins that can be loaded at any one time. The number is variable and dependent on the code generation settings of all loaded plug-ins. See “Handling the Thread Local Storage (TLS) Limit” on page 174.
- The plug-in attempts to register with the same extensionName as another plug-in that has already loaded. In this case, Acrobat or Adobe Reader displays an error message indicating the problem.
- You cloned your project from an existing plug-in project that uses a .def file and forgot to change the LIBRARY entry in the .def file.
- The DLL is bad. This can occur even if the plug-in compiled and linked with out errors. Generally, rebuilding the plug-in completely (doing a Rebuild All) solves the problem.
Macros and Project Settings

The following macros must be defined or set in your preprocessor definitions:

- WIN_ENV
- WIN_PLATFORM (preferred)
- WIN32
- _WINDOWS

For a plug-in to be loaded, it must export the symbol **PlugInMain**. This can be done by including a .def file in the project for the plug-in or by including the line

/EXPORT:PlugInMain in the project settings for the plug-in.

If you are developing an Adobe Reader plug-in, you also must define a macro to access HFTs available to Adobe Reader. For details, see “Rules to Follow” on page 25.

Interapplication Communication

Plug-ins can add their own DDE messages to those supported by Acrobat or Adobe Reader, by registering to receive DDE messages directly. The DDEClnt sample in the Acrobat SDK shows how to do this.

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

It is not possible for a plug-in to implement OLE automation or be an ActiveX® server through the use of MFC. This is because Acrobat or Adobe Reader uses MFC to implement its OLE automation and there cannot be two MFC-based OLE automation servers in the same process.

OLE or ActiveX server plug-ins must be implemented using the ActiveX Template Library.
Platform-Specific Techniques
Windows Techniques

Debugging

Generally, the debugger built into Visual C++ is adequate to debug plug-ins, but CodeView and other techniques can sometimes be useful. Debugging a Windows plug-in compiled with Visual C++ is quite simple if you remember a few things:

- Specify the Acrobat plug-ins directory under the link | output tab in the Project settings dialog box.
- Specify the Acrobat or Adobe Reader executable under the executable for debug session in the Project settings dialog box.
- The first time you build a plug-in, do a Rebuild All.
- Set breakpoints in your source code by selecting the line and clicking the hand icon or pressing the F9 key.
- After setting breakpoints, press the F5 key to have MSVC launch Acrobat or Adobe Reader. Because Acrobat and Adobe Reader do not ship with symbols, you may see a dialog box displaying the following message:
  The file 'ACROBAT . EXE' does not contain debugging information. Do you want to continue?
  Respond with Yes.
- The plug-in has a .API extension. (Without this, the plug-in will not load.)

There are two common reasons why breakpoints may not be hit:

- You launched Acrobat or Adobe Reader from the File Manager or Program Manager. Acrobat or Adobe Reader must be launched from with MSVC to debug plug-ins.
- You copied your plug-in into Acrobat's plug-in directory, instead of specifying the plug-ins directory in the link | output dialog box.

Handling the Thread Local Storage (TLS) Limit

There is a limit to the number of plug-ins that Acrobat or Adobe Reader can load at any given time. The limit is caused by a limitation of the multi-threading model used by the Win32 API and is dependent on the code generation settings of the plug-ins being loaded. The following is background information on the limit and guidelines for you to follow in minimizing the impact of their plug-ins on the ability of Acrobat and Adobe Reader to load plug-ins.

Background

When a process is created, an array of bit flags is allocated for the management of thread specific data. In the current Win32 implementation, this array is limited to 64 members or TLS slots. Every DLL/plug-in that uses thread local storage is allocated at least one slot when loaded using LoadLibrary. This includes system DLLs, plug-ins, and all the DLLs they load. When all of the TLS slots have been occupied for a process, LoadLibrary will fail for any DLL requiring a TLS slot.
**NOTE:** Acrobat and Adobe Reader do not currently raise an error when a plug-in fails to load due to the TLS limit.

**Rules to Minimize Use of TLS Slots**

The following guidelines will minimize the TLS slots occupied by plug-ins:

- Plug-ins that are not multi-threaded should only link with the single threaded runtime libraries that do not occupy a TLS slot.

- Plug-ins that are multi-threaded should consider linking with the multi-threaded DLL runtime library. Both the DLL and static versions of the runtime libraries occupy a TLS slot. However, many plug-ins shipped with Acrobat or Adobe Reader use the DLL version so the runtime DLL will not occupy another TLS slot after it has been loaded by the process.

**Using Modal Dialogs**

**NOTE:** You can use the Adobe Dialog Manager (ADM) to create and manage cross-platform modal and modeless dialogs for your plug-in. For details, see the *Adobe Dialog Manager Programmer's Guide and Reference*.

If you write plug-ins that contain modal dialogs on the Windows platform, and you are not using ADM to create your dialogs, you need to perform the following steps:

1. When you are creating your dialog, get the parent `HWND` of the dialog using the `WinAppGetModalParent` method. Then use this `HWND` when creating the dialog.

   If you do not have an `AVDoc` open, use the global handle for the Acrobat or Adobe Reader window in your dialog box (`gHWND`).

   Make sure to get the mouse capture before putting up your dialog box so that Acrobat or Adobe Reader does not receive the mouse clicks. After your dialog box returns, set the mouse capture back.

   Example:
   ```cpp
   HWND CapturehWnd, hParent;
   CapturehWnd = GetCapture();
   if ( CapturehWnd != NULL )
     ReleaseCapture();

   hParent = WinAppGetModalParent(AVAppGetActiveDoc());
   nRetVal = DialogBox(gHINSTANCE, MAKEINTRESOURCE(IDD_PROPS), hParent, PropsDialogProc);
   if ( CapturehWnd != NULL )
     SetCapture( CapturehWnd );
   ```
2. As soon as you have an HWND for the dialog itself, usually in response to the WM_INITDIALOG message, you should acquire a new AVWindow using the AVWindowNewFromPlatformThing method. Save this AVWindow in some place where you can access it when the dialog is destroyed. Then pass the AVWindow to the AVAppBeginModal method.

Example: Here is code that is called in response to a WM_INITDIALOG message:

```c
static   AVWindow sAVWin;
....
/* hWnd is the window handle of the dialog window */
sAVWin = AVWindowNewFromPlatformThing(AVWLmodal, 0, NULL,
    gExtensionID, hWnd);
AVAppBeginModal(sAVWin);
```

3. At dialog destroy time, usually in response to a WM_DESTROY message, end the modal operations using AVAppEndModal. If you are not using MFC, destroy the AVWindow for which you saved the handle with AVWindowDestroy.

Example: Here is a section of code called in response to a WM_DESTROY message:

```c
AVAppEndModal();
AVWindowDestroy(sAVWin);
```

If you are using MFC to put up your dialog box, do not call AVWindowDestroy in the WM_DESTROY message (MFC will cause Acrobat or Adobe Reader to destroy the AVWindow automatically).
The Acrobat 7.0 SDK has three folders containing header files:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SDK</strong></td>
<td>Contains header files that are common to most plug-ins and are generally referenced from PIMain.c. See Acrobat SDK Header Files.</td>
</tr>
<tr>
<td><strong>API</strong></td>
<td>Contains API header files specific to core and extended APIs. See API Header Files.</td>
</tr>
<tr>
<td><strong>ADM</strong></td>
<td>Contains header files for use with the Adobe Dialog Manager (ADM). See ADM Header Files.</td>
</tr>
</tbody>
</table>

**Acrobat SDK Header Files**

<table>
<thead>
<tr>
<th>Table 15.1</th>
<th><strong>SDK Header Files</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Header File</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>AVCmdDefs.h</td>
<td>Defines the AVCommand names, parameters, etc. to allow developers to drive the built-in AVCommands.</td>
</tr>
<tr>
<td>DebugWindow.h</td>
<td>Catalog of functions exported by the DebugWindow plug-in.</td>
</tr>
<tr>
<td>MacPIHeaders.h</td>
<td>Macintosh specific includes and defines.</td>
</tr>
<tr>
<td>PIHeaders.h</td>
<td>A general include file for Acrobat Headers. This file is included in the Macintosh precompiled header file PIHeaders.pch.</td>
</tr>
<tr>
<td>WinPIHeaders.h</td>
<td>Windows specific includes and defines.</td>
</tr>
</tbody>
</table>
## API Header Files

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acroassert.h</td>
<td>Extended definition of assert macros for various Acrobat supported platforms (for debugging).</td>
</tr>
<tr>
<td>AcroColorCalls.h</td>
<td>Defines names for referencing AcroColor APIs via the corresponding HFT.</td>
</tr>
<tr>
<td>AcroColorExpT.h</td>
<td>Types, macros, structures, etc. required to use the AcroColor Host Function Table.</td>
</tr>
<tr>
<td>AcroColorHFT.h</td>
<td>The Acrobat color HFT</td>
</tr>
<tr>
<td>AcroColorProcs.h</td>
<td>AcroColor API prototype definitions.</td>
</tr>
<tr>
<td>AcroColorVers.h</td>
<td>AcroColor HFT names and versions</td>
</tr>
<tr>
<td>AcroErr.h</td>
<td>Error codes are used in the ASRaise and DURING/HANDLER mechanism established in Except.h.</td>
</tr>
<tr>
<td>AF_ExpT.h</td>
<td>Public data types and structures to handle Acrobat Forms objects.</td>
</tr>
<tr>
<td>AF_Sel.h</td>
<td>Selectors for all AcroForm HFT functions</td>
</tr>
<tr>
<td>AFTTS_Sel.h</td>
<td>Selectors for all AcroTTS HFT functions</td>
</tr>
<tr>
<td>ASCalls.h</td>
<td>AcroSupport level APIs</td>
</tr>
<tr>
<td>ASExpT.h</td>
<td>Types, macros, structures, etc. required to use the AcroSupport HFT</td>
</tr>
<tr>
<td>ASEExtraCalls.h</td>
<td>ASEExtra HFT</td>
</tr>
<tr>
<td>ASEExtraExpT.h</td>
<td>Types, macros, structures, etc. required to use the ASEExtra HFT</td>
</tr>
<tr>
<td>ASEExtraProcs.h</td>
<td>Catalog of functions exported by the ASEExtra HFT</td>
</tr>
<tr>
<td>ASEExtraVers.h</td>
<td>ASEExtra HFT name and version</td>
</tr>
<tr>
<td>ASKey.h</td>
<td>Definition of standard key codes for Mac and Windows</td>
</tr>
<tr>
<td>ASProcs.h</td>
<td>Catalog of functions exported by AcroSupport</td>
</tr>
<tr>
<td>Header File</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ASRaiseAware.h</td>
<td>Code needed for making a class safe across exceptions.</td>
</tr>
<tr>
<td>AVCalls.h</td>
<td>AV HFT</td>
</tr>
<tr>
<td>AVCompat.cpp</td>
<td>Code to call functions from previous versions of the SDK from current SDK</td>
</tr>
<tr>
<td>AVCompat.h</td>
<td>Header file used by AVCompat.cpp</td>
</tr>
<tr>
<td>AVExpT.h</td>
<td>Types, macros, structures, etc. required to use the AcroView HFT</td>
</tr>
<tr>
<td>AVExpTObsolete1.h</td>
<td>Types used in former versions of Acrobat</td>
</tr>
<tr>
<td>AVExpTObsolete2.h</td>
<td>Types used in former versions of Acrobat</td>
</tr>
<tr>
<td>AVPrefsD.h</td>
<td>Defines AV_PREFERENCES, the list of AVAppPreferences</td>
</tr>
<tr>
<td>AVProcs.h</td>
<td>Catalog of functions exported by AcroView</td>
</tr>
<tr>
<td>AVSPCalls.h</td>
<td>AV Sweet Pea HFT</td>
</tr>
<tr>
<td>AVSPProcs.h</td>
<td>Catalog of SweetPea functions exported by AcroView</td>
</tr>
<tr>
<td>AVSPVers.h</td>
<td>AcroView SweetPea HFT name and version</td>
</tr>
<tr>
<td>cathft.h</td>
<td>Catalog of functions exported by Acrobat Catalog</td>
</tr>
<tr>
<td>CAVAlert.h</td>
<td>Definition for a wrapper class of the AVAlert implementation.</td>
</tr>
<tr>
<td>ConsExpT.h</td>
<td>Data structures for PDF Consultant</td>
</tr>
<tr>
<td>ConsHFT.h</td>
<td>Catalog of functions exported by PDF Consultant</td>
</tr>
<tr>
<td>ConsObTp.h</td>
<td>PDFObjectType enumeration definition used by PDF consultant.</td>
</tr>
<tr>
<td>CorCalls.h</td>
<td>Core HFT</td>
</tr>
<tr>
<td>CoreExpT.h</td>
<td>Types, macros, structures, etc. required to use the Core HFT</td>
</tr>
<tr>
<td>CorProcs.h</td>
<td>Catalog of the core exported functions; this table is handed off to the plug-in at initialization time</td>
</tr>
<tr>
<td>CosCalls.h</td>
<td>Cos HFT</td>
</tr>
</tbody>
</table>
### Table 15.2 API Header Files

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CosExpT.h</td>
<td>Types, macros, structures, etc. required to use the Cos HFT</td>
</tr>
<tr>
<td>CosProcs.h</td>
<td>Catalog of functions exported by Cos</td>
</tr>
<tr>
<td>DigSigHFT.h</td>
<td>Digital Signature interface for Acrobat Digital Signature handlers</td>
</tr>
<tr>
<td>DirectoryHFT.h</td>
<td>Directory HFT</td>
</tr>
<tr>
<td>Environ.h</td>
<td>Checks for the existence of a PLATFORM macro and, if it exists, #includes the file it points to</td>
</tr>
<tr>
<td>FormsHFT.h</td>
<td>HFT definitions for Acrobat Form Plugin</td>
</tr>
<tr>
<td>HFTLibrary.h</td>
<td>Product file for Acrobat library definition</td>
</tr>
<tr>
<td>Library.h</td>
<td>Product file for Acrobat library definition</td>
</tr>
<tr>
<td>MacCalls.h</td>
<td>HFT for Platform (Mac Developers)</td>
</tr>
<tr>
<td>MacPlatform.h</td>
<td>Platform file for Macintosh development</td>
</tr>
<tr>
<td>MacProcs.h</td>
<td>Catalog of functions exported by the Mac Viewer</td>
</tr>
<tr>
<td>NSelExpT.h</td>
<td>list of NSelector constants and function prototypes required to register for notifications</td>
</tr>
<tr>
<td>PagePDECntCalls.h</td>
<td>HFT and prototypes for Page PDEContent methods</td>
</tr>
<tr>
<td>PDBasicExpT.h</td>
<td>Types required to use the PDModel HFT. ONLY handles to exported types are defined in this file.</td>
</tr>
<tr>
<td>PDCalls.h</td>
<td>HFT for PD functions</td>
</tr>
<tr>
<td>PDClassDefs.h</td>
<td>Macros commonly used by all PDClasses</td>
</tr>
<tr>
<td>PDClasses.h</td>
<td>Class consolidating all the C++ wrappers for the PD Objects</td>
</tr>
<tr>
<td>PDExpT.h</td>
<td>Types, macros, structures, etc. required to use the PDModel HFT</td>
</tr>
<tr>
<td>PMetadataCalls.h</td>
<td>PMetadata HFT</td>
</tr>
<tr>
<td>PMetadataError.h</td>
<td>Definition of error codes for PMetadata</td>
</tr>
<tr>
<td>PMetadataExpT.h</td>
<td>Types required to use the PMetadata HFT</td>
</tr>
<tr>
<td>PMetadataHFTVers.h</td>
<td>PMetadata HFT name and version</td>
</tr>
</tbody>
</table>
### API Header Files

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDMetadataProcs.h</td>
<td>Procedures that let you store and access descriptive metadata associated with a PDF document and with components within a PDF document</td>
</tr>
<tr>
<td>PDProcs.h</td>
<td>Catalog of functions exported by the PDModel HFT</td>
</tr>
<tr>
<td>PDSError.h</td>
<td>Definition of error codes for PDSEdit</td>
</tr>
<tr>
<td>PDSExpT.h</td>
<td>Types, macros, structures, etc. required to use the PDSEdit HFT</td>
</tr>
<tr>
<td>PDSReadCalls.h</td>
<td>PDSRead HFT</td>
</tr>
<tr>
<td>PDSReadHFTVers.h</td>
<td>PDSRead HFT name and version</td>
</tr>
<tr>
<td>PDSReadProcs.h</td>
<td>Catalog of functions exported by PDSEdit</td>
</tr>
<tr>
<td>PDSWriteCalls.h</td>
<td>PDSWrite HFT</td>
</tr>
<tr>
<td>PDSWriteHFTVers.h</td>
<td>PDSWrite HFT name and version</td>
</tr>
<tr>
<td>PDSWriteProcs.h</td>
<td>Catalog of functions exported by the PDSWrite HFT</td>
</tr>
<tr>
<td>PDSysFont.h</td>
<td>PDSysFont header file</td>
</tr>
<tr>
<td>PDSysFontExpT.h</td>
<td>PDSysFont Types header file</td>
</tr>
<tr>
<td>PDSysFT.h</td>
<td>PDSysFont Types header file</td>
</tr>
<tr>
<td>PEError.h</td>
<td>PDF Edit Error codes used with ASRaise(PDFEditError(peErrMumble))</td>
</tr>
<tr>
<td>PEExpT.h</td>
<td>Types, macros, structures, etc. required to use the PDFEdit HFT</td>
</tr>
<tr>
<td>PERCalls.h</td>
<td>PDFEditReade HFT</td>
</tr>
<tr>
<td>PERProcs.h</td>
<td>Catalog of functions exported by the PDFEditRead HFT</td>
</tr>
<tr>
<td>PETypes.h</td>
<td>Defines PDFEdit error codes</td>
</tr>
<tr>
<td>PEVers.h</td>
<td>PDFEdit HFT names and versions</td>
</tr>
<tr>
<td>PEWCalls.h</td>
<td>PDFEditWrite HFT</td>
</tr>
<tr>
<td>PEWProcs.h</td>
<td>Catalog of functions exported by the PDFEditWrite HFT</td>
</tr>
<tr>
<td>PgCntProcs.h</td>
<td>Header for for PDF page content server methods</td>
</tr>
</tbody>
</table>
### Table 15.2 API Header Files

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICommon.h</td>
<td>Globals and function declarations used by every plug-in.</td>
</tr>
<tr>
<td>PIExcept.h</td>
<td>Defines the <code>setjmp</code>, <code>longjmp</code>, and <code>jmp_buf</code> equivalents for Acrobat plug-ins. Plug-in writers will probably not directly #include this file; most likely they will #include &quot;CorCalls.h&quot; and use the <code>DURING/HANDLER/END_HANDLER</code> macros</td>
</tr>
<tr>
<td>PIMain.c</td>
<td>Source code that must be linked into every plug-in</td>
</tr>
<tr>
<td>PIMain.h</td>
<td>Include file for PIMain.c, which contains <code>#defines</code>, macros, function prototypes, and global variable declarations.</td>
</tr>
<tr>
<td>PIPokes.h</td>
<td>List of prototypes for notifications</td>
</tr>
<tr>
<td>PIPrefix.h</td>
<td>The bare minimum prefix file required to compile a Macintosh plug-in for Acrobat or Adobe Reader.</td>
</tr>
<tr>
<td>PIRequir.h</td>
<td>Used to determine which HFTs (and which versions of those HFTS) the plug-in requires.</td>
</tr>
<tr>
<td>PIVersn.h</td>
<td>Data structures, types, and other things related to plug-ins and version changes. This file is shared between Acrobat applications and plug-ins. Also, contains handshaking versioning types and data.</td>
</tr>
<tr>
<td>Plugin.h</td>
<td>PRODUCT file for plugin product configuration.</td>
</tr>
<tr>
<td>PSFCalls.h</td>
<td>PDSysFont HFT</td>
</tr>
<tr>
<td>PSFProcs.h</td>
<td>Catalog of functions exported by the PDSysFont HFT</td>
</tr>
<tr>
<td>PubSecHFT.h</td>
<td>Public key security interface for Acrobat public-key security handlers</td>
</tr>
<tr>
<td>SafeResources.cpp</td>
<td>Glue implementation of &quot;safe&quot; version of Macintosh toolbox routines that put the plug-in's resource file on top of the resource chain before calling the &quot;real&quot; implementation of the trap</td>
</tr>
<tr>
<td>SafeResources.h</td>
<td>Interface definitions for SafeResources.cpp.</td>
</tr>
<tr>
<td>SmartPDPage.h</td>
<td>Class containing a thin wrapper for PDPage objects that ensures the reference count is decremented when the object is destroyed</td>
</tr>
</tbody>
</table>

Acrobat SDK Plug-in Guide
### Table 15.2 API Header Files

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speller_Sel.h</td>
<td>Selectors for all Spelling HFT functions</td>
</tr>
<tr>
<td>SpellerHFT.h</td>
<td>HFT definitions for Acrobat Spell Check procedures</td>
</tr>
<tr>
<td>SrchCls.h</td>
<td>Acrobat Search HFT</td>
</tr>
<tr>
<td>SrchHFT.h</td>
<td>Catalog of functions exported by Acrobat Search</td>
</tr>
<tr>
<td>SrchPrs.h</td>
<td>Catalog of functions exported by Acrobat Search</td>
</tr>
<tr>
<td>SrchType.h</td>
<td>Data types for Acrobat Search API</td>
</tr>
<tr>
<td>TtsHFT.h</td>
<td>HFT definitions for Acrobat TTS procedures</td>
</tr>
<tr>
<td>UnixCalls.h</td>
<td>Unix PLATFORM HFT</td>
</tr>
<tr>
<td>UnixPlatform.h</td>
<td>PLATFORM file for Unix development</td>
</tr>
<tr>
<td>UnixPrs.h</td>
<td>Catalog of functions exported by the Unix Viewer</td>
</tr>
<tr>
<td>WinCalls.h</td>
<td>Windows PLATFORM HFT</td>
</tr>
<tr>
<td>WinExpT.h</td>
<td>Types required to use the Windows HFT</td>
</tr>
<tr>
<td>WinPltfm.h</td>
<td>PLATFORM file for Windows development</td>
</tr>
<tr>
<td>winprocs.h</td>
<td>Catalog of functions exported by the Windows Viewer</td>
</tr>
<tr>
<td>WLHFT.h</td>
<td>Contains all of the public definitions for using WebLink</td>
</tr>
</tbody>
</table>

### ADM Header Files

For information on the ADM header files, see the Adobe Dialog Manager Programmer’s Guide and Reference.
Plug-in Header Files
ADM Header Files
# Index

## A

"about" box and splash screen 33
ACRO_SDK_LEVEL 151
Acrobat viewer’s user interface 32
adjusting the cursor 31
ADM 155
Adobe LiveCycle™ Reader Extensions 19
Adobe Reader 20
annotations 87
  creating 92
  finding 87
  getting and setting 90
APIs
  new 18
Apple events 149
array 123
ASGetConfiguration 18, 153
ASRaise 57

## B

backward compatibility 151
bookmarks 95
  creating 99
  finding 96
buttons 61

## C

callbacks 29, 40, 42
certified plug-in 16
CodeWarrior 9.2 16
coordinate systems 33, 75
core API
  handshaking and initialization 27
  mechanics 15, 23
Cos objects 119
CosDoc 121
CosObj 121

creating HFTs 25

## D

DDE messages 149
DEBUG 30
documents 65
drawing 36
DURING/HANDLER 58

## E

event handling 31
Events 31
exceptions 55
exporting HFTs 28

## H

handling events 31
  adjust cursor 31
  key presses 32
  mouse clicks 31
handshaking 27
handshaking with plug-ins 27
header files 152, 177
help files 33
HFT 23, 109
  creating 25
  exporting 28, 109
  header file 114
  importing 28, 118
  replacing 103
  replacing entries in 104
  servers 24
  source file 115
  using 24
HFT servers 24
host function table (HFT) 23
Index

I
importing HFTs 28
initialization of plug-ins 27
initialization, plug-in 27
Interapplication Communication (IAC) 149

K
key presses 32

M
Mac OS X 16, 159, 166
Macintosh 159
mechanics of the core API 15, 23
menus and menu items 32
message handling, adding 149
methods, replacing 103
mouse click processing 36
mouse clicks 31

N
notification 106
notifications 30

P
page view 73
page view layers 36
PDDoc 68
PDFEdit 76
PDPage 76
PDWord 81
PDWordFinder 82
PIHandshake 42
PIMain.c 39, 44
platforms 16
plug-in
certified 16
initialization 26, 29
lifecycle 40
loading 26
reducing conflicts among 37
registering 161
unloading 29
plug-in initialization 27
plug-in naming conventions 32
plug-in prefixes 32
PluginExportHFTs 40
PluginImportReplaceAndRegister 40
PluginInit 41
plug-ins, Adobe Reader-enabled 16, 18
PluginUnload 41
portability 166
Portable Document (PD) layer
PDWord 81
PDWordFinder 82

R
Reader-enabled plug-ins 16, 18
registering 107
replacing entries in HFTs 104
replacing methods 103
rights-enabled documents 19

S
ShowException 57
splash screen 33
StarterInit.cpp 39, 40
string 123

T
toolbar 33, 61

U
upgrading
Macintosh 159
Windows 158
user interface 32
"about" box and splash screen 33
help files 33
menus and menu items 32
toolbar 33

V
Visual Studio 2003 16

W
Windows 158

X
XML forms 154