Developing Plugins and Applications

1/28/21 Adobe, Inc.

Contents

List of Examples

1 Introduction to Plugin Development

Developing Plug-ins and Applications provides details to C/C++ developers about plugin and Adobe PDF Library application development using the Acrobat SDK. It shows how your plugin can manipulate and enhance both the Acrobat and Adobe Reader user interface as well as the contents of underlying Adobe PDF documents. This guide also describes how to upgrade plugins across versions, provides platform-specific techniques for developing plugins, and lists the Acrobat SDK header files.

You can use the Acrobat SDK to create plugins for Adobe Reader and Acrobat as well as stand-alone applications that interact with and manipulate PDF documents. The Acrobat SDK contains two libraries: the Acrobat core API and the PDF Library API.

The Acrobat core API contains a set of interfaces that let you develop plugins that integrate with Acrobat and Adobe Reader. The PDF Library API lets you develop applications that interact with and manipulate PDF documents. It overlaps with the Acrobat core API (with the important exception of the AV-layer, which is only part of the Acrobat core API); however, the PDF Library API also extends the Acrobat core API with a small number of interfaces specific to the PDF Library API.

This chapter introduces the Acrobat core API and PDF Library API. The API descriptions appear in the Acrobat and PDF Library API Reference.

About plugins

A plugin is an application that uses the resources of Acrobat or Adobe Reader as a host environment. This means that a plugin does not require complex user interface elements. However, it must perform certain basic functions to let Adobe Reader or Acrobat know of its presence.

Plugins are dynamically-linked extensions to Acrobat or Adobe Reader and are written using the Acrobat core API, which is an ANSI C/C++ library. Plugins add custom functionality and are equivalent to dynamically-linked libraries (DLLs) on the Microsoft Windows platform; however, the plugin file name extension is .api, not .dll. On Mac OS, the file name extension of a plugin is `acroplugin`.

Acrobat and Adobe Reader plugins are grouped in the following categories:

- Regular plugins. Most plugins fall under this category.
- Reader-enabled plugins. Reader-enabled plugins access the limited set of APIs supported by Adobe Reader. These plugins are developed with permission from Adobe and require special processing to load under Adobe Reader. (See "Creating an Adobe Reader plugin" on page 279.)
- Certified plugins. Certified plugins have undergone extensive testing to ensure that they do not compromise the integrity of the Acrobat security model. There is currently no way for third party plugins to be certified by Adobe. Certified plugins are reserved for Adobe only.

  To ensure that only certified plugins are loaded with your installation of Acrobat or Adobe Reader, select the "Use only certified plugins" checkbox in the General panel of the Preferences dialog.

For security, DLLs cannot be loaded from subdirectories. Only DLLs located in the "plugins" folder will be loaded.

On the Mac OS platform, the third-party plugins should be installed in either of these directories:

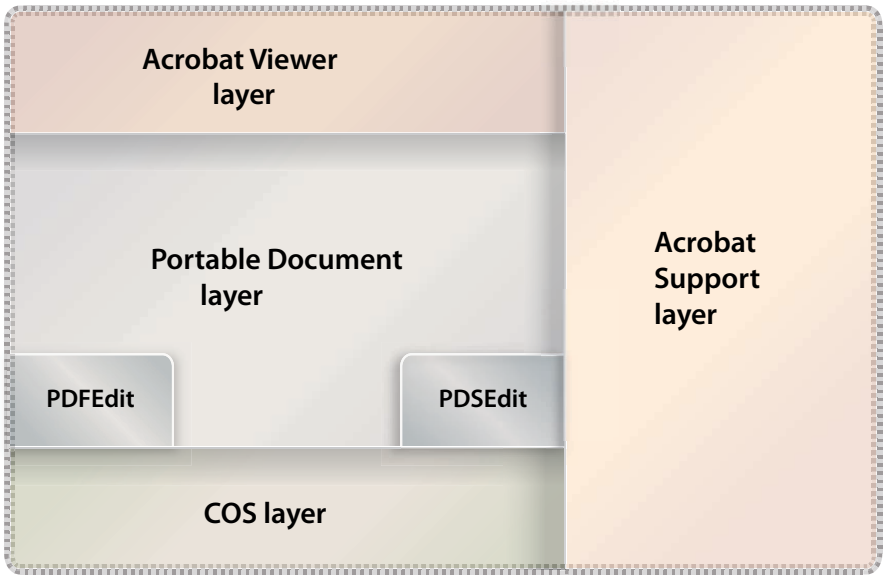
- ~/Library/Application Support/Adobe/Acrobat/(version)/Plugins

- Library/Application Support/Adobe/Acrobat/(version)/Plugins

About the Acrobat core API

The Acrobat core API consists of methods that operate on objects located within PDF documents. The Acrobat core API is implemented as a standard ANSI C programming library where methods are C functions and objects are opaque data types that encapsulate their data. The Acrobat core API is supported on Windows (32- and 64-bit) and Mac OS.

The following diagram illustrates the hierarchy of the Acrobat core API.



Acrobat Viewer layer

The Acrobat Viewer (AV) layer enables plugins to control Acrobat and modify its user interface. Using AV methods, you can, for example, add menus and menu commands, add buttons to toolbars, open and close files, display simple dialog boxes, and perform many other application-level tasks.

Note: AV layer methods are not available through the PDF Library API.

Portable Document layer

The Portable Document (PD) layer provides access to PDF document components such as pages and annotations. Closely related to the PD layer are two method groups, each of which controls a different aspect of a PDF document:

- PDFEdit methods deal with the physical representation of a PDF document. More specifically, PDFEdit methods handle page content as a list of objects whose values and attributes are modifiable. These methods allow your plugin or PDF Library application to read, write, edit, and create page contents and page resources, which may contain fonts, images, and so on.
- PDSEdit methods deal with the logical structure of a PDF document. A PDF document's logical structure is built independent of its physical representation, with pointers from the logical structure to the physical representation, and the reverse. PDSEdit methods store the logical structure information. They enable your plugin or PDF Library application to access PDF files by means of a structure tree.

Having logical structure in PDF files facilitates navigating, searching, and extracting data from PDF documents. For example, PDSEdit methods can obtain logically-ordered content, independently of the drawing order.

Acrobat Support layer

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

Cos layer

The Cos Object System (Cos) layer provides access to the building blocks used to construct PDF documents. Cos methods allow plugins and PDF Library applications to manipulate low-level data in a PDF file, such as dictionary and data streams. (See "Working with Cos Objects" on page 169.)

Platform-specific methods

In addition to the method groups represented in the previous diagram, the Acrobat core API includes platform-specific plugin utilities to handle issues that are unique to Windows and Mac OS. For information about these methods, see the Acrobat and PDF Library API Reference.

Acrobat core API objects

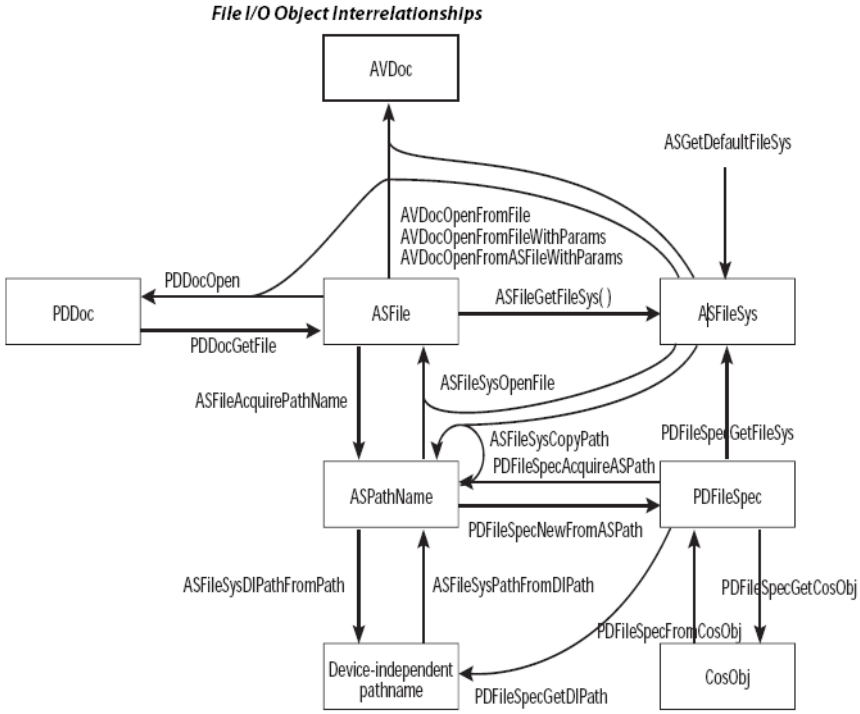
Most objects accessible through AV and PD layer methods are opaque. That is, they are neither pointers nor pointers to pointers. They provide equivalent functionality in that they reference an object's data rather than storing it. If you assign one object to another variable, both variables affect the same internal object.

Objects are typically named using the following conventions:

- The name of the concrete definition for a complex type ends in `Rec` (for record).
- A pointer to a simple or complex type ends in `P` (for pointer).
- Opaque types do not contain a `P` suffix. For example, a `PDDoc` object references a PDF document.
- Three names identify complex types that provide callback methods:
  - **Monitor**: A set of callbacks for an enumeration method.
  - **Server**: An implementation of a service added by a plugin.
  - **Handler**: An implementation for a subtype of object handled by a plugin
- Callback method names typically contain the suffix `Proc` (for procedure).

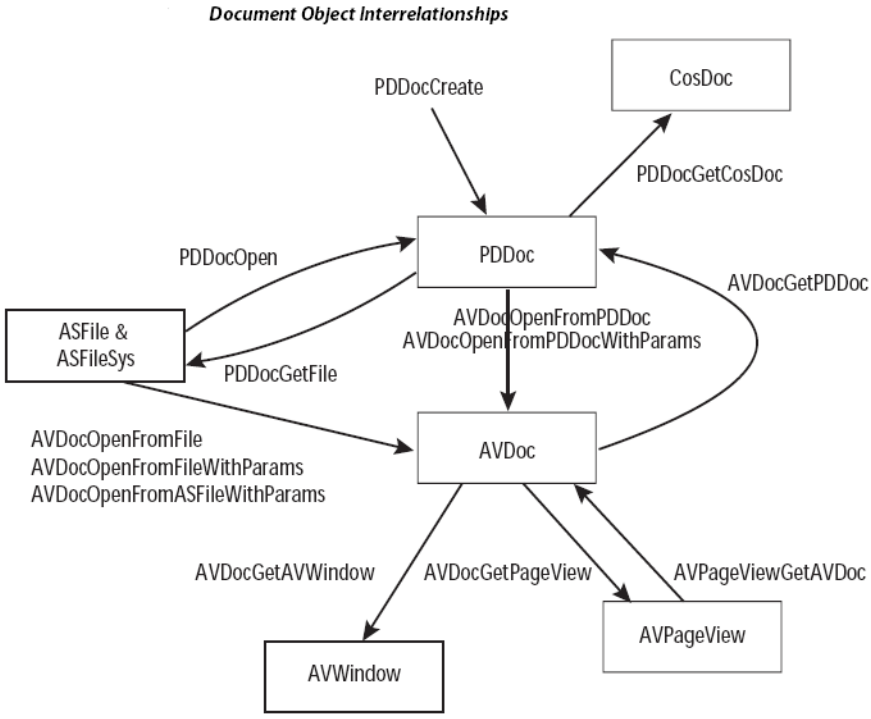
File object interrelationships

The following diagram shows file object interrelationships and how certain objects can be obtained by using other objects.



Document object interrelationships

The following diagram shows document object interrelationships and how certain objects can be obtained by using other objects.



Acrobat core API methods

Acrobat core API method names typically conform to the following syntax:

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

layer: identifies the method's layer (for example, AV for Acrobat Viewer layer).

object: identifies the object upon which the method acts (for example, menu).

verb: specifies an action that the method performs (for example, get or set). See the table that follows this list for the most commonly used verbs in method names.

thing: specific to each method, usually an object of the operation. May not always be present.

The following table lists some common verbs that are used in method names and describes their meaning.

Verb	Description
Acquire	Obtains a shareable resource from a parent object or increments a reference counter for an object. The shared object is not destroyed until all acquires have released it. Example: `AVMenuItemAcquire`
Add	Adds an object as a child to the current object. Example: `PDBookmarkAddChild`

Verb	Description
AddNew	Creates a new object using the specified parameters and adds the new object to the current object. Example: `PDBookmarkAddNewChild`
Close	Destroys an object that was opened and closes the underlying storage or stream. Example: `ASFileClose`
Create	Creates a new object of a given type. Example: `PDDocCreatePage`.
Delete	Removes the second object from the current object and destroys the second object. Example: `PDDocDeletePages`
Destroy	Destroys the specified object and releases its resources immediately. Example: `PDBookmarkDestroy`
Enum	Enumerates the specified descendant objects of the current object. Example: `PDDocEnumFonts`
Get	Retrieves a specific object attribute. Example: `AVWindowGetTitle`
Is	Retrieves a Boolean attribute of the object. Example: `PDBookmarkIsOpen`
New	Creates a new unattached object. Example: `AVMenuNew`
Open	Opens an object from storage or a stream. Example: `AVDocOpenFromFile`
Release	Releases a shared object. Example: `PDPageRelease`
Remove	Removes the second object from the current object but does not destroy it. Example: `AVMenuRemove`
Set	Sets an attribute of the object. Example: `PDAnnotSetFlags` Note: Cos methods uses the verb Put.

While many API method names follow the syntax specified in this section, there are exceptions. For example, conversion methods conform to the following syntax:

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

An example is the `AVPageViewPointToDevice` method. (See the Acrobat and PDF Library API Reference.)

Get and Set methods are used for getting and setting object attributes. Each object type has zero or more attributes. For example, an annotation object (`PDAnnot`) contains attributes such as color and date. You can obtain and modify attribute values by using methods such as `PDAnnotGetColor` and `PDAnnotSetDate`.

In some cases, the return value of a Get method is another object. For example, the `AVDocGetAVWindow` method returns an `AVWindow` object corresponding to the specified `AVDoc` object.

Other methods that return objects have the word Acquire in their name. These methods are always paired with a corresponding Release method, and have the additional side effect of incrementing or decrementing a reference count. The Acrobat core API uses `Acquire` and `Release` methods to perform various tasks such as determining whether it is safe to free a memory structure representing an object. Failure to match `Acquire` and `Release` method pairs can result in Acrobat complaining that a document cannot be closed due to non-zero reference counts. For more information, see "Acquiring and releasing objects" on page 34.

Data types

The Acrobat core API consists of the following data types:

- Scalar
- Simple
- Complex
- Opaque
- Cos

Scalar types

Scalar (non-pointer) types are based on underlying C language types, but have platform-independent sizes. They are defined in the header file CoreExpT.h. All scalar types are AS layer types. For portability, enumerated types are defined using a type of known size, such as `ASEnum16`. (See "Acrobat Support layer" on page 17.)

The following table describes scalar types.

Type	Byte Size	Description
`ASBool`	2	Boolean
`ASUns8`	1	unsigned char
`ASUns16`	2	unsigned short
`ASUns32`	4	unsigned long
`ASInt8`	1	char
`ASInt16`	2	signed short
`ASInt32`	4	signed long
`ASInt64`	8	signed long
`ASEnum8`	1	enum (127 values)
`ASEnum16`	2	enum (32767 values)

Type	Byte Size	Description
`ASFixed`	4	fixed point integer
`ASSize_t`	4	size of objects (as in size_t)

Simple types

Simple types represent abstractions such as a rectangle or matrix. These objects have fields that do not change. The following are examples of simple data types:

- `ASFixedRect`
- `ASFixedMatrix`
- `AVRect32`

Complex types

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

- To transfer a large number of parameters to or from a method. For example, the `PDFontGetMetrics` method returns font metrics by filling out a complex structure (`PDFontMetrics`).
- To define a data handler or server. For example, your plugin must provide a complex structure populated with callback methods (`AVAnnotHandlerRec`) when it registers an annotation handler.

Opaque types

Many methods hide the concrete C-language representation of data structures. Most methods accept an object and then perform an action on the object. Examples of opaque objects are `PDDoc` and `AVPageView` objects.

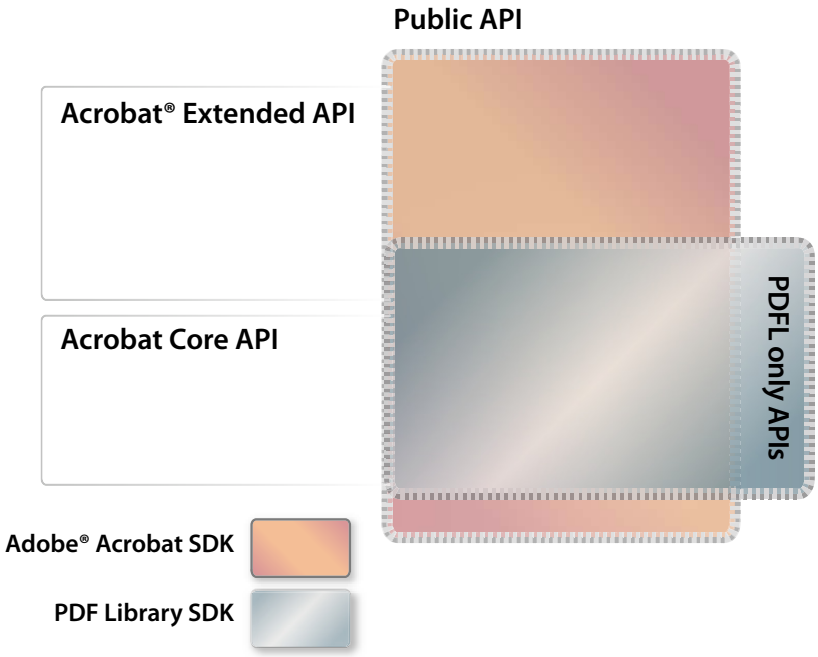
Cos types

A Cos object refers to its corresponding Cos object in the PDF document. Cos objects are represented as opaque 8-byte structures. They have subtypes of boolean, integer, real, name, string, array, dict, and stream. (See "Working with Cos Objects" on page 169.)

About PDF Library and plugin applications

The Acrobat core API and the PDF Library API let you create plugins and PDF Library applications to enhance and manipulate PDF document content and to customize Acrobat and Adobe Reader to meet your requirements. The PDF Library API is a subset of the Acrobat core API, with some additional functions that are available only in PDF Library.

The following diagram shows the relationship between the PDF Library API and the Acrobat core API.



For information about creating an Acrobat core API or project or PDF Library API, see "Creating Plugin and PDF Library Applications" on page 38.

Note: The remaining parts of this section describe tasks that you can perform by using either the Acrobat core API or the PDF Library API and refer you to the corresponding sections located in this guide.

Manipulating Acrobat and Adobe Reader

Plugins can control the Acrobat and Adobe Reader interface. For example, you can create and load new buttons. (See "Attaching a button to a toolbar" on page 96.)

Displaying a PDF document in an external window

Plugins can have Acrobat draw into an arbitrary window, allowing plugins to support PDF file viewing within their own user interface. For example, you can view a PDF document in an external window while Acrobat is displaying another PDF document. That way, you can view two separate PDF documents from within the same instance of Acrobat. (See "Opening a PDF document in an external window" on page 65.)

Indexed searching

Indexed searching enables you to catalog, index, search, and highlight text in PDF files. Simple sequential text searching may be too time consuming for long documents, and completely inadequate for searching a large collection of documents. (See "Working with Words" on page 124.)

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

Modifying file access

Plugins can provide their own file access procedures that read and write data when requested by the Acrobat core API. Using this capability, a plugin can enable PDF documents to be read from on-line systems, e–mail, document management, or database programs. (See "Accessing non-PDF files" on page 77.)

Creating new annotation types

Plugins can create their own annotation types, including any data they need. A custom annotation type can enable a user to draw (not just type) in an annotation, it can provide support for multiple fonts or text styles, or it can support annotations that can only be viewed by specific users. For example, you can use the Acrobat core API to create 3D annotations. (See "Creating 3D Annotations" on page 194.)

Dynamically adding text to PDF documents

You can use the Acrobat core API or the PDF Library API to dynamically modify a PDF document. For example, a plugin or PDF Library application can retrieve data from an enterprise database and insert the data into a PDF document. (See "Inserting Text into PDF Documents" on page 54.)

Understanding your target application

Both Acrobat and Adobe Reader accept plugins. Adobe Reader is designed predominantly for viewing and printing PDF documents. Acrobat Pro and Acrobat Pro Extended let you create PDF files, and offer advanced control over document exchange, review, and output. Acrobat Standard also lets you create PDF files and exchange and review comments.

Rights-enabled PDF documents

PDF documents that are *rights-enabled* can access specific functionality in Adobe Reader that would otherwise be unavailable. When a PDF document is rights-enabled, additional APIs become available for plugin development.

Adobe Reader plugins

Adobe Reader only accepts Reader-enabled plugins. (See "Creating an Adobe Reader plugin" on page 279.)

You may want your Reader-enabled plugin to access APIs that are available when the plugin 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 invoke these APIs only if your plugin is running with Acrobat. Failure to do so exposes the user to a variety of error messages. You can display a message to the user by invoking the `AVAlertNote` method. (See the Acrobat and PDF Library API Reference.)

If Adobe Reader attempts to load a plugin that is not Reader-enabled, Adobe Reader notifies the user that the plugin cannot function fully and then proceeds in one of these ways:

- Does not load the plugin.
- Omits toolbar buttons and menu items that enable editing.
- Displays dimmed toolbar buttons and menu items that enable editing.

Plugins that need to check whether or not they are running under Adobe Reader should do so as early in initialization as possible. Plugins 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.

Registering plugins for use by the plugin finder

You can register your plugin with Adobe to ensure that users are prompted to download and install it when they open PDF documents that require it.

Registering your plugin means that Adobe adds information about your plugin to a list of registered plugins. Each entry in this list associates a PDF dictionary extension and other characteristics with a URL from which users can download and install the plugin that processes that extension. Acrobat and Adobe Reader access the list over the web. The list is not publicly displayed on adobe.com. That is, it cannot be used to advertise the availability of your plugins. If you wish to post on adobe.com information about your plugin , see your Adobe representative.

PDF dictionary extensions

PDF documents can include dictionaries that are extensions to the current PDF specification. Such dictionary extensions have names of the form `prefix_propertyName` that associate it with a particular plug-in developer.

To avoid collisions over company names and company-specific extension names, Adobe (on behalf of ISO) maintains a prefix name registry. This registry is used to designate a 4-character, case-sensitive prefix that identifies a company or other entity. This prefix is used to create second-class names (dictionary or property names) of the form `MYCO_aPropertyName`.

2 Understanding Plugins

About plugin initialization

Plugin loading and initialization

When Acrobat or Adobe Reader is started, it searches the plugins directory (co-located with the Acrobat executable) to locate and load plugins. In addition, Acrobat or Adobe Reader searches folders that may be located within this folder. This search goes one level deep.

Acrobat and Adobe Reader display a progress message in the bottom line of the splash screen at start-up. As each plugin is loaded, the progress message shows the plugin name. No plugins are loaded if the Shift key is held down while Acrobat or Adobe Reader start. Also, if Acrobat or Adobe Reader are running in certified mode, no third-party plugins are loaded.

When creating methods for your plugin, keep the following rule in mind:

Implement a `PluginUnload` procedure: This procedure frees allocated memory. This routine is invoked if any of the initialization routines returns `false`. Under normal conditions, this procedure is not invoked until the user closes Acrobat or Adobe Reader.

Handshaking

Acrobat and Adobe Reader perform a handshake with each plugin as it is opened and loaded. During handshaking, the plugin specifies its name, several initialization procedures, and an optional unload procedure.

A plugin must implement the following handshaking function:

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

During handshaking, the plugin receives the `hsData` data structure (defined in the PIVersn.h file). Acrobat and Adobe Reader convert all function pointers that are passed in this data structure into callbacks using the `ASCallbackCreateProto` method. For information about this method, see the Acrobat and PDF Library API Reference.

The DUCallbacks.h header file declares all callback methods that must be located in your plugin. The following shows the function signatures of these callback methods:

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

All callbacks return `true` if your plugin's procedure completes successfully or if the callbacks are optional and are not implemented. If your plugin's procedure fails, it returns `false`. If either Acrobat, Adobe Reader, or a plug-in aborts handshaking, Acrobat or Adobe Reader displays an alert dialog box showing a brief explanation before loading other plugins. At minimum, a plugin must implement the `PluginInit` callback.

To ensure your plugin does not hinder Acrobat startup, you must limit code executed in your handshake functions to the minimum.

Note: The handshaking function is located in the PIMain.c file. This source code located in this file is functional and must not be modified.

The following example shows how a plugin's `PIHandshake` method specifies the plugin callbacks provided during handshake and initialization. The tasks performed by each function is described in the next sections. For general information about creating callback functions, see Using callback functions.

```
ACCB1 ASBool ACCB2 PIHandshake(Uns32 handshakeVersion, void *handshakeData)
{
   if(handshakeVersion == HANDSHAKE_V0200)
   {
      PIHandshakeData_V0200 *hsData = (PIHandshakeData_V0200 *)handshakeData;
      hsData->extensionName = GetExtensionName();
      hsData->exportHFTsCallback =
         (void*)ASCallbackCreateProto(PIExportHFTsProcType,
           &PluginExportHFTs);
      hsData->importReplaceAndRegisterCallback =
         (void*)ASCallbackCreateProto(PIImportReplaceAndRegisterProcType,
           &PluginImportReplaceAndRegister);
      hsData->initCallback =
         (void*)ASCallbackCreateProto(PIInitProcType, &PluginLoad);
      hsData->unloadCallback =
         (void*)ASCallbackCreateProto(PIUnloadProcType, &PluginUnload);
      return true;
   }
   return false;
}
```

Exporting HFTs

A Host Function Table (HFT) is the mechanism through which plugins invoke methods in Adobe Reader or Acrobat, as well as in other plugins. After Acrobat finishes handshaking with all the plugins, it invokes each plug-in's `PluginExportHFTs` callback procedure.

In the `PluginExportHFTs` procedure, a plugin may export any HFTs it intends to make available to other plugins. This callback should only export an HFT, not invoke other Acrobat core API methods. (See "Working with Host Function Tables" on page 156.)

Note: The only time a plugin can export an HFT is during execution of its `PluginExportHFTs` procedure.

Importing HFTs and registering for notifications

After Acrobat or Adobe Reader completes invoking each plugin's `PluginExportHFTs` callback method, it invokes each plugin's `PluginImportReplaceAndRegister` callback method. In this method, plugins perform three tasks:

1. Import any special HFTs they use (the standard Acrobat HFTs are automatically imported). Plugins also may import HFTs any time after this while the plugin is running.

2. Register for notifications by using the `AVAppRegisterNotification` method. Plugins also may register and unregister for notifications while the plugin is running. A plugin may receive a notification any time after it has registered for it, even if the plugin's initialization callback has not yet been called. This can occur if another plugin initializes first and performs an operation, such as creating a PDF document, which causes a notification to be sent. Plugins must be prepared to correctly handle notifications as soon as they register for them.

3. Replace any of the Acrobat API's replaceable HFT methods. (See "Replacing HFT methods" on page 165.)

Note: The only time a plugin may import an HFT or replace a standard API method is within its `PluginExportHFTs` callback procedure. Plugins may register for notifications at this time or any time afterward.

Initialization

After Acrobat or Adobe Reader completes calling each plugin's `PluginImportReplaceAndRegister` callback method, it invokes each plugin's `PluginInit` procedure. Plugins can 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 plugin is running.

When creating the initialization portion of a plugin, keep the following rules in mind:

Avoid creating dialog boxes: Do not create a dialog box in your plugin's initialization or do anything else that may interfere with the successful startup of Acrobat or Adobe Reader. The application may be started by using an interapplication communication (IAC) event, in which case a user would not be present to respond to your dialog box.

Avoid invoking methods that cause Acrobat to load non-critical components: Here are actions your plugin should avoid, listed in order of importance:

- Avoid invoking JavaScript, which loads the EScript plugin and starts the JavaScript engine.
- Avoid invoking functions referenced from HFTs exported by Adobe plugins. Adobe plugins are not fully initialized until they are invoked or otherwise triggered.

  Non-Adobe plugins are fully initialized during Acrobat startup. Invoking functions declared in HFTs exported by non-Adobe plugins will not significantly delay Acrobat startup, unless those plugins violate the principles listed here.
- Avoid invoking system methods that load more system libraries, such as accessing the disk.

Do create your menu items: Creating your menu items during initialization ensures that your menu items will be available in any PDF document opened using Acrobat or Adobe Reader.

If your plugin must carry out a task after all plugins are initialized, it should register for the `AVAppDidInitialize` notification. This notification is invoked when Acrobat finishes initializing and is about to enter its event loop.

Unloading

A plugin's `PluginUnload` procedure should free any memory the plugin allocated and remove any user interface changes it made. Acrobat invokes this procedure when it terminates or when any of the other handshaking callbacks return `false`. This function should perform the following tasks:

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

Currently, plugins unload only when Acrobat exits.

Summarizing a plugin's life cycle

The following steps describe the life cycle of a plugin:

1. At startup, Adobe Reader or Acrobat searches its plugin directory for plugin files.
2. For each plugin file, Adobe Reader or Acrobat attempts to load the file. If the plugin is successfully loaded, Adobe Reader or Acrobat invokes routines in PIMain.c that complete the handshaking process.
3. Adobe Reader or Acrobat invokes callback functions in this order:
   - `PluginExportHFTs`
   - `PluginImportReplaceAndRegister`
   - `PluginInit`

This sequence establishes the linkages between the plugin and Acrobat or Adobe Reader, and between the plugin and any other plugins. After all plugins are loaded, Acrobat or Adobe Reader continues its own loading and starts the user interface. It adds plugin tools to the toolbar, and plugin menu items to the menu bar. Then it starts the user session.

Using callback functions

Acrobat or Adobe Reader invokes callback functions that you define to perform a specific task. For example, when a user clicks a button located on a toolbar, a callback method is invoked. (See "Creating toolbar button callback functions" on page 98.)

To create a callback function, you can invoke the `ASCallbackCreateProto`, `ASCallbackCreateReplacement`, and `ASCallbackCreateNotification` methods to convert functions into callback functions and to perform type checking. This enables the compiler to determine whether the correct prototypes are used for a specific callback function. For information about these methods, see the Acrobat and PDF Library API Reference.

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 plugin.

The following code example shows the syntax to create a callback function:

```
AVExecuteProc ExecProcPtr = NULL;
ExecProcPtr= ASCallbackCreateProto(AVExecuteProc, &ShowMessage);
```

The `ASCallbackCreateProto` macro returns a callback of the specified type that invokes the user-defined function whose address is passed as the second argument. In this example, the `ShowMessage` function is converted to a callback function (the `ShowMessage` function is a user-defined function that is invoked when a specific action occurs).

The `ASCallbackCreateProto` macro returns a pointer to a function that can be invoked by a plugin or by Acrobat or Adobe Reader. Use the `ASCallbackDestroy` method to dispose of a callback that is no longer required.

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:

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

Notifications

The Acrobat core API provides a notification mechanism so that plugins can synchronize their actions with Acrobat or Adobe Reader. Notifications enable a plugin to indicate that it has an interest in a specified event, such as an annotation being modified, and to provide a procedure that Acrobat invokes each time that event occurs. (See "Registering for Event Notifications" on page 143.)

Handling events

You can use the Acrobat core API to handle various types of events.

Mouse clicks

Mouse clicks are passed to any procedure registered using the `AVAppRegisterForPageViewClicks` method. 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.

You can query the state of the mouse buttons in a manner appropriate for drag operations by invoking the `AVSysTrackMouse` method. (See the Acrobat and PDF Library API Reference.)

Adjust cursor

Adjust cursor events are passed to any procedures registered using the `AVAppRegisterForPageViewAdjustCursor` method. 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.

Using plugin prefixes

It is important to correctly name all items located in your plugin, such as HFTs, menus, toolbars, and so on, to ensure they function properly. Failure to do so may cause your plugin to produce unpredictable results when your plugin collides with a plugin of another developer who used the same names.

Prevent spaces from being used in tokens you intend to use as names in a PDF file. This can happen, for example, if you allow a user to type a name into the PDF file and your plugin does not check the input before writing it to the file.

Obtaining a developer PDF name prefix

In support of ISO 32000, Adobe maintains a list of the prefixes of second class names for PDF. It is freely open to all developers and organizations following the specifications given in Annex E of ISO 32000. For example, Adobe uses ADBE or Acro. This chapter uses ADBE in the examples. Your plugins will use your unique PDF name. Register your name at https://github.com/adobe/pdf-names-list.

Note: Registering a PDF name ensures that you name is unique. It is up to you to ensure that names are unique among all plugins you or your company write (for example, you must ensure that two of your plugins do not both use ADBE_save as a menu item name).

Using a developer prefix

Every plugin must use the prefix to name its various elements as well as private data it writes into PDF documents. The following sections describe and provide an example of each element that must use a prefix.

Plugin name

`ExtensionName`, used in plugin handshaking, must use the following syntax: `Prefix_PluginName`.

```
hsData->extensionName = ASAtomFromString("ADBE_SuperCrop");
```

Menu prefixes

Menu names must use the following syntax: `Prefix:MenuName`.

```
SuperCropMenu = AVMenuNew(SuperCrop, "ADBE:SuperCropMenu", gExtensionID);
```

For information about invoking the `AVMenuNew` method, see "Adding menu commands to menus" on page 83.

Menu items prefixes

Menu item names must use the following syntax: `Prefix:MenuItem`.

```
SelSuperCropTool = AVMenuItemNew(SuperCrop, "ADBE:SuperCropMItem", NULL,
false, '9', AV_OPTION, SuperCropIcon, gExtensionID);
```

Tool prefixes

Tools names must use the following syntax: `Prefix:Tool`.

```
static ACCB1 ASAtom ACCB2 SuperCropToolGetType(AVTool tool)
{
   return ASAtomFromString("ADBE:SuperCropTool");
}
```

Toolbar button prefixes

Toolbar buttons must use the following syntax: `Prefix:ToolbarButton`.

```
myButton = AVToolButtonNew(ASAtomFromString("ADBE:HelloWorld"), (void *)
myBM, false, false);
```

For information about creating a toolbar button, see "Creating toolbar buttons" on page 92.

Private data identified using second-class names

If you add private data to keys defined in a Cos dictionary and you want the document to be readable to others outside your company, you must identify the data using second-class names. Such key names use the syntax *`Prefix_PrivateDataName`*, where *`Prefix`* identifies the company or other entity that introduces the key, and *`PrivateDataName`* identifies the data.

When adding 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. Therefore, there is no need to use a prefix for the latter two keys.

/ACME_aName << /First 2 /Second << /Third [2 3] >> >>

Note: For information about Cos dictionaries, such as the one shown in the previous example, see "Working with Cos dictionaries" on page 177.

Action prefixes

Actions must use the following syntax: `Prefix_ActionName`.

```
AVAppRegisterActionHandler(&BkmkHandler, NULL, "ADBE_HWAction", "HWAct");
```

Annotation prefixes

Annotation prefixes must use the following syntax: `Prefix_AnnotType`.

```
return(ASAtomFromString("ADBE_MarkUpAnnot");
```

HFT prefixes

When your plugin exposes any HFTs of its own, it must use an HFT name that conforms to the following syntax: `Prefix_HFTName`.

```
gDebuggWinHFTServer =
HFTServerNew("ADBE_DebugWin",provideDebugWinHFTCallback, NULL, NULL);
```

For information about HFTs, see "Working with Host Function Tables" on page 156.

Modifying the Acrobat or Adobe Reader user interface

This section describes typical operations that a plugin can perform to modify the Acrobat or Adobe Reader user interface. To modify the user interface, you must invoke methods that belong to the Acrobat Viewer Layer. As a result, you cannot modify the Acrobat or Adobe Reader user interface by using the PDF Library API. (See "Acrobat Viewer layer" on page 16.)

Adding or removing menus and menu items

You can use the Acrobat core API to add new menus and add commands to existing menus. You can also remove a menu or a menu command.

Menu commands can have shortcuts (keyboard accelerators). Acrobat and Adobe Reader do not ensure that plugins add unique shortcuts, but it is possible to programmatic-ally check which shortcuts are already in use before adding them.

You are encouraged to have your plugin add its menu commands to the Tools menu. When the plugin starts, Acrobat or Adobe Reader automatically adds this menu, as well as the About Plugins and Plugin Help menus. After Acrobat or Adobe Reader loads all plugins, it checks these three menus and removes any that are empty. (See "Creating Menus and Menu Commands" on page 82.)

Modifying toolbars

You can add new buttons to the toolbar, although the size and resolution of the user's monitor can limit the number of tool buttons that are displayed. You can also remove buttons from an existing toolbar. (See "Creating Toolbars and Buttons" on page 90.)

Controlling the About box and splash screen

You can set values in the preferences file by invoking the `AVAppSetPreference` method to prevent the Acrobat or Adobe Reader About box or splash screen from appearing before displaying the first document. These changes take effect the next time Acrobat or Adobe Reader is started.

About Adobe Plugins is a standard menu command in the Help menu. This menu command contains a submenu. You are encouraged to have your plugin add a menu command to the submenu to bring up its own About box.

Creating help files

The Help directory that accompanies Acrobat or Adobe Reader provides a standard location for your plug-in help files. You can place a help file either in the Help directory or in a subdirectory of the Help directory. If, for example, your plugin is localized for Japanese, you may want to place its Japanese help file in a Help_JPN subdirectory. To help open locale-specific help files, the Acrobat core API provides the `AVAppOpenHelpFile` method. (See the Acrobat and PDF Library API Reference.)

User interface guidelines

Follow these guidelines when modifying the Acrobat or Adobe Reader user interface:

- During time-consuming operations, provide feedback to the user by using features such as progress monitors, cancel button, hourglass cursor, 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. 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.
- Invoke the `AVToolBarIsRoomFor` method to determine if there is room on the toolbar to accommodate a new button. If there is not sufficient room, then do not attach the button to the toolbar. Because space is limited, add a menu command for each button you add to a toolbar. This provides a way for users to access a plugin's functionality, and also enables users to access functionality by using a shortcut key (if the menu command contains one).

Acquiring and releasing objects

Ensure that calls to `Acquire` and `Release` methods match. Objects obtained by `Acquire` methods must ultimately be released. If they are not released, Acrobat or Adobe Reader raises exceptions when a non-zero reference count is discovered. An exception can occur when Acrobat or Adobe Reader quits or when a document is closed.

When allocating memory to objects, follow these guidelines:

- Use `ASmalloc` and `ASfree` instead of `malloc` and `free`, or you risk memory leaks.
- Ensure that `ASmalloc` and `ASfree` pairs match or you will create memory leaks.
- Use `ASmalloc` to allocate memory for methods that state that Acrobat or Adobe Reader free the memory for you.
- 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.

If you use an `Acquire` method to obtain an object, you must subsequently use a `Release` method to correctly update the reference counter, as shown in the following example:

```
PDDoc doc;
PDPage page;

//Acquire a page from a PDF document
doc = PDDocOpenFromASFile("myPDF.pdf", null, true);
page = PDDocAcquirePage(doc, 42);

//Perform a task using the page

//Release the page
PDPageRelease (page);
```

Notice that the `PDPage` object is acquired by invoking the `PDDocAcquirePage` method and is released by invoking the `PDPageRelease` method. For information about working with pages, see "Working with Page Views and Contents" on page 116.

Debugging plugins

When debugging your plugin, consider the following points:

- Include the `#define DEBUG 1` statement to ensure that parameter type checking is performed by macros such as `ASCallbackCreateProto` and to enable the debug exception-handling macros.
- The `AVSysBeep` method provides a simple way to add an audible indication that a certain point has been reached in a plugin's code. Likewise, the `AVAlertNote` method displays a message box that indicates whether a certain point of code is 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.

Page view layers

Acrobat and Adobe Reader drawing and mouse click processing rely 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.

Layer	Item
`0`	Page contents
`LINK_LAYER (1)`	Links
`NOTE_LAYER (3)`	Annotations, open and closed. Open annotations are drawn above closed annotations.

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 annotations, and finally open annotations. As a result, open annotations are drawn over any closed annotations that 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 "Tool callbacks" on page 139.)

Annotation handlers provided by plugins can reside in any layer. For example, a plugin 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. 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 invoke `AVAnnotHandlerGetLayerProc` callbacks for changes in value, so be sure to invalidate the page rectangle of an annotation when its layer changes.

For information about page views, see "Working with Page Views and Contents" on page 116.

Minimizing screen redrawing

Minimize screen redrawing by using the `AVPageViewBeginOperation` and `AVPageViewEndOperation` methods 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 invoking the `AVPageViewBeginOperation` method before the sequence and the `AVPageViewEndOperation` method after it, only one redraw occurs.

Storing private data in PDF files

Plugins can store private data in PDF files, although private data must be stored in such a way that the file can still be drawn by Acrobat. Adobe maintains a registry of private PDF dictionary key names to reduce the possibility of a plugin's key names conflicting with names belonging to other plugins. For information about dictionaries, see "Working with Cos dictionaries" on page 177.

Private dictionary keys exist in the following categories:

- Specific keys that are proposed by third parties but are generally useful. Adobe maintains a registry of these names.
- Keys registered by third parties as well as keys whose prefix is registered that are applicable only to a limited set of users. Adobe maintains a registry of these names and prefixes.
- Keys that begin with a special prefix reserved by Adobe for private extensions. These keys are intended for use in files that are never seen by other third parties, since these keys may conflict with keys defined by others.

Exporting data from PDF document objects

Using the Acrobat core API, you can export data from PDF document objects to XML files. Object data contain property-value pairs. For example, consider a PDF document object that contains the following XML elements:

Node	Name
nodeTag	xyz_Node
nodeNameTag	xyz_NodeName
propTag	xyz_Property
propNameTag	xyz_PropertyName
propValTag	xyz_Value

After you export the data from this object into an XML file, the data would appear as shown in the following diagram.

```
<xyz_Node xyz_NodeName="blue bit">
        <xyz_Property xyz_PropertyName="Part Number" xyz_Value="1234567890"/>
        <xyz_Property xyz_PropertyName="Cost" xyz_Value="$4.99"/>
        <xyz_Property xyz_PropertyName="Qty In Stock" xyz_Value="7"/>
        <xyz_Node xyz_NodeName="blue bit subpart A">
                <xyz_Property xyz_PropertyName="Part Number" xyz_Value="1234567890-A"/>
        </xyz_Node>
</xyz_Node>
```

To retrieve data from a PDF document object, invoke the `PDDocExportUserProperties` method and pass the following arguments:

- A `PDDoc` object that represents a PDF document that contains the object from which data is extracted. (See "Creating a PDDoc object" on page 76.)

- A `PDSElement` instance that represents PDF structural elements.
- An `ASStm` object that represents XML content converted from information from labels.
- An `ASBool` value that specifies whether to save object data of the specified element (`false`) or the whole subtree (`true`).
- An `ASBool` value that specifies whether to include hidden content of the element.
- An instance of the `PDUserPropertiesXMLLabels` data structure that specifies information for converting object data to XML. For information about this data structure, see the Acrobat and PDF Library API Reference.

3 Creating Plugin and PDF Library Applications

Use the Acrobat SDK and the Adobe PDF Library SDK to create plugin applications as well as stand-alone applications that interact with PDF documents.

Supported environments

The following table specifies the supported platforms, operating systems, and compilers for Acrobat SDK and PDF Library SDK development.

Platform	Processor	Operating system	Compiler
Windows 32-bit and 64-bit	Intel	Windows 7, 8, and 10 (32-bit and 64-bit);	Visual Studio 2019 Service Pack 1
Mac 32-bit and 64-bit	Intel	Mac OS X v. 10.11	XCode 9.2

Note: While it may be possible to use the Acrobat SDK and the PDF Library SDK in other development environments, such use is not supported. The project files for the sample applications are created and supported only in the listed compiler versions.

Working with platform-specific techniques

The Acrobat API is nearly platform-independent. By using the memory allocation and file system APIs provided by Acrobat or Adobe Reader, many parts of a plugin are highly portable across platforms. While this chapter contains platform-specific development information for Windows and Mac, the guidelines here should help you plugins more portable among the various supported platforms.

About platform-dependent data

The following are platform-specific data types that appear explicitly in the Acrobat core API:

platform data structures: Data structures such as the Win32 data structure that represents a window.

platform path values: The data structure that represents the path to a file.

platform event: The data structure that represents mouse clicks, key presses, window activation, and so on.

Return value: 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 plugins:

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 let you display adjacent to a menu item.

Menu items: Remember that not all Acrobat or Adobe Reader implementations have the same menu items.

Portability techniques

The following techniques can improve your plugin'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`, `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 plugin, so that you can easily recognize and rewrite platform-dependent sections.

Windows techniques

Developing Windows plugins

You can put your plugins in the default Acrobat plugin location the plugins folder (in the same directory as the Acrobat executable).

You are encouraged to use the plugin samples BasicUI and Starter as a basis for developing plugins. These samples have all of the appropriate project settings. The Starter sample only builds a loadable plugin while the BasicUI sample adds menu items.

Locating and loading plugins

When Acrobat or Adobe Reader starts, it scans the plugins 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 plugins in folders. When it locates a file with the extension .API, it looks for the `PlugInMain` exported symbol, which specifies the entry point for your plugin. Acrobat or Adobe Reader loads the plugin by invoking the `LoadLibrary` function and then calls the function referenced by the `PlugInMain` symbol.

The `LoadLibrary` function calls your plugin's `DLLMain` entry point with the parameter `DLL_PROCESS_ATTACH` passed. Your plugin can run some initialization code in DLLMain, such as allocating memory, before its `PluginMain` function is called by Acrobat or Adobe Reader.

If you allocate memory in your plugin's `DLLMain` entry point, it must deallocate that memory when `DLLMain` is called with `DLL_PROCESS_DETACH`. If your plugin relies on its implementation of the `PluginUnload` function to deallocate memory, it could fail if Acrobat or Adobe Reader unloads the plugin immediately without calling the plugin's handshaking callbacks. This would happen in the following situations:

- If the plugin is not Adobe-certified and the user has specified the Certified Plugins Only option in the Preferences settings.
- If the plugin is running under Adobe Reader, but it is not enabled for Adobe Reader. This could potentially cause a crash when Acrobat or Adobe Reader closes.

Why a plugin might not load

There are several reasons why a plugin may not load successfully:

- The plugin's filename extension was not changed from .dll to .api.
- Too many plugins are being loaded by either Acrobat or Adobe Reader. The number of plugins that can be loaded at any one time depends on the code generation settings of all loaded plugins.
- The plugin attempts to register with the same `extensionName` as another plugin 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 plugin 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 plugin compiled and linked without errors. Generally, rebuilding the plugin 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 plugin to be loaded, it must export the symbol `PlugInMain`. This task can be accomplished by including a .def file in the project for the plugin or by including the line `/EXPORT:PlugInMain` in the project settings for the plugin. If you are developing an Adobe Reader plugin, you also must define a macro to access HFTs available to Adobe Reader. (See "Creating an Adobe Reader plugin" on page 279.)

Interapplication communication

Plugins 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.

A plugin cannot 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 plugins must be implemented using the ActiveX Template Library. Plugins should use the DDEML library to handle DDE messages. Problems may arise if they do not.

Debugging

Generally, the debugger built into Visual C++ is adequate to debug plugins. Debugging a Windows plugin compiled with Visual C++ is quite simple if you remember a few things:

- Specify the Acrobat plugin 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 plugin, 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 Microsoft Visual Studio start Acrobat or Adobe Reader.

Two common reasons why breakpoints may not be hit are:

- You started Acrobat or Adobe Reader from the File Manager or Program Manager. Acrobat or Adobe Reader must be started from within Microsoft Visual Studio to debug plugins.
- You copied your plugin into Acrobat's plugin directory, instead of specifying the plugin directory in the Link | Output dialog box.

Handling the thread local storage (TLS) limit

There is a limit to the number of plugins that Acrobat or Adobe Reader can load at any given time. This is due to a limitation of the multi-threading model used by the Win32 API and is dependent on the code generation settings of the plugins being loaded.

The following information can help maximize the ability of Acrobat and Adobe Reader to load plugins.

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/plugin that uses thread local storage is allocated at least one slot when loaded using LoadLibrary. This includes system DLLs, plugins, 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.

The following guidelines will minimize the TLS slots occupied by plugins:

- Plugins that are not multi-threaded should only link with the single-threaded run-time libraries that do not occupy a TLS slot.
- If your plugin is multi-threaded, consider linking it with the multi-threaded DLL run-time library. Both the DLL and static versions of the run-time libraries occupy a TLS slot. However, many plugins shipped with Acrobat or Adobe Reader use the DLL version so the run-time DLL does not occupy another TLS slot after it is loaded by the process.

Note: Acrobat and Adobe Reader do not currently generate an error when a plugin fails to load due to the TLS limit.

Using modal dialog boxes

If you write plugins that contain modal dialog boxes on the Windows platform, you need to perform the following steps:

1. When you are creating your dialog box, get the parent `HWND` of the dialog box using the `WinAppGetModalParent` method. Then use this `HWND` when creating the dialog box.

   Ensure that you 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.

   ```
   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 box 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 box is destroyed. Then pass the `AVWindow` to the `AVAppBeginModal` method.

   Here is code that is called in response to a `WM_INITDIALOG` message:

   ```
   static AVWindow sAVWin;
   .....
   // hWnd is the window handle of the dialog box window
   sAVWin = AVWindowNewFromPlatformThing(AVWLmodal, 0, NULL,
   gExtensionID, hWnd);
   AVAppBeginModal(sAVWin);
   ```

3. At the time the dialog box is destroyed, 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`. Here is a section of code called in response to a `WM_DESTROY` message:

   ```
   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.

Mac OS techniques

Developing a Mac OS plugin

Apple Xcode 9.2is the currently-supported development environment for developing plugins. Apple developer tools contain the correct frameworks and libraries in addition to extensive documentation on making plugins (and applications) Mach-O and Carbon compliant.

With macOS 10.14.5 and macOS 10.15 Catalina, Apple has mandated notarization of all applications. Conforming to this requirement, October 2019 updates for both Adobe Acrobat and Adobe Acrobat Reader applications on DC and 2017 tracks are notarized. Adobe recommends that third-party plug-in developers should get their plug-ins notarized by Apple. Without notarization, your plug-ins will fail to load in Adobe Acrobat and Adobe Acrobat Reader on macOS 10.14.5 and above.

For more information on the Apple Notarization process, see Notarizing Your App Before Distribution.

Note: Acrobat SDK samples are built against the MacOSX10.11 as universal binaries.

Using the samples

You are encouraged to use the Starter plugin sample as a basis for developing your plugins. This sample contains the appropriate project settings as defined in the supplied Xcode project configuration files. The Starter sample does nothing other than build a loadable plugin. In addition, other plugins that could be useful as a starting point for developing plugins are available.

The Info.plist file contains a list of properties used by the package. Adobe provides a common info.plist file. It uses project settings to define properties appropriately for each plugin.

Establishing Carbon or Cocoa compliance

Carbon and Cocoa are application environments of the Mac OS X operating system. Each includes programming interfaces that include header files, a library, and a runtime.

Acrobat 9.0 uses property lists (Info.plists), which are stored with the executable files and resources that make up an application, known as an application bundle. For more information about converting existing Mac OS applications into Carbon, see http://developer.apple.com/documentation/Carbon/Conceptual/carbon_porting_guide/.

Note: To prevent problems with older style event handling, plugins must replace calls to `WaitNextEvent` with calls to `RunCurrentEventQueue` or `ReceiveNextEvent`.

Xcode configuration files

Mac OS plugin sample build settings are defined in SDK and project-level configuration files and not within the projects themselves. Xcode configuration files include lists of build settings definitions that can be applied to multiple projects and/or multiple targets.

The configuration files and settings have a hierarchical structure modeled after Apple Developer documentation located at the following URL:

http://developer.apple.com/documentation/DeveloperTools/Conceptual/XcodeUserGuide/Contents/Resources/en.lproj/05_05_build_configs/chapter_33_section_6.html

Each project is based on a project-level build settings file(s) that includes SDK-level settings.

At the SDK level, there are separate configuration files for SDK plugin settings (Default.xcconfig), environmental variables (Environment.xcconfig) and resource settings (Resources.xcconfig). Global target settings for _debug and _release targets are stored in Debug.xcconfig and Release.xcconfig, respectively.

At the project level, there are four configuration files:

- ProjectDefault.xcconfig
- ProjectResources.xcconfig
- Project_debug.xcconfig
- Project_release.xcconfig.

Each project level configuration file includes the settings from its related (parent) SDK configuration file (for example, ProjectDefault.xconfig includes Default.xcconfig and ProjectResources.xcconfig includes Resources.xcconfig). Generally, SDK-level setting definitions are not included directly, but rather are included through project-level configuration files.

Each SDK plugin project includes a single (Default) configuration based on the ProjectDefault.xcconfig build settings which include the SDK-level Default.xcconfig build settings. Each project has two targets: a

_debug target and a _release target. The targets' build settings are based on Project_debug.xcconfig and Project_release.xconfig, respectively. Similar to the project configuration files, each target configuration settings include its parent SDK configuration file; for instance, Project_debug includes Debug.xcconfig settings.

Project-level configuration files whose names begins with Project are the default project settings included with most SDK plugin samples. Project-level configuration files that are prefixed with a specific sample's name include settings specific to that sample. The build settings for most SDK projects are extremely similar with most definitions residing in the SDK configuration files.

Using SetGWorld rather than SetPort

With the move to carbonization and double buffering, you should use GetGWorld rather than the toolbox call SetPort. Using both calls in the same plugin can cause the current port to get out of sync with the current device. Using only GetGWorld 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. The following code is an example.

```
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 plugins

When Acrobat or Adobe Reader starts, it scans the plugin folder to locate and load plugins with the acroplugin file extension. PowerPC plugins must have creator `CARO` (CFBundleSignature) and type `XTND` (CFBundlePackageType). Each plugin exports a single main entry point, `AcroPluginMain`. When loading a plugin, Acrobat or Adobe Reader jumps to the plugin's entry point to begin handshaking. (See "Handshaking" on page 26.)

Using memory

The Acrobat or Adobe Reader memory allocator gets its memory from the system and not from the application's memory partition. (See "Acquiring and releasing objects" on page 34.)

Memory allocation guidelines are particularly important in Mac OS to ensure that memory is allocated from the system rather than from the application partition. Otherwise, your plugin is very likely to cause Acrobat or Adobe Reader to run out of memory.

Resource file considerations

Acrobat or Adobe Reader open a plugin's resource file with read-only permissions. In addition, plugins cannot assume that their resource file is on top of the resource chain each time they are entered by using

an `ASCallback`. Plugins 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 invokes `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 plugin 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 plugin 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 plugin'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 PICommon.h.

Macros

The following macros must be defined:

- `POWER_PC` must be defined
- `PLATFORM` must be defined as MacPlatform.h
- `PRODUCT` must be defined as Plugin.h

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.

Mac OS-only methods

Plugins should not use the `ASPathFromPlatformPath` method in Mac OS. Instead, they should invoke `ASFileSysCreatePathName`. The `AVAppDidOrWillSwitchForDialog` method is only useful to plugins in Mac OS.

Interapplication communication

Plugins can add their own Apple events to those supported by Acrobat or Adobe Reader by hooking into the Apple event handling loop for Acrobat or Adobe Reader. This is done by replacing the

`AVAppHandleAppleEvent` method in the API. If the plugin receives an Apple event it does not want to handle, it should invoke the implementation of the method it replaced, allowing other plugins or Acrobat or Adobe Reader the opportunity to handle the Apple event.

Creating a sample plugin

When you start a new Acrobat plugin for the Windows platform, it is recommended that you use the Starter sample plugin as a starting point. On Windows, the project file is named Starter.sln and can be found in the following directories:

C:\Acrobat SDK\PluginSupport\Samples\Starter\win32

C:\Acrobat SDK\PluginSupport\Samples\Starter\win64

However, to improve your understanding of creating plugins, the remaining parts of this section discuss what tasks you must perform when creating a plugin from a blank project. When using the Starter sample plugin, it is not necessary to perform some of the tasks discussed in this section. For example, you do not need to start a new project, include header files, or add the PIMain source file. However, you still have to add application logic, compile, and build your project.

If you are developing on Windows using Visual Studio, you can use the Plugin Wizard tool to set up your plugin project. This tool includes the Acrobat SDK header files required for specific types of plugin solutions, and it adds the PIMain source file. The Wizard creates classes that uses `ToDo` markers to identify logic you must supply. You must still compile and build your plugin, as described in this section. For information on the Plugin Wizard, see Tools.

For information on developing an Adobe Reader plugin, see Creating an Adobe Reader plugin.

To create a plugin:

1. Start a new C project.
2. Include Acrobat SDK header files.
3. Add the PIMain source file to your project.
4. Add application logic to meet your business requirements.
5. Compile and build your plugin.

Including Acrobat SDK library files

To create a plugin, you must include Acrobat SDK library files, such as header files, into your project. You can link to these library files from within your development environment. For more information, see the documentation that accompanies your development environment.

The Acrobat SDK library files are separated into the following categories:

- Header files that are common to most plugins and generally referenced from PIMain.c.
- Header files specific to core and extended APIs.

You can find these header files in the following directory:

- Acrobat SDK\PluginSupport\Headers

Adding the PIMain source file

You must add the PIMain.c file to your project in order to create a plugin. This source file contains application logic such as handshaking methods, that are required by plugins. You can find this file in the following directory:

\Acrobat SDK\PluginSupport\Headers\API

After you add this file, you can add application logic to your project.

Note: As a plugin developer, you will never have to create the application logic that is located in the PIMain.c file or modify this file. However, you must include this file in your project.

Adding application logic

You must add a source file to your project that contains the following methods:

- `PluginExportHFTs`
- `PluginImportReplaceAndRegister`
- `PluginInit`
- `PluginUnload`
- `GetExtensionName`
- `PIHandshake`

You can copy the source code that is located in the StartInit.cpp file (located in the Starter plugin) and paste it. For information about these methods, see "About plugin initialization" on page 26.

The entry point to a plugin is the `PluginInit` method. For example, if you add the following line of code to this method, an alert box is displayed when Adobe Reader or Acrobat is started:

```
AVAlertNote("This is your first plugin");
```

You can add an application to the `PluginInit` method to meet your business requirements. You can invoke other user-defined functions that you create or you can add application logic to this method that performs a specific task. For example, you can add application logic to this method that adds a new menu item to Adobe Reader or Acrobat. (See "Creating Menus and Menu Commands" on page 82.)

Compiling and building your plugin

You must compile your plugin to build the API file. As stated earlier in this guide, plugins are equivalent to Windows DLLs; however, the file extension is .api, not .dll. Once you create an API file, you must add it to the following directory:

\Program Files\Adobe\Acrobat\plugins

After you add the plugin to this directory, you must restart Acrobat for the plugin to take effect.

Creating a sample PDF Library application

A PDF Library application does not have the same overhead as a plugin. That is, unlike a plugin, a PDF Library project does not require handshaking and initialization methods. A PDF Library application is a standard C/C++ project with PDF Library files included.

This section helps you get started with development using the Adobe PDF Library Software Developers Kit (SDK). It describes the contents of each directory in the PDF Library SDK installation, lists available code samples, and provides platform-specific information on how to set up the development environment.

Note: For a detailed discussion about using the PDF Library API, see "Inserting Text into PDF Documents" on page 54.

Contents of the PDF Library SDK

The Adobe PDF Library SDK consists of the following components:

- Core libraries that provide PDF Library functionality
- Header files that provide access to the libraries
- Fonts used in the library's basic operations
- Sample applications and code snippets showing how to use the library for a variety of purposes
- Documentation discussing development techniques and the PDF Library APIs.

Including library files

The following components are shipped with the PDF Library SDK:

- Acrobat PDF Library

  These are DLLs on the Windows platform and a shared object library Mac OS. In Windows, an interface library must be included in your Microsoft Visual Studio project. The following are the file names of these libraries:

  AdobePDFL.lib: The interface library for the Windows PDF Library DLL.

  AdobePDFL.dll: The Windows PDF Library DLL.

  libpdfl.so: The shared object library for supported UNIX platforms (deprecated).

  AdobePDFL framework: The framework for Mac OS.

- PDF Library SDK header files

  The PDF Library SDK include directory contains headers for accessing the API methods. You can link to these library files from within your development environment. Consult the documentation that accompanies your development environment for information about linking to library files.

  These files perform the same task in the PDF Library SDK as in the Acrobat SDK. For example, the PDCalls.h provides HFT functionality for PD layer functions. (See "Including Acrobat SDK library files" on page 46.)

Sample code

Samples are provided for the Windows and Mac OS in two forms:

- Stand-alone sample programs
- The SnippetRunner, an environment and infrastructure for code snippets that illustrate specific functions or techniques.

Sample code is intended to demonstrate the use of the PDF Library API and is not necessarily robust enough for a final implementation. The sample code itself is platform-independent, as is the majority of

the PDF Library API; the only difference between the sample source code for different platforms is the line-endings.

The Mac OS samples are provided as application packages. This format is normal for double-clickable applications, but they can also be run from the command line. To run them from the command line, you can either specify the command line arguments in the Xcode project file and execute within the IDE, or you can target the actual executable, which is in the Contents/MacOS folder inside the package. For example, from the Terminal window:

```
$ cd helowrld.app/Contents/MacOS/
$ helowrld
```

The MT (multi-threading) samples require command line arguments (a default set is added to the project files). Therefore, execution from within the IDE is preferred. Also, for those samples you must use absolute paths for the command line arguments.

Stand-alone samples

The following table lists the stand-alone sample applications that accompany the PDF Library SDK.

Sample application	Description
`addelem`	Shows how to modify existing pages in a PDF file. It adds a footer to each page and shifts the first line of each text run.
`all`	Used to compile all samples at the same time. Available for Windows and Mac OS only.
`CreatePattern`	Shows how to create tiling patterns in a PDF document.
`Decryption`	Shows how to programmatic-ally decrypt a PDF document encrypted with Acrobat standard security options.
`drawtomemory`	Shows how to render a page to memory using the `PDPageDrawContentsToMemory` PDF Library method, and creates a PDF file with a bitmap image rendered on the page.
`fontembd`	Shows font enumeration and font embedding.
`helowrld`	Shows the basics of creating a PDF document.
`JPXEncode`	Re-encodes PDF embedded images with the JPX filter and writes out a new PDF file with the re-encoded images embedded.
`mergepdf`	Shows how to merge two PDF files.
`MTInMemFS`	Demonstrates use of an in-memory file system for a simple workflow within a multi-threaded context.
`MTSerialNums`	Demonstrates creation of multiple threads to simultaneously generate multiple PDFs.
`MTTextExtract`	Demonstrates multiple threads concurrently processing multiple PDF documents.

Sample application	Description
`Peddler`	Shows how to add hyperlinking (specifically targeting URIs) capabilities to an existing PDF document.
`printpdf`	Shows how to print a PDF file to a printer or to a file using the PDF Library method PDFLPrintDoc.

SnippetRunner application

SnippetRunner allows you to quickly prototype code containing PDF Library API calls without the overhead of writing and verifying a complete application. It provides an infrastructure and utility functions to support execution and testing of code snippets, which are small but complete portions of PDF Library application code.

SnippetRunner consists of these major components:

- An application that acts as a back-end server and that provides the basic functionality, including a parameter input mechanism, debug support, and exception handling.
- A graphical user interface that acts as a client to the back-end server. (This user interface, called the Common User Interface, is also provided with the Acrobat SDK, which uses an Acrobat plugin for its back end.)

For more information about SnippetRunner, see the Snippet Runner Cookbook.

Developing applications with the Adobe PDF Library

This section details the compiler environment variables (macros) required to build applications against the Adobe PDF Library. On all platforms, you must define the PRODUCT macro for the preprocessor.

```
PRODUCT=\"HFTLibrary.h\"
```

This macro is used as a trigger for conditional compilation and allows the same headers to be used for both the Acrobat core API and the Adobe PDF Library.

Windows

The following macros must also be defined in the IDE project settings for applications to compile correctly on the Windows platform:

- `WIN_ENV`
- `WIN32`
- `WIN_PLATFORM`

The Adobe PDF Library is compiled with code generation set to Multithreaded. Applications linking with the Adobe PDF Library must have code generation settings that match or there will be conflicts between the Microsoft libraries `MSVCRT` and `LIBCMT`.

In Visual Studio, the Ignore Libraries settings (click Project Settings > Link > Input > Ignore libraries) should not ignore LIBCMT (other versions of PDF Library do ignore it).

The Adobe PDF Library is distributed as an interface library (AdobePDFL.lib) and matching DLL (AdobePDFL.dll). You should link the interface library into your application.

The operating system must be able to access the Adobe PDF Library at runtime. It does so by searching the paths specified by the PATH environment variable, as well as the folder in which the application was launched.

Mac OS

The Mac OS libraries use a precompiled header and prefix file to define the appropriate macros. See Precompile.pch in the Samples:utils directory of the Adobe PDF Library SDK for the macros required to successfully compile the samples.

Initialization and termination

Applications must initialize and terminate the Adobe PDF Library appropriately:

- Call `PDFLInit` to set up internal data structures, locate required resources such as fonts, and perform initialization (such as setting client-provided memory allocation routines). Calling most library functions without successfully initializing the library results in error conditions. The rest of this section provides details on using PDFLInit.
- Call `PDFLTerm` to clean up before an application terminates or when access to PDF Library functionality is no longer needed.

Since the PDF Library supports thread-safety (since version 6.1.2), initialization and termination are handled on a per-thread basis.

The `PDFLInit` function takes as a parameter a `PDFLData` structure, defined in the API header file PDFInit.h. You must provide valid values for the following members of the structure before passing it to `PDFLInit`:

- size denotes the size of the structure and can be obtained with `sizeof(PDFLDataRec)`.
- listLen is the number of directories listed in dirList.
- dirList is an array of directories that contain font resources. The following discussion explains how to use this member on each of the supported platforms.

In Windows and Mac OS, the PDF Library searches for fonts in the default system and in their subdirectories (to 99 levels). You can specify additional font directories to search (also to 99 levels) in the dirList array. (Note that this can affect performance.)

Here is an example showing how to pass the font paths to dirList for Windows:

```
pdflLibData.dirList[0]= strdup("C:\\Myfontfolder\\CMap");
pdflLibData.dirList[1]= strdup("C:\\Myfontfolder\\CIDFont");
pdflLibData.dirList[2]= strdup("C:\\Myfontfolder\\Font");
```

The paths can be either full paths or paths relative to the directory from which the executable linking in the Adobe PDF Library was started. You can set the value `kPDFLInitIgnoreDefaultDirectories` in the flags field of the `PDFLData` structure to indicate that the default font directories should not be searched but only the directories provided in dirList.

For more details, see the functions `PDFLGetDirList_Win` and `PDFLGetDirList_Mac` in the MyPDFLibUtils.cpp file in the `Samples/utils` directory.

Multithreading

When using the thread-safe PDF Library, initialization and termination now additionally need to be performed for each thread that calls into the library, as well as at the process level. The interfaces for per-thread initialization/termination are the same as before.

Since each thread acquires an independent PDF Library memory context, you should not share PDF Library data and resources among threads. This includes sharing the same PDF file.

The Adobe PDF libraries are thread-safe. To use threads, simply make the appropriate system call (`beginthreadex` on Windows). Multiple threads cannot share PDF Library data types. However, they share the same process heap; therefore, an application can share generic data types between threads. Multiple threads can open the same file read-only; however, multiple threads should not attempt to write to the same PDF document.

In Windows, `CreateThread` is not recommended if the application is using most stdio.h-defined functions, including file I/O and string manipulation. It is best to `use _beginthreadex` on Windows, which performs extra bookkeeping to ensure thread safety.

Upgrading existing plugins

This section discusses how to upgrade an existing Acrobat plugin to work with a newer version of Acrobat.

Refer to the Release Notes.

Detecting supported APIs

Acrobat Pro and Acrobat Pro Extended support the full set of APIs. For Acrobat Standard and Adobe Reader, if you try to use an API that is not supported, nothing will happen. The same HFT version numbers are used across products, so all APIs are callable on all products, but some APIs simply do not work on certain products.

Additionally, the Extended APIs provided by plugins do not work if an Acrobat product does not support the use of those APIs. The HFTs do not load, so you must check whether the HFT was successfully imported.

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:

```
gAcroColorHFT = ASExtensionMgrGetHFT(ASAtomFromString(AcroColorHFTNAME),
PI_ACROCOLOR_VERSION);
```

Plugins that use new HFTs introduced with the current Acrobat version do not run on earlier Acrobat versions. Whether or not an attempt to load these HFTs forces the plugin to fail is controlled by flags in PIMain.c of the form `PI_HFT_OPTIONAL`. By default, these flags are undefined, so if your plugin attempts to load HFT and cannot, initialization fails. If you define `PI_HFT_OPTIONAL` with some value (preferably 1) and the load is not successful, initialization continues.

Use the `ASGetConfiguration` method to determine the configuration on which the plugin 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.

Under Adobe Reader, when a rights-enabled PDF file is opened, a flag is set that allows a plugin 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 plugin menus and toolbars appropriately at initialization. Ensure that you make calls only to APIs supported on the platform detected.

Migrating a PDF Library application from CodeWarrior to Xcode

For the PDF Library, the supported Mac OS X development environment is Xcode (formerly the supported environment was Metrowerks CodeWarrior). With this change comes a new set of headers, frameworks and libraries that may or may not be compatible with existing plugin code and with existing CodeWarrior projects.

You can migrate a PDF Library application that was created using CodeWarrior to Xcode. As a starting point, it is recommended that you read the information that is located at the following URL: http://developer.apple.com/documentation/DeveloperTools/Conceptual/MovingProjectsToXcode/index.html.

The PDF Library SDK samples have debug and release targets that are built against the MacOSX10.4u.sdk SDK. These are carbon applications that create universal binaries linked to universal Adobe libraries. For a complete list of compatible application build settings, see the MacSDKConfiguration Xcode configuration files and ProjectConfigurations files included in the PDF Library SDK.

4 Inserting Text into PDF Documents

Use the PDF Library API to insert text into a PDF document. As stated earlier in this guide, the PDF Library API is a subset of the Acrobat core API that enables your application to interact and manipulate PDF documents. The PDF Library API is not used to create plugins for Adobe Reader or Acrobat. It is also not used to modify the user interface of Adobe Reader or Acrobat, such as by adding a toolbar or menu item.

About inserting text into a PDF document

You can use the PDF Library API to programmatically insert text into a PDF document. This functionality is useful for dynamically updating a PDF document with information obtained during run-time. You can insert text into a PDF document by performing the following tasks:

1. Create a new PDF document.
2. Create a new page and insert it into the PDF document.
3. Create a container for the page content.
4. Acquire a font and set its attributes.
5. Insert the text element into page content.
6. Convert content into resource and content objects.
7. Place resource and content objects onto the page.
8. Save the PDF document.
9. Release all objects.

Note: Assume that all code examples specified in this chapter are located in an entry function named `MainProc`.

Creating a new PDF document

You can programmatic-ally create a new PDF document by invoking the `PDDocCreate` method. This method returns a `PDDoc` object that only contains one COS object, which is a Catalog. For information about COS objects, see "Working with Cos Objects" on page 169.

The following code example creates a `PDDoc` object by invoking `PDDocCreate` method.

Example: *Creating a new PDF document*

```
PDDoc pdDoc ;
pdDoc = DDocCreate();
```

Creating a new page

You can create a new page by invoking the `PDDocCreatePage` method. However, before invoking this method, set the page size by creating an `ASFixedRect` object, which represents a rectangle region that specifies the page size. After you declare an `ASFixedRect` object, you can specify the page size by setting the `ASFixedRect` object's `left`, `top`, `right`, and `bottom` attributes.

Next, invoke the `PDDocCreatePage` method and pass the following arguments:

- A `PDDoc` object that was created.
- An `ASInt32` value that specifies the page number after which the new page is inserted. Specify `PDBeforeFirstPage` to insert the new page at the beginning of the PDF document.
- An `ASFixedRect` object that specifies the page size.

This method returns a `PDPage` object. The following code example creates a new page, sets it size, and inserts it into the PDF document represented by the `PDDoc` object.

Example: *Creating a new page*

```
PDPage pdPage;
ASFixedRect mediaBox;

//Set the page size
mediaBox.left = fixedZero;
mediaBox.top = Int16ToFixed(4*72);
mediaBox.right = Int16ToFixed(5*72);
mediaBox.bottom = fixedZero;

//Create a page and insert it as the first page
PDPagepdPage = PDDocCreatePage(pdDoc, PDBeforeFirstPage, mediaBox);
```

Creating a container

To insert text into a PDF document, you must create a PDF container. A container contains the modifiable contents of a PDF page. To create a PDF container, invoke the `PDPageAcquirePDEContent` method and pass the following arguments:

- A `PDPage` object.
- An `ASExtension` object that represents the caller. If you invoke this method from a PDF Library project, you can pass `NULL`. In contrast, if you invoke this method from a plugin, you would pass the `gExtensionID` extension.

The `PDPageAcquirePDEContent` method returns a `PDEContent` object, as shown in the following code example.

Example: *Creating a PDEContent object*

```
PDEContent pdeContent;
pdeContent = PDPageAcquirePDEContent(pdPage, NULL);
```

Acquiring fonts

You must acquire the font that you will use to insert text into a PDF document. You can reference a font that is installed on the host computer by creating a `PDSysFont` object and use this object to create a

`PDEFont` object, which represents the font that is used to draw text on a page. To acquire a font, perform the following tasks:

1. Create a `PDEFontAttrs` object.
2. Allocate the size of the `PDEFontAttrs` object's buffer by using the `memset` method.
3. Set the `PDEFontAttrs` object's `name` and `type` attributes. The `name` attribute defines the font name. For example, you can specify `CourierStd`. The `type` attribute defines the font subtype. For example, you can specify `Type1`, which is a Type 1 PostScript font. For information about valid font subtype values, see the Acrobat and PDF Library API Reference.
4. Create a `PDSysFont` object by invoking the `PDFindSysFont` method and passing the following arguments:
   - The address of the `PDEFontAttrs` object.
   - Size of the `PDEFontAttrs` object in bytes.
   - A value that specifies a `PDSysFontMatchFlags` value. (See the Acrobat and PDF Library API Reference.)
5. Create a `PDEFont` object that represents the font to use within a PDF document by invoking the `PDEFontCreateFromSysFont` method and passing the following arguments:
   - A `PDSysFont` object that references a system font.
   - An `ASUnit32` value that indicates whether to embed the font or whether to subset the font. You can pass `kPDEFontDoNotEmbed` to this argument. For information about other values, see the Acrobat and PDF Library API Reference.

The following code example acquires a font that is used to draw text on a page.

Example: ***Acquiring a font that is used to draw text on a page***

```
PDSysFont sysFont;
PDEFont pdeFont;

//Set the size of the PDSysFont and set its attributes
memset(&pdeFontAttrs, 0, sizeof(pdeFontAttrs));
pdeFontAttrs.name = ASAtomFromString("CourierStd");
pdeFontAttrs.type = ASAtomFromString("Type1");

//Get system font
sysFont = PDFindSysFont(&pdeFontAttrs, sizeof(PDEFontAttrs), 0);

//Create a font that is used to draw text on a page
pdeFont = PDEFontCreateFromSysFont(sysFont, kPDEFontDoNotEmbed);
```

Creating a PDEGraphicState object

You must create a `PDEGraphicState` object that is used to define attributes of the `PDEText` object. After you create a `PDEGraphicState` object, set the following attributes:

strokeColorSpec: The stroke color specification. The default value is `DeviceGray`.

miterLimit: The miter limit, corresponding to the M (setmiterlimit) operator. The default value is `fixedTen`.

flatness: The line flatness corresponding to the i (setflat) operator. The default value is `fixedZero`.

lineWidth: The line width corresponding to the w (setlinewidth) operator. The default value is `fixedOne`.

For more information about attributes that belong to this object, see the Acrobat and PDF Library API Reference.

The following code example creates a `PDEGraphicState` object and sets it attributes.

Example: *Creating a PDEGraphicState object*

```
//Create a DEGraphicState object and set its attributes
PDEGraphicState gState;
PDEColorSpace pdeColorSpace =
PDEColorSpaceCreateFromName(ASAtomFromString("DeviceGray"));
memset(&gState, 0, sizeof(PDEGraphicState));
gState.strokeColorSpec.space = gState.fillColorSpec.space = pdeColorSpace;
gState.miterLimit = fixedTen;
gState.flatness = fixedOne;
gState.lineWidth = fixedOne;
```

Creating an ASFixedMatrix object

Create an `ASFixedMatrix` object that represents the transformation matrix for the text. By creating this object, you can set text properties such as the font size and the width and height of the text. The following code example creates an `ASFixedMatrix` object and sets its attributes.

Example: *Creating an ASFixedMatrix object*

```
//Create an ASFixedMatrix object
memset(&textMatrix, 0, sizeof(textMatrix));    /* Set the buffer size */
textMatrix.a = Int16ToFixed(24);     /* Set font width and height */
textMatrix.d = Int16ToFixed(24);     /* to 24 point size      */
textMatrix.h = Int16ToFixed(1*72);   /* x,y coordinate on page */
textMatrix.v = Int16ToFixed(2*72);   /* in this case, 1" x 2"  */
```

Inserting text

You can insert text into the PDF document by performing the following tasks:

1. Create a `PDEText` object by invoking the `PDETextCreate` method. This method does not require arguments.
2. Invoke the `PDETextAdd` method and pass the following arguments:
   - A `PDEText` object that you created.

- A `PDETextFlags` value that specifies the type of text to add. Specify `kPDETextRun` for a text run.
- An `ASInt32` value that specifies the index after which to add the text run. You can specify `0` to add the text run to the beginning.
- A character pointer that specifies the text to insert into the PDF document.
- An `ASInt32` value that specifies the length of the text run.
- The `PDEFont` object that you created. (See "Acquiring fonts" on page 55.)
- The address of the `PDEGraphicState` object that you created. (See "Creating a PDEGraphicState object" on page 57.)
- The size of the `PDEGraphicState` object. You can use the `sizeof` method to get this information.
- The address of the `PDETextState` object. You can pass `NULL`.
- The size of the `PDETextState` object.
- The address of the `ASFixedMatrix` object that you created. (See "Creating an ASFixedMatrix object" on page 57.)
- The address of the `ASFixedMatrix` object that holds the matrix for the line width when stroking text. It is recommended that you pass `NULL`.

3. Insert text into the page content by invoking the `PDEContentAddElem` method and passing the following arguments:
   - The `PDEContent` object that you created. (See "Creating a container" on page 55.)
   - The location to where the text is added. Pass `kPDEAfterLast` to add the text to the end of the `PDEContent` object.
   - The `PDEText` object and cast this object to `PDEElement`.
4. Set the page's `PDEContent` back into the page by invoking the `PDPageSetPDEContent` method and passing the following arguments:
   - A `PDPage` object that you created. (See "Creating a new page" on page 55.)
   - An `ASExtension` object that represents the caller. If you invoke this method from a PDF Library project, you can pass `NULL`. In contrast, if you invoke this method from a plugin, you would pass the `gExtensionID` extension.

The following code example inserts text into a PDF document.

Example: *Inserting text into a PDF document*

```
//Create a PDEText object
pdeText = PDETextCreate();

//Create a character pointer
char *HelloWorldStr = "Hello There";

//Create new text run
PDETextAdd(pdeText,          //Text container to add to
     kPDETextRun,           // kPDETextRun
     0,                    // index
     HelloWorldStr,        // Text to add
     strlen(HelloWorldStr),// Length of text
     pdeFont,               // Font to apply to text
```

```
    &gState,                //Address of PDEGraphicState object
    sizeof(gState),        //Size of graphic state to apply to text
    NULL,
    0,
    &textMatrix,            //Transformation matrix for text
    NULL);                  //Stroke matrix

//Insert text into page content
PDEContentAddElem(pdeContent, kPDEAfterLast, (PDEElement) pdeText);
```

Saving the PDF document

You can save the PDF document by invoking the `PDDocSave` method and passing the following arguments:

- A `PDDoc` object that represents the PDF document.
- A `PDSaveFlags` object that specifies save options. For example, you can pass `PDSaveFull` to this argument, which results in the entire document being saved. For more information about this parameter, see the Acrobat and PDF Library API Reference.
- An `ASPathName` object that represents the path to which the file is saved.
- An `ASFileSys` object that represents the file system. (See "Creating an ASFileSys object" on page 153.)
- A progress monitor value. Invoke the `AVAppGetDocProgressMonitor` method to obtain the default. You can pass `NULL`, in which case no progress monitor is used.
- The address of a `progMonClientData` object that contains user-supplied data to pass to `progMon` each time it is called. Pass `NULL` if the previous argument (progMon) is `NULL`.

The following code example saves the PDF document to the local directory as out.pdf.

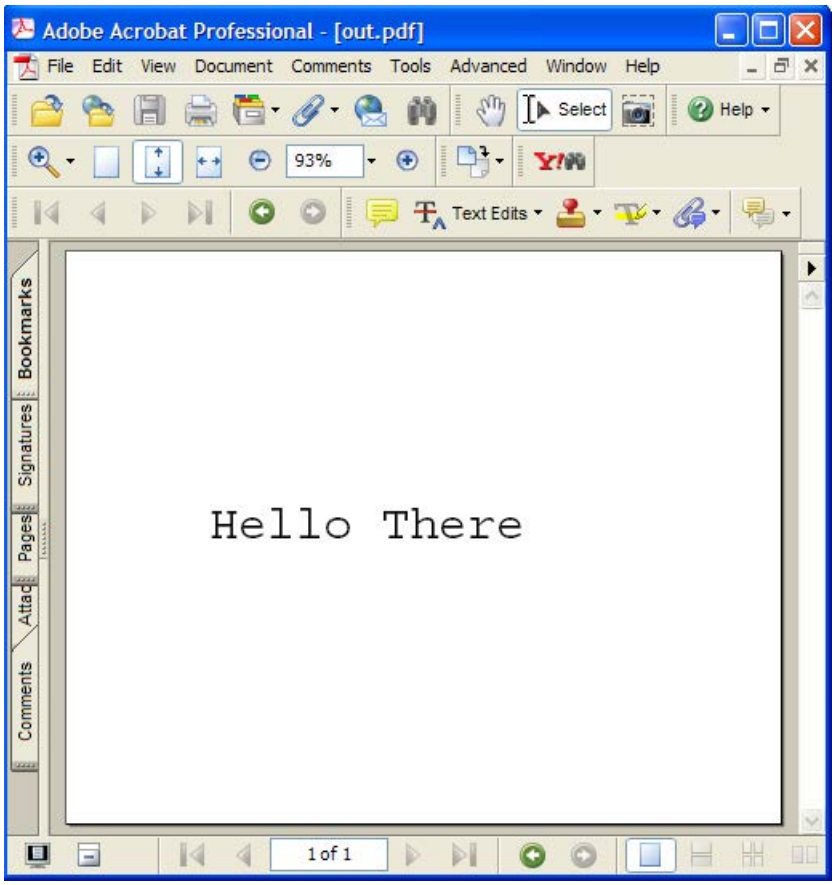
Example: *Saving a PDF document*

```
//Save the PDF document
#if !MAC_ENV
   PDDocSave(pdDoc, PDSaveFull | PDSaveLinearized ,ASPathFromPlatformPath
("out.pdf"), NULL, NULL, NULL);
#else
   ASPathName macPath = GetMacPath("out.pdf");
   PDDocSave(pdDoc, PDSaveFull | PDSaveLinearized, macPath , NULL, NULL, NULL);
   ASFileSysReleasePath(NULL,macPath);
```

Examining a PDF Library application source file

This section shows the entire source file that is responsible for importing text into a PDF document.

The following image shows the PDF that is created when this source file is executed.



The following code example is a source file that inserts text into a PDF document.

Example: *Examining a PDF Library application source file*

```
#ifdef MAC_ENV
#include <Carbon/Carbon.h>
#else
#include <sys/types.h>
#include <sys/stat.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

#include "PDFInit.h"
#include "CosCalls.h"
#include "CorCalls.h"
#include "ASCalls.h"
#include "PDCalls.h"
#include "PSFCalls.h"
#include "PERCalls.h"
#include "PEWCalls.h"
#include "PIExcept.h"
```

```
#include "PagePDECntCalls.h"

#ifdef MAC_ENV
#include "macUtils.h"
#endif

void MainProc(void)
{
   char buf[256];
   ASInt32 i;
   ASBool b;
   PDDoc pdDoc;              /* Reference to a PDF document  */
   PDPage pdPage;            /* Reference to a page in doc */
   PDEContent pdeContent;    /* Container for page content */
   ASFixedRect mediaBox;     /* Dimensions of page */
   PDSysFont sysFont;        /* Used by PDEFont creation */
   PDEFont pdeFont;          /* Reference to a font used on a page */
   PDEFontAttrs pdeFontAttrs; /* Font attributes    */
   PDEText pdeText;           /* Container for text */
   ASFixedMatrix textMatrix;  /* Transformation matrix for text */
   PDEColorSpace pdeColorSpace; /* ColorSpace */
   PDEGraphicState gState;    /* Graphic state to apply to operation */
   char *HelloWorldStr = "Hello There"; // Text to write to the PDF
#ifdef MAC_ENV
   ASPathName macPath;
#endif


/*====================================================================*\
                Create the doc, page, and content container
\*====================================================================*/

DURING

   pdDoc = PDDocCreate();                 /* Create new document  */

   mediaBox.left   = fixedZero;            /* Dimensions of page   */
   mediaBox.top    = Int16ToFixed(4*72);    /* In this case 5" x 4" */
   mediaBox.right  = Int16ToFixed(5*72);
   mediaBox.bottom = fixedZero;

   //Create a page with those dimensions and insert it as the first page
   pdPage = PDDocCreatePage(pdDoc, PDBeforeFirstPage, mediaBox);

   //Acquire a PDEContent container for the page
   pdeContent = PDPageAcquirePDEContent(pdPage, NULL);

/*====================================================================*\
     Acquire a font, add text, and insert into page content container
\*====================================================================*/

   //Set the size of the PDSysFont and set its attributes
   memset(&pdeFontAttrs, 0, sizeof(pdeFontAttrs));
   pdeFontAttrs.name = ASAtomFromString("CourierStd");
   pdeFontAttrs.type = ASAtomFromString("Type1");
```

```
    sysFont = PDFindSysFont(&pdeFontAttrs, sizeof(PDEFontAttrs), 0);
    pdeFont = PDEFontCreateFromSysFont(sysFont, kPDEFontDoNotEmbed);

    /* The following code sets up the default graphics state.  We do this so that
     * we can free the PDEColorSpace objects
     */

    //PDEColorSpace pdeColorSpace; /* ColorSpace */
    //PDEGraphicState gState;    /* Graphic state to apply to operation */

    //Create a PDEGraphicState object and set its attributes
    pdeColorSpace =
  PDEColorSpaceCreateFromName(ASAtomFromString("DeviceGray"));
    memset(&gState, 0, sizeof(PDEGraphicState));
    gState.strokeColorSpec.space = gState.fillColorSpec.space = pdeColorSpace;
    gState.miterLimit = fixedTen;
    gState.flatness = fixedOne;
    gState.lineWidth = fixedOne;

    //Create an ASFixedMatrix object
    memset(&textMatrix, 0, sizeof(textMatrix));    /* Set the buffer size */
    textMatrix.a = Int16ToFixed(24);     /* Set font width and height */
    textMatrix.d = Int16ToFixed(24);     /* to 24 point size      */
    textMatrix.h = Int16ToFixed(1*72);   /* x,y coordinate on page */
    textMatrix.v = Int16ToFixed(2*72);   /* In this case, 1" x 2"  */

    //Create a PDEText object
    pdeText = PDETextCreate();          /* Create new text run    */

    PDETextAdd(pdeText,       /* Text container to add to  */
      kPDETextRun,           /* kPDETextRun, kPDETextChar */
      0,                     /* Index */
      HelloWorldStr,         /* Text to add    */
      strlen(HelloWorldStr), /* Length of text */
      pdeFont,               /* Font to apply to text */
      &gState,
      sizeof(gState),       /* Graphic state to apply to text  */
      NULL,
      0,                     /* Text state and size of structure*/
      &textMatrix,           /* Transformation matrix for text  */
      NULL);                 /* Stroke matrix  */

   /* Insert text into the page content */
    PDEContentAddElem(pdeContent, kPDEAfterLast, (PDEElement) pdeText);


  /*====================================================================*\
          Convert to objects, add to page, and release resources
  \*====================================================================*/

    /* Set the PDEContent for the page */
    b = PDPageSetPDEContent(pdPage, NULL);

    /* Save document to a file */
```

```
#if !MAC_ENV
   PDDocSave(pdDoc, PDSaveFull | PDSaveLinearized ,ASPathFromPlatformPath
("out.pdf"), NULL, NULL, NULL);
#else
   macPath = GetMacPath("out.pdf");
   PDDocSave(pdDoc, PDSaveFull | PDSaveLinearized, macPath , NULL, NULL, NULL);
   ASFileSysReleasePath(NULL,macPath);
#endif


  //Release all objects
   PDERelease((PDEObject) pdeFont);
   PDERelease((PDEObject) pdeText);
   i = PDPageReleasePDEContent(pdPage, NULL);
   PDERelease((PDEObject)pdeColorSpace);
   PDPageRelease(pdPage);
   PDDocRelease(pdDoc);

HANDLER
   ASGetErrorString(ERRORCODE, buf, sizeof(buf));
   fprintf(stderr, "Error code: %ld, Error Message: %s\n", ERRORCODE, buf);
   exit (-1);
END_HANDLER
}

#define INCLUDE_MYPDFLIBAPP_CPP1
#include "MyPDFLibApp.cpp"
#undef INCLUDE_MYPDFLIBAPP_CPP
```

5 Working with Documents and Files

This chapter explains how to use the Acrobat core API to perform operations on PDF documents and files, such as opening a PDF document in an external window. When working with documents and files, you use the following typedefs:

- An `AVDoc` typedef (from the 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` typedef (from the PD layer) represents a document as a PDF file. You work with `PDDoc` objects to make changes to a document's contents. Using a `PDDoc` object, you can access components, such as a `PDPage` object.
- An `ASFile` typedef (from the AS layer) represents a document as an open file.

Opening PDF documents

You can use the Acrobat core API to create application logic that opens a PDF document in Acrobat or Adobe Reader. The method that opens a PDF file is `AVDocOpenFromFile`. Before you invoke this method, you must create an `ASPathName` object, which is a platform-independent path value that specifies the PDF file to open.

With Acrobat or Adobe Reader 8.0 and later, you can open PDF documents that are based on Unicode-named files. (See "Working with Unicode Paths" on page 152.)

You can create an `ASPathName` object by using different techniques. This section describes how to create an `ASPathName` object by invoking the `ASFileSysCreatePathName` method to convert a platform specific path name to an `ASPathName` object. This method requires the following arguments:

- An `ASFileSys` object that represents the file system in which you are creating an `ASPathName` object. (See "Creating an ASFileSys object" on page 153.)
- An `ASAtom` object that specifies the data type of the third argument. You can specify `Cstring`, which is accepted by the default file system on all platforms. In Windows, the path can be absolute (`C:\\folder\\file.pdf`) or relative (`...\folder\file.pdf`). In Mac OS, it must be an absolute path separated by colons (`VolumeName:Folder:file.pdf`). For information about additional supported values, see the `ASFileSysCreatePathName` method in the Acrobat and PDF Library API Reference.
- A null- terminated character that specifies the PDF file on which the `ASPathName` object is based.
- A null- terminated character that specifies additional data that you can use. You can pass `NULL`. For more information, see the `ASFileSysCreatePathName` method in the Acrobat and PDF Library API Reference.

The `ASFileSysCreatePathName` method returns an `ASPathName` object. You can also create an `ASPathName` object by displaying an open dialog box. (See "Displaying an open dialog box" on page 68.)

To programmatic-ally open a PDF file in Acrobat or Adobe Reader, invoke the `AVDocOpenFromFile` method and pass the following arguments:

- An `ASPathName` object that specifies the PDF file to open.

- An `ASFileSys` object that represents the file system in which the PDF file is located. (See "Creating an ASFileSys object" on page 153.)
- An `ASText` object that specifies a string value to display in the Adobe Reader or Acrobat title bar.

The following code example opens a PDF document that is based on a file named PurchaseOrder.pdf.

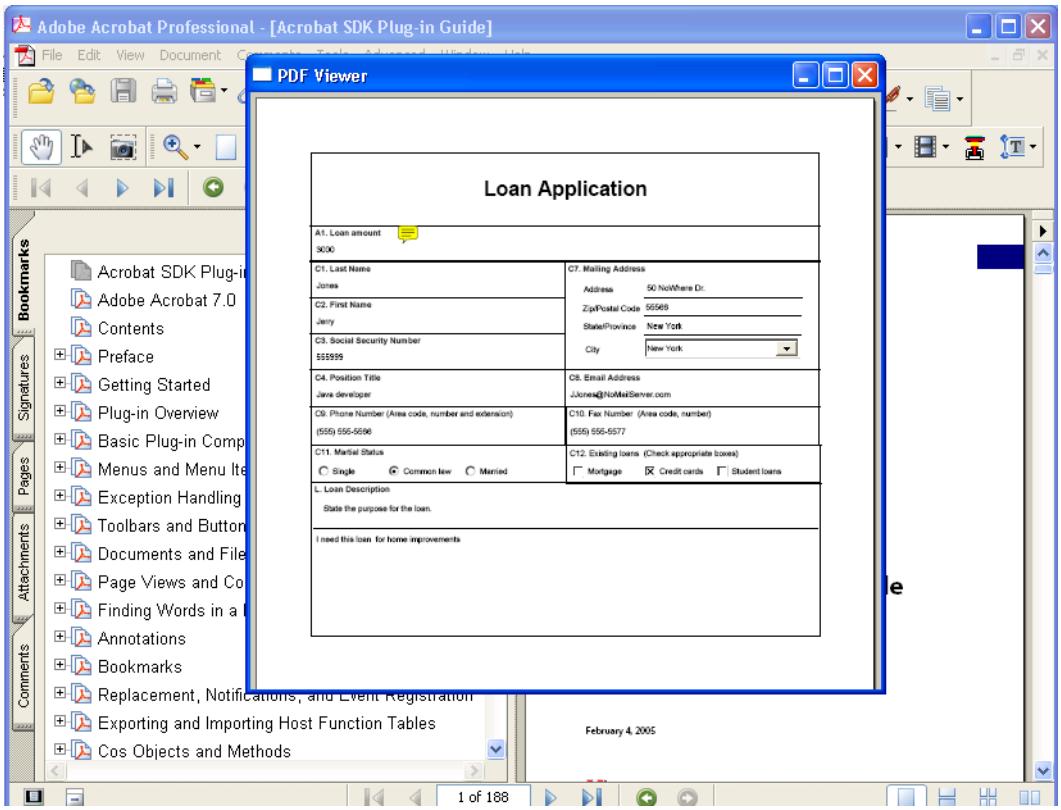
Example: *Opening a PDF file*

```
//Specify the PDF file to open (host encoded names only)
const char* myPath = "C:\\PurchaseOrder.pdf";
ASAtom pathType = ASAtomFromString("Cstring");

//Create an ASText object
ASText titleText = ASTextNew();
ASTextSetPDText(titleText, "This PDF was opened by using the Acrobat SDK");

//Create an ASPathName object
ASFileSys fileSys = ASGetDefaultFileSysForPath(pathType, myPath);
ASPathName pathName = ASFileSysCreatePathName(fileSys, pathType, myPath,
NULL);

//Open the PDF file
AVDoc myDoc = AVDocOpenFromFile(pathName, fileSys, titleText);

//Do some clean up
ASFileSysReleasePath(fileSys, pathName);
ASTextDestroy(titleText);
```

Opening a PDF document in an external window

You can use the Acrobat core API to open a PDF document in an external window. For example, from within Adobe Reader or Acrobat, you can let a user select a PDF file and then display the PDF document in an external window. This lets a user view two separate PDF documents at the same time.

The following image shows a PDF document displayed in an external window.



In addition to using the typedefs introduced in this chapter, you must also use the following typedefs to open a PDF document in an external window:

- `AVDocOpenParamsRec` - defines required parameters for opening a PDF document in a window. This typedef lets you control the external window's size, location, and visibility.
- `ExternalDocServerCreationDataRec` - defines data that is used in conjunction with an `AVDoc` object that is displayed in an external window.

To open a PDF document in an external window, perform the following tasks:

1. Create a new window.
2. Create an `AVDocOpenParamsRec` object.
3. Create a `ExternalDocServerCreationDataRec` object.
4. Display an open dialog box to enable a user to retrieve a PDF file to open.
5. Display the PDF document that the user selected within a window.

Note: The remaining parts of this section describes these steps in detail.

Creating a Window

You must programmatic-ally create the external window in which to display a PDF document. To create a window, you invoke standard platform C functions and not the Acrobat core API. For example, if you are creating your plugin in Windows, you can use the Win32 API to create a window. In this situation, part of

the process of creating a window is to register the window class and define the window procedure. After you perform these tasks, you can obtain a handle to the window by invoking the `CreateWindow` function. For information about this function, see the MSDN online help at http://msdn.microsoft.com/library/default.asp.

Note: Application logic that creates a window is shown in "Opening a PDF document in an external window" on page 71.

Defining the parameters for an external window

You must create an `AVDocOpenParamsRec` object in order to open and display a PDF document in an external window. This object defines required parameters for opening a PDF document within an external window. After you create an `AVDocOpenParamsRec` object, allocate its buffer size and set the following attributes:

size: The size of the data structure. This attribute must be set to `sizeof(AVDocOpenParamsRec)`.

useFrame: If `true`, the `frame` attribute specifies the size and location of the window in which the PDF document is displayed. If `false`, the `frame` attribute is ignored and a default frame is used.

frame: An `AVRect` object specifying the size and location of the window in which the PDF document is displayed. In the Windows operating system, the coordinates are MDI client coordinates. In Mac OS, the coordinates are global screen coordinates. This attribute is ignored if the `useFrame` attribute is `false`.

useVisible: If `true`, the `visible` attribute determines whether the window is visible after the PDF document is opened. If this attribute is `false`, then the `visible` attribute is ignored.

visible: Specifies the window's visibility. If this attribute is `false` and the `useVisible` attribute is `true`, then the `frame` attribute is ignored regardless of the value of the `useFrame` attribute.

If you are using the Windows operating system and this attribute is `true`, the PDF document is opened in a visible window. If this attribute is false, the PDF document is opened in a minimized window. This attribute is ignored if the `useVisible` attribute is `false`.

useServerType: An `ASBool` value that specifies whether the `serverType` and `serverCreationData` attributes are used.

serverType: The name of the `AVDoc` server for this `AVDoc` object. Specify `EXTERNAL` for an external window.

serverCreationData: Platform-dependent server data to associate with the `AVDoc` server. If the `serverType` attribute is `EXTERNAL`, this attribute must be assigned the address of the `ExternalDocServerCreationData` object.

useViewType: An `ASBool` value that specifies whether the `viewType` attribute is used.

viewType: Specifies the appearance of the window that contains the PDF document. The following values are valid:

- **AVPageView**: Displays only the page view of the document. User interface components such as display scrollbars, toolbars, bookmarks, and thumbnails panes are not displayed. Annotations, such as links, are active.
- **AVDocView**: Displays the page view of the document as well as scroll bars, bookmarks, and the thumbnails pane. Annotations, such as links, are active.
- **AVExternalView**: Displays the page view of the document as well as scroll bars, toolbars, bookmarks, and the thumbnails pane. Annotations, such as links, are active.
- **AVEmbeddedView**: Embeds the PDF file in an external document such as an HTML file. The first page of the PDF document is displayed; however, user-interface component such as scroll bars,

toolbars, bookmarks, and the thumbnails pane are not displayed. Annotations, such as links, are not displayed or active.

useReadOnly: An `ASBool` value that specifies whether the `ReadOnly` attribute is used.

readOnly: An `ASBool` value that opens the document in read-only mode if set to `true`.

useSourceDoc: An `ASBool` value that specifies whether the `SourceDoc` attribute is used.

sourceDoc: An `AVDoc` whose window is taken over by the new PDF document.

Note: Application logic that creates an `AVDocOpenParamsRec` object is shown in "Opening a PDF document in an external window" on page 71.

Creating a handler for an external window

You must create an `ExternalDocServerCreationDataRec` object in order to open a PDF document in an external window. This object contains callback functions that implement a window handler. After you create an `ExternalDocServerCreationDataRec` object, allocate its buffer size and set the following attributes:

size: The size of the data structure. This attribute must be set to `sizeof(ExternalDocServerCreationDataRec)`.

platformWindow: A platform-dependent structure of type `ExternalDocWindowData` representing a window. This is a platform specific value:

- **Windows**: use a `HWND` value cast as `ExternalDocWindowData`
- **Mac OS**: use an `ExternalDocWindowData` object

acrobatProc: An `AVExecuteProc` callback that is invoked when the Acrobat button (if present) is clicked in an external application. This attribute is optional.

acrobatProcData: Client specific data that is used in the `AVExecuteProc` callback. This attribute is optional.

For a complete list of attributes that belong to an `ExternalDocServerCreationDataRec` object, see the Acrobat and PDF Library API Reference.

Note: Application logic that creates an `ExternalDocServerCreationDataRec` object is shown in "Opening a PDF document in an external window" on page 71.

Displaying an open dialog box

You can display an open dialog box that enables a user to select a PDF document to open. In addition to describing how to create an open dialog box, this section also describes how to use a dialog box to create an `ASPathName` object and an `ASFileSys` object, which are both used to open a PDF document.

The Acrobat core API has a typedef named `AVOpenSaveDialogParamsRec` that you use to display an open dialog box. The `AVOpenSaveDialogParamsRec` typedef contains the following attributes that you programmatic-ally set:

size: The size of the `AVOpenSaveDialogParamsRec` object's buffer.

flags: An `AVOpenSaveDialogFlags` value that specifies the dialog box's appearance. The following values are valid:

- **kAVOpenSaveAllowAllFlag**: Specifies `All Files (*.*)` for the dialog box. This value is only applicable for an open dialog box.

- **kAVOpenSaveAllowMultiple**: Allows multiple files to be opened through the dialog box. This value is only applicable for an open dialog box.
- **kAVOpenSaveAllowForeignFileSystems**: Allows file systems other than the default to be used to open the files. You need to select this value to open Unicode-named files. This value is only applicable for an open dialog box.
- **kAVOpenSaveAllowSettingsButton**: Adds a settings button to the dialog box. This value is applicable to both open and save dialog boxes.
- **kAVOpenSaveMergeTogetherPassedFilters**: Meaningful only for open dialog boxes with more than one filter.

parentWindow: An `AVWindow` object that specifies the parent window for the dialog box. This attribute is ignored on Mac OS.

windowTitle: An `ASText` object that specifies the title for the dialog box. This attribute can be `NULL`, in which case the default title is used.

actionButtonTitle: An `ASText` object that specifies the title of the action button. This attribute can be `NULL`, in which case the default title is used.

cancelButtonTitle: An `ASText` object that specifies the title of the cancel button. This attribute can be `NULL`, in which case the default title is used.

initialFileSys: An `ASFileSys` object that specifies the default file system. This value can be `NULL` if the `flags` attribute does not contain `kAVOpenSaveAllowForeignFileSystems`.

initialPathName: An `ASPathName` object that specifies an initial path location. This value can be `NULL`.

initialFileName: Specifies an initial file to save. This value is ignored for an open dialog box.

fileFilters: An array of pointers to addresses of `AVFileFilterRec` objects that act as a file filter (this attribute is set in the example that is described in this section).

numFileFilters: Specifies the number of `AVFileFilterRec` pointers assigned to the `fileFilters` attribute.

To display an open dialog box, perform the following tasks:

1. Create an `AVOpenSaveDialogParamsRec` object.

   ```
   AVOpenSaveDialogParamsRec dialogParams;
   ```

2. Create an `AVFileFilterRec` object used to store a series of file type descriptors that define a file filter for an open or save dialog box. Secondly, create a pointer to an `AVFileFilterRec`.

   ```
   AVFileFilterRec filterRec,*filterRecP;
   ```

3. Create an `AVFileDescRec` object used to store file extensions in an open dialog box.

   ```
   AVFileDescRec descRec;
   ```

4. Set the `AVFileDescRec` object's `extension` attribute with the value `PDF`.

   ```
   strcpy (descRec.extension, "pdf");
   ```

5. Allocate the size of the `AVFileFilterRec` object's buffer by using the `memset` method. Next, set the `AVFileFilterRec` object's `fileDescs` and `numFileDescs` attributes. The `fileDescs` attribute defines file type and extension information. You can assign the address of the `AVFileDescRec` object to the `fileDescs` attribute. The `numFileDescs` attribute specifies the number of `AVFileDescRecs` objects assigned to the `fileDescs` attribute.

```
memset (&filterRec, 0, sizeof(AVFileFilterRec));
filterRec.fileDescs = &descRec;
filterRec.numFileDescs = 1;
```

6. Assign the `AVFileFilterRec` pointer with the address of the `AVFileFilterRec` object (this pointer is used to set an attribute that belongs to the `AVOpenSaveDialogParamsRec` object).

   ```
   filterRecP = &filterRec;
   ```

7. Set attributes that belong to the `AVOpenSaveDialogParamsRec` object. In this example, the `size`, `fileFilters`, and `numFileFilters` attributes are set. Assign the `fileFilters` attribute with the address of the pointer that points to `AVFileFilterRec`.

   ```
   memset (&dialogParams, 0, sizeof (AVOpenSaveDialogParamsRec));
   dialogParams.size = sizeof(AVOpenSaveDialogParamsRec);
   dialogParams.fileFilters = &filterRecP;
   dialogParams.numFileFilters = 1;
   ```

8. Set the `AVFileFilterRec` object's `filterDescription` attribute by invoking the `ASTextSetEncoded` method and passing the following arguments:

   - An `ASText` object that is used to store the string value.
   - A pointer to a `char` data type.
   - An `ASHostEncoding` value that specifies an encoding type.

   ```
   filterRec.filterDescription = ASTextNew();
   ASTextSetEncoded (filterRec.filterDescription, "Adobe PDF Files",
   ASScriptToHostEncoding(kASRomanScript));
   ```

9. Set the `AVOpenSaveDialogParamsRec` object's `windowTitle` attribute by invoking the `ASTextSetEncoded` method (see step 8 for a description of this method).

   ```
   dialogParams.windowTitle = ASTextNew();
   ASTextSetEncoded (dialogParams.windowTitle, "Select A PDF Document
   To Open",
   ASScriptToHostEncoding(kASRomanScript));
   ```

10. Display the open dialog box by invoking the `AVAppOpenDialog` method and passing the following arguments:

    - The address of an `AVOpenSaveDialogParams` object that represents the dialog box to open.
    - The address of an `ASFileSys` object. This method will populate the `ASFileSys` object with the file system in which the file that the user selects is located. This argument can be `NULL` if `kAVOpenSaveAllowForeignFileSystems` is not set as the `flags` value.
    - The address of a pointer that points to an `ASPathName` typedef. This argument is populated with the file that was select by the user.
    - The address of an `AVArraySize` object. This value can be NULL if `kAVOpenSaveAllowMultiple` is not set as the `flags` value.
    - The address of an `AVFilterIndex` object. This value can be NULL.

    The `AVAppOpenDialog` method returns `true` if the user clicks the action button (for example, the Open button). If the user clicks the Cancel button, the method returns `false`.

    ```
    ASPathName * pathName = NULL;
    ASBool bSelected = AVAppOpenDialog(&dialogParams,
   NULL,(ASPathName**)&pathName, NULL, NULL);
    ```

11. Release existing `ASText` objects by invoking the `ASTextDestroy` method.

```
ASTextDestroy (filterRec.filterDescription);
ASTextDestroy (dialogParams.windowTitle);
```

Note: This application logic is shown in "Opening a PDF document in an external window" on page 71.

Displaying a PDF document within a window

Before you attempt to display a PDF document in an external window, you must programmatic-ally create the window. (See "Creating a Window" on page 66.)

To display a PDF document within an external window, perform the following tasks:

1. Open a PDF file by invoking the `ASFileSysOpenFile` method and passing the following arguments:
   - An `ASFileSys` object that represents the file system in which the PDF file is located. (See "Creating an ASFileSys object" on page 153.)
   - An `ASPathName` object that represents the path in which the PDF file is located.
   - An `ASFileMode` object that represents the mode in which to open the file. For example, specify `ASFILE_READ` to open the PDF document in read mode.
   - The address of an `ASFile` object. This method populates this argument with the file that was opened (file information is obtained from the `ASPathName` object).

   If the `ASFileSysOpenFile` method is successful, then `0` is returned. Otherwise, an error value is returned.

2. Display the PDF document within the window by invoking the `AVDocOpenFromASFileWithParams` method and passing the following arguments:
   - An `ASFile` object that specifies a PDF file to display (you can use the same `ASFile` object whose address was passed to the `ASFileSysOpenFile` method).
   - An `ASText` object that specifies the text to display in the window's title bar. Create an `ASText` object by invoking the `ASTextFromScriptText` method.
   - The address of the `AVDocOpenParamsRec` object that you created. (See "Defining the parameters for an external window" on page 67.)

The following code example opens a PDF document in an external window. The application logic that is located within the `OpenExternalWindow` user-defined function creates the window and displays a PDF document within the window. A structure named `gDocInfo` that stores information such as the file to open is also defined.

This code example contains a user-defined method named `AVWndProc` that acts as a window procedure and another user-defined function named `InitializeWindowHandler` that registers the windows class that is specified as an argument to `CreateWindow`.

Also shown in this code example is application logic that displays an open dialog box that enables a user to select a PDF file to open. This application logic is located within a user-defined function named `OpenPDFFile`. (See "Displaying an open dialog box" on page 68.)

Example: *Opening a PDF document in an external window*

```
struct t_ExternDocInfo
{
   AVDoc doc;
```

```
   ASFile file;
} gDocInfo;

//Create a function that opens a PDF document in an external window
ACCB1 void ACCB2 OpenExternalWindow (void * data)
{
//Declare local variables
ASPathName pathName;
ASInt32 retVal;
ASFileSys myFileSys;
bool bWindowIsOpen = false;

//Create an AVDocOpenParamsRec object
AVDocOpenParamsRec params;

//Declare an ExternalDocServerCreationDataRec object
ExternalDocServerCreationDataRec creationData;

//Invoke CreateWindow to obtain a handle to a window
HWND externHWnd = CreateWindow ("ExternalWindow", "PDFViewer",
WS_OVERLAPPEDWINDOW,50, 50, 500, 500, 0, 0, gHINSTANCE, NULL);
if (externHWnd)
   {
   //Set attributes that belong to the AVDocOpenParamsRec object
   memset(&params, 0, sizeof(AVDocOpenParamsRec));
   params.size = sizeof(AVDocOpenParamsRec);
   params.useFrame = false;
   params.useVisible = true;
   params.visible = true;
   params.useServerType = true;
   params.serverType = "EXTERNAL";
   params.serverCreationData = (void*) &creationData;
   params.useViewType = true;
   params.viewType = "AVDocView";
   params.useReadOnly = true;
   params.readOnly = true;
   params.useSourceDoc = false;
   params.useViewDef = false;

   //Set attributes that belong to the ExternalDocServerCreationDataRec object
   memset(&creationData, 0, sizeof(ExternalDocServerCreationDataRec));
   creationData.size = sizeof(ExternalDocServerCreationDataRec);
   creationData.platformWindow = (ExternalDocWindowData)externHWnd;
   creationData.acrobatProc = NULL;
   creationData.crossDocLinkWithDestData = (void *)externHWnd;

   //Invoke OpenPDFFile to display an open dialog box that
   //enables a user to select a PDF file
   //Pass the address of the ASFileSys object
   pathName = OpenPDFFile(&myFileSys);

   //If PathName is valid
   if (pathName) {

   //Open the file specified in the file system
```

```
   retVal = ASFileSysOpenFile (myFileSys, pathName, ASFILE_READ,
   &gDocInfo.file);

   if (retVal ==0) {
      //Create a ASText object used to display in the new window
      ASText title = ASTextFromScriptText ("PDF Viewer", kASRomanScript);

      //Opens and displays a document from a file, using the specified
      //parameters to control the window's size, location, and visibility
      AVDocOpenFromASFileWithParams (gDocInfo.file, title, &params);
      ASTextDestroy(title);
      bWindowIsOpen = true;
} else
      AVAlertNote("Failed to open document.");
   }
 }
}

//Define the AVWndProc function
LRESULT CALLBACK AVWndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
if (msg == WM_DESTROY)
   memset (&gDocInfo, 0, sizeof(gDocInfo));
return DefWindowProc (hwnd, msg, wParam, lParam);
}

//Define the InitializeWindowHandler function
ASBool InitializeWindowHandler (void)
{
WNDCLASS wndClass ;

//This is the window class used to display the PDF
wndClass.cbClsExtra    = 0;
wndClass.hInstance     = gHINSTANCE;
wndClass.style         = CS_DBLCLKS;
wndClass.hCursor       = (HCURSOR)NULL;
wndClass.hbrBackground  = (HBRUSH)NULL;
wndClass.lpfnWndProc    = AVWndProc;
wndClass.hIcon          = (HICON)NULL;
wndClass.lpszMenuName   = NULL;
wndClass.cbWndExtra     = 0;
wndClass.lpszClassName  = "ExternalWindow";

if (!RegisterClass(&wndClass))
   return false;
return true;
}

//Display an open dialog box that enables a user to select a PDF file
ASPathName OpenPDFFile(ASFileSys * ASF)
{
//Declare an AVOpenSaveDialogParamsRec object
AVOpenSaveDialogParamsRec dialogParams;

//Create local variables
```

```
AVFileFilterRec filterRec,*filterRecP ;
AVFileDescRec descRec;
ASPathName * pathName = NULL;
ASFileSys fileSys = NULL;
ASBool bSelected = false;
char errorBuf[256];

//Set up the PDF file filter description
strcpy (descRec.extension, "pdf");
descRec.macFileType = 'PDF ';
descRec.macFileCreator = 'CARO';

//Set attributes that belong to the AVFileFilterRec object
memset (&filterRec, 0, sizeof(AVFileFilterRec));
filterRec.fileDescs = &descRec;
filterRec.numFileDescs = 1;
filterRecP = &filterRec;

//Set attributes that belong to the AVOpenSaveDialogParamsRec object
memset (&dialogParams, 0, sizeof (AVOpenSaveDialogParamsRec));
dialogParams.size = sizeof(AVOpenSaveDialogParamsRec);
dialogParams.fileFilters = &filterRecP;
dialogParams.numFileFilters = 1;

DURING
   //Set the AVFileFilterRec object's filterDescription attribute
   filterRec.filterDescription = ASTextNew();
   ASTextSetEncoded (filterRec.filterDescription, "Adobe PDF Files",
   ASScriptToHostEncoding(kASRomanScript));

   //Set the AVOpenSaveDialogParamsRec object's windowTitle attribute
   dialogParams.windowTitle = ASTextNew();
   ASTextSetEncoded (dialogParams.windowTitle, "Select A PDF Document To
Open",ASScriptToHostEncoding(kASRomanScript));

   //Display the Open dialog box - pass the address of the ASFileSys object
   bSelected = AVAppOpenDialog(&dialogParams, &fileSys,
(ASPathName**)&pathName, NULL, NULL);

HANDLER
   //Display an error message to the user
   ASGetErrorString (ASGetExceptionErrorCode(), errorBuf, 256);
   AVAlertNote (errorBuf);
END_HANDLER

//Destroy the ASText object then return
ASTextDestroy (filterRec.filterDescription);
ASTextDestroy (dialogParams.windowTitle);

//Point the ASFileSys argument to the address of the ASFileSys object
*ASF = fileSys;

return bSelected ? *pathName : NULL;
}
```

Note: The `OpenExternalWindow` function can be invoked from an Adobe Reader or Acrobat user interface component, such as a toolbar button or a menu command. Ensure that the `InitializeWindowHandler` method is invoked before the `OpenExternalWindow` method; otherwise, the window class is not registered and the PDF file is not displayed in the external window.

Determining the PDF version of a document

All PDF documents contain PDF version information in the header, such as 1.7. Additionally, PDF files that conform to ISO 32000 can include PDF version extension information that indicates that the PDF document may contain PDF extensions added by third-party developers.

The Acrobat API provides methods that let you obtain the PDF version and the PDF version extension information.

PDF version

The `PDDocGetVersion` method lets you obtain the PDF version of a PDF document. This PDF version indicates the PDF specification to which the document conforms, such as 1.4 or 1.7.

PDF version extensions

Acrobat 8.1 and later add extensions to the PDF specification. These extensions are specified in the *Adobe Supplement to ISO 32000*. Each extension in that supplement is associated with a base version and an extension level. The `Extensions` dictionary located in the `Catalog` dictionary specifies PDF extensions added by Adobe or others.

For example, Acrobat 8.1 added a PDF extension to 3D annotations that enabled support for the PRC format, a highly compressed 3D representation. If a PDF document contains a 3D annotation that specifies a PRC file, the PDF document also contains an `Extensions` dictionary in the `Catalog` dictionary that specifies a base version of 1.7 and an extension level of "1" or greater. The following PDF segment shows the appearance of such an `Extensions` dictionary.

```
%PDF 1.7
<</Type /Catalog
   /Extensions
   <</ADBE
      << /BaseVersion /1.7 /ExtensionLevel 3 >>
   >>
>>
```

Setting the extension level of a document

When you add PDF extensions specified by Adobe or others, you must ensure that the `Catalog` dictionary contains an `Extensions` dictionary that describes the greatest extension level that can appear in the PDF document. The `Extensions` dictionary can contain one dictionary for each company or other entity that adds PDF extensions.

To set the extension level, use the `Cos` object methods described in Working with Cos Objects.

Getting the extension level of a document

Beginning with Acrobat 9.0, the `PDDocGetVersionEx` method lets you obtain the Adobe-specific version extension information and the PDF version.

To obtain version extension information introduced by entities other than Adobe, use the `Cos` object methods described in Working with Cos Objects.

Bridging core API layers

To operate on the contents of a PDF document, and not just manipulate its representation in Adobe Reader or Acrobat, you must use a `PDDoc` object. For example, to create an annotation for a PDF document, you must create a `PDDoc` object. (See "Creating Annotations" on page 100.)

You can create a `PDDoc` object by using an `AVDoc` object. To get a `PDDoc` object that corresponds to an `AVDoc` object, invoke the `AVDocGetPDDoc` method. This method is referred to as a bridge method because it allows your plugin to access other layers from the AV layer. There are several bridge methods which connect the different API layers (AV, PD, and Cos).

All `AVDoc` objects 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 plugin 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.

Creating a PDDoc object

You can access a PDF file's contents without displaying the PDF file in Acrobat or Adobe Reader. To perform this task, create a `PDDoc` object by invoking the `PDDocOpen` method and passing the following arguments:

- An `ASPathName` object that specifies the PDF file on which the `PDDoc` is based. For information about creating an `ASPathName` object, see "Opening PDF documents" on page 64.
- An `ASFileSys` object that represents the file system in which the PDF file is located. (See "Creating an ASFileSys object" on page 153.)
- A `PDAuthProc` authorization callback that is invoked only if the file has been secured (that is, if the file has either the user or the master password set). This callback must obtain required information to determine whether the user is authorized to open the file. You can pass `NULL` if an authorization callback is not required. For information about callbacks, see "Using callback functions" on page 29.
- An `ASBool` value. When set to `true`, an attempt to repair a damaged file is performed. If `false`, an attempt to repair a damaged file is not performed.

The following code example creates a `PDDoc` object that is based on a PDF file named PurchaseOrder.pdf and located in the root of the C drive. This code example is located within a user-defined function named `getDocument`.

Example: *Creating a PDDoc object*

```
PDDoc getDocument()
{
   //Declare a PDDoc object
   PDDoc myPDDoc;
```

```
   //Declare a PDAuthProc callback
   PDAuthProc myAuthProcPtr;

   //Specify the PDF file on which to base the PDDoc object
   const char * myPath = "C:\\PurchaseOrder.pdf";
   ASAtom pathType = ASAtomFromString("Cstring");

   //Create an ASPathName object
   ASFileSys fileSys = ASGetDefaultFileSysForPath(pathType, myPath);
   ASPathName pathName = ASFileSysCreatePathName(fileSys,pathType,
   myPath,NULL);

   //Create the authentication callback
   myAuthProcPtr = ASCallbackCreateProto (PDAuthProc,&authProc);

   //Create a PDDoc object
   myPDDoc = PDDocOpen (pathName, fileSys, myAuthProcPtr, false);

   //Release the callback
   ASCallbackDestroy (myAuthProcPtr);
   return myPDDoc;
}

ASBool authProc(PDDoc TheDoc)
{
   //Define business logic that authenticates the user
   return true;
}
```

Creating a PDDoc object based on an open PDF document

You can create a `PDDoc` object that is based on a PDF document that is currently open by performing the following tasks:

1. Create an `AVDoc` object by invoking the `AVAppGetActiveDoc` method. This method gets the frontmost document located in Adobe Reader or Acrobat.
2. Create a `PDDoc` object by invoking the `AVDocGetPDDoc` method. This method requires an `AVDoc` object and returns a `PDDoc` object.

The following code example creates a `PDDoc` object that is based on an open PDF document.

Example: *Creating a PDDoc object based on an open PDF document*

```
AVDoc avDoc = AVAppGetActiveDoc();
PDDoc myPDDoc = AVDocGetPDDoc(avDoc);
```

Accessing non-PDF files

You can use the Acrobat core API to access non-PDF files. Your plugin can open a non-PDF file, write data to it, and then read the data at a later time by using `ASFile` methods. For example, your plugin can use a text file to track log information or other type of information.

The following code example opens a file, writes data to it, and reads data from it. The following Acrobat core API methods are invoked in this code example:

- `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.

In the following code example, the `CreateDataFile` user-defined function creates a file and writes data to it. The `ReadDataFile` user-defined function opens a file and reads data from it.

Example: *Accessing a non-PDF file*

```
//Create a global character pointer
char* gDataBuf = "This is some data in the file.";

ACCB1 void ACCB2 ExeProc (void* data)
{
   CreateDataFile("C:\\DataFile.txt");
   ReadDataFile("C:\\DataFile.txt");
}

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

```
   }

//Read data from a file
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 (NULL, 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(Data);
   return;
}

//Create a platform-independent path
ASPathName MakeCrossPlatformASPathName (char* platformPathname)
{
ASPathName ThePathName = NULL;
ThePathName = ASFileSysCreatePathName (NULL,ASAtomFromString("Cstring"),
platformPathname, 0);
return ThePathName;
}
```

Printing documents

You can use the Acrobat core API to print documents by using one of the following methods:

- `AVDocPrintPages`
- `AVDocPrintPagesWithParams`

The `AVDocPrintPages` method prints a document without displaying any user dialog boxes. The current printer, page settings, and job settings are used. Printing is complete when this method returns.

The `AVDocPrintPagesWithParams` method prints a document with a full range of options. Printing is complete when this method returns. It performs embedded printing; that is, it allows a PDF page to print within a bounding rectangle on a page. It allows interactive printing to the specified printer.

To print a document with a range of options, invoke the `AVDocPrintPagesWithParams` method and pass the following arguments:

- An `AVDoc` object that represents the PDF document from which to print page. For information about creating an `AVDoc` object, see "Opening PDF documents" on page 64.
- An instance of the `AVDocPrintParams` data structure that defines printing parameters. (See the Acrobat and PDF Library API Reference.)

Working with the PDF/X format

Beginning PDF Library XI, you can convert an existing PDF file to the PDF/X format. The converter aims at creating an output PDF/X document that is self-contained and whose visual fidelity is maintained over time. The converter manages operations like font embedding, changes in colors while printing, etc., to give a seamless PDF/X conversion process. PDF/X-1a:2001 and PDF/X-3:2003 are the two PDF/X standards that are supported by the PDF processor plugin. For files containing objects that cannot be converted as per the PDF/X standards, the conversion process is aborted.

Note: PDFL SDK XI does not support the conversion of PDF packages or of PDF portfolios to the PDF/X format.

The following lists how various file elements are handled during conversion:

- Color Spaces: All color spaces in the input file are converted to PDF/X compliant color spaces. For PDF/X-3:2003, conversions to the US SWOP colorspace is supported.
- AcroForms: AcroForms are flattened in the PDF/X compliant output file. This may result in loss of interactivity if the source file includes annotations. In the case of text fields, only the visible text is saved.
- Digital Signatures: Digital signatures are flattened and replaced by equivalent graphics operators with the same visual appearance.
- Annotations and Actions: The PDF/X standard does not allow the use of annotations or actions inside the print area. Any actions present in the input file will not be available in the output PDF/X file Annotations, if present, are moved outside the print area in the converted PDF/X file. You can choose to turn off the relocation of annotations which will result in the removal of annotations altogether.
- Tagged PDF: PDF files that include tags are not supported. All tagging information is lost when a tagged PDF file is converted to the PDF/X format.
- Fonts: The conversion is aborted if fonts in the input file are not embedded, the fonts are not present on the system, or the font embedding permissions are not available. Any invisible text in the input file will not be available in the output PDF/X file.
- Transparency: Transparency is not supported in the PDF/X format. All transparency is flattened when a file is converted to the PDF/X format

The following APIs are available for conversion using the PDF Processor:

- PDFProcessorConvertAndSaveToPDFA

- PDFProcessorConvertToPDFA
- PDFProcessorTerminate
- PDFProcessorConvertAndSaveToPDFX
- PDFProcessorConvertToPDFX.

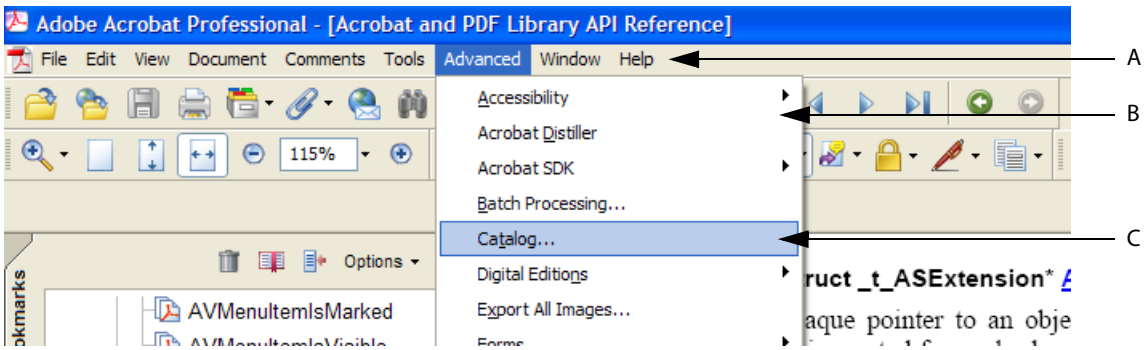
See the Acrobat and PDF Library API Reference for more information.

6 Creating Menus and Menu Commands

The Acrobat core API supports creating new menu commands and modify existing menus. A menu command lets a user interact with your plugin by clicking a menu command that is displayed in an Adobe Reader or Acrobat menu. When the user clicks the menu command, application logic that is defined in the callback function associated with the menu command is executed. It is your responsibility to define the application logic that is located in the callback function.

About menus

Adobe Reader and Acrobat have a standard menu bar and menu that let a user invoke specific functionality. You can, for example, click the Open command that is located on the File menu to open an existing PDF file. Using the Acrobat core API, you can programmatic-ally modify the menu bar. The following illustration shows the menu bar located in Acrobat.



The following table describes the arrows in the previous illustration.

Letter	Description
A	The Acrobat menu bar.
B	An Acrobat menu.
C	A menu command.

About AVmenubar typedefs

An `AVmenubar` typedef represents the menu bar located within Adobe Reader or Acrobat and contains a list of all menus. A plugin can only have one `AVmenubar` object. You can add new menus to the menu bar, remove menus from the menu bar, or hide the menu bar by using the Acrobat core API.

About AVMenu typedefs

An `AVMenu` typedef represents a menu in the Adobe Reader or Acrobat menu bar. You can create new menus, add menu commands to a menu, and remove menu items. Deleting an `AVMenu` removes it from the menu bar (if it was attached) and deletes all the menu items it contains.

Submenus (also called pull right menus) are `AVMenu` objects that are attached to an `AVMenuItem` object instead of to the menu bar.

Each menu has a title and a language-independent name. The title is the string that appears in the user interface, while the language-independent name is the same regardless of the language used in the user interface. Language-independent names enable a plugin to locate a specific menu, such as the File menu, without knowing, for example, that it is called Fichier in French.

You are strongly encouraged to begin your language-independent menu names with the plugin name (separated by a colon) to avoid name collisions when more than one plugin is present. For example, if a plugin is named myPlug, it may add a menu whose name is `myPlug:Options`. (See "Using plugin prefixes" on page 30.)

About AVMenuItem typedefs

An `AVMenuItem` is a menu command within a menu. It contains the following attributes:

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

An `AVMenuItem` object can also serve as a separators between menu commands. You are encouraged to position your plugin menu commands relative to a separator. This helps ensure that if a block of menu commands is moved in a future version of Acrobat, your plugin menu commands also are moved.

A plugin can simulate a user selecting a menu command by invoking the `AVMenuItemExecute` method. If the menu command is disabled, the `AVMenuItemExecute` method returns without performing a task. The `AVMenuItemExecute` method works even when the menu item is not displayed (for example, if it has not been attached to a menu or if the menu bar is not visible). You can set the attributes of all menu commands that you create; however, do not modify the `Execute` procedure of the Acrobat built-in menu commands.

Your plugin can specify menu command names using either the names seen by a user, or language-independent names. The latter allows your plugin to locate a specific menu command without knowing, for example, that it is called Imprimer in French.

You are strongly encouraged to avoid name collisions when naming menu commands by using your plugin name and a colon. For example, if a plugin is named `myPlug`, you can name a menu command `myPlug:Open` or `myPlug:Find`. (See "Using plugin prefixes" on page 30.)

Adding menu commands to menus

You can use the Acrobat core API to add a new menu command to an existing menu. For example, you can add a menu named Acrobat SDK to the Adobe Reader or Acrobat menu bar.

To add a new menu command to an existing menu, perform the following tasks:

1. Retrieve the Adobe Reader or Acrobat menu bar that is represented by an `AVmenubar` object by invoking the `AVAppGetMenubar` method.

```
AVMenubar Themenubar = AVAppGetMenubar();
```

2. Retrieve the menu that will contain the new menu command by invoking the `AVMenubarAcquireMenuByName` method and passing the following arguments:
   - An `AVmenubar` object that represents the menu bar.
   - The name of the menu. For example, to reference the File menu, specify `File`.

   This method returns an `AVMenu` object that corresponds to the specified menu name.

   ```
   AVMenu FileMenu = AVmenubarAcquireMenuByName (Themenubar, "File");
   ```

   If the menu does not exist, you can programmatically create it (see the next step).

3. If necessary, programmatically create a new menu by invoking the `AVMenuNew` method and passing the following arguments:
   - A string value that represents the text displayed in Adobe Reader or Acrobat.
   - A unique, language-independent name for the menu. (See "Using plugin prefixes" on page 30.)
   - An `ASExtension` value that registers the menu. (See the Acrobat and PDF Library API Reference.)

   The `AVMenuNew` method returns an instance of `AVMenu`.

   ```
   AVMenu NewMenu = AVMenuNew ("Acrobat SDK", "ADBE:Acrobat_SDK",
   gExtensionID);
   ```

   After you create a new menu, you must attach it to the menu bar. (See "Adding a menu command to a new menu" on page 86.)

4. Create a new menu command by invoking the `AVMenuItemNew` method and passing the following arguments:
   - A string value that represents the menu command's text. On Windows only, you can specify a keyboard shortcut by including an ampersand character (&) in the title.[I wonder whether an example would help here.] In Acrobat or Adobe Reader, an underscore character (_) is placed under the character following the ampersand (*char*).[I wonder whether an example would help here.] The user can then press alt+*char* to select the item.
   - The language-independent name of the menu command to create. (See "Using plugin prefixes" on page 30.)
   - An `AVMenu` object that represents a submenu for which this menu command is the parent. Pass `null` if this menu item does not have a submenu.
   - A Boolean value. If `true`, the menu item is visible only when the user selects Full Menus. If `false`, the menu item is visible for both Full Menus and Short Menus modes. This value is ignored in Adobe Reader or Acrobat 3.0 or later.
   - The key to use as a shortcut for the menu command (an ASCII character). Use `NO_SHORTCUT` if the menu command does not have a shortcut.
   - Modifier keys, if any, that are used as part of the shortcut. Must be an OR of the Modifier Keys values, except that `AV_COMMAND` cannot be specified.
   - An `AVIcon` object that represents the icon to show in the menu command, or `null` if no icon is shown. In Windows, a valid icon is a 24x24 sample monochrome `HBITMAP`. In Mac OS, an icon is a handle to a standard SICN resource. For information about creating an `AVIcon` object, see "Creating toolbar buttons" on page 92.

- An `ASExtension` value that registers this menu command. For information about an `ASExtension` value, see the Acrobat and PDF Library API Reference.

The `AVMenuItemNew` method returns an `AVMenuItem` object.

```
AVMenuItem menuItem = AVMenuItemNew ("Show Message", "ADBE:ExternWin",
NULL, true, NO_SHORTCUT, 0, NULL, gExtensionID);
```

5. Attach the menu command to a menu by invoking the `AVMenuAddMenuItem` method and passing the following arguments:
   - An `AVMenu` object to which a menu command is attached.
   - An `AVMenuItem` object that is attached.
   - The location in the menu that specifies where the command is added. You can specify `APPEND_MENUITEM` to append the menu command to the end of the menu.

   If this method is successful, the menu command is added to the specified menu.

   ```
   AVMenuAddMenuItem (FileMenu, menuItem, APPEND_MENUITEM);
   ```

6. Release the typedef instances to free memory. To release an `AVMenu` instance, invoke the `AVMenuRelease` method and pass the `AVMenu` instance. To release an `AVMenuItem` instance, invoke the `AVMenuItemRelease` method and pass the `AVMenuItem` instance.

Adding a menu command to an existing menu

The following code example creates a new menu command that displays the text `Show Message` and attaches it to the File menu.

Example: *Adding a menu command to an existing menu*

```
//Declare menu variables
AVMenubar Themenubar = NULL;
AVMenu FileMenu = NULL;
AVMenuItem NewMenuCommand = NULL;

//Retrieve the menu bar in Adobe Reader or Acrobat
Themenubar = AVAppGetMenubar();

//Retrieve the File menu
FileMenu = AVMenubarAcquireMenuByName(Themenubar, "File");

//Create a new menu command
NewMenuCommand = AVMenuItemNew("Show Message", "ADBE:ExternWin", NULL,
true, NO_SHORTCUT, 0, NULL, gExtensionID);
if (NewMenuCommand == NULL)
{
   AVAlertNote ("Unable to create the menu command");
   AVMenuItemRelease(NewMenuCommand);
   return;
}

//Attach the new menu command to the File menu
AVMenuAddMenuItem (FileMenu, NewMenuCommand, APPEND_MENUITEM);

//Release the typedef instances
```

```
AVMenuItemRelease(NewMenuCommand);
AVMenuRelease(FileMenu);
```

Note: This code example creates a new menu command that displays Show Message in the File menu. However, before the menu command performs an action, you have to create a callback menu function. (See "Creating menu callback functions" on page 87.)

Tip: To see how the global `gEntensionID` variable is declared, see the plugin samples that accompany the Acrobat SDK.

Adding a menu command to a new menu

The following code example creates a new menu command that displays the text `Show Message` and attaches it to a new menu. The new menu is attached to the menu bar by invoking the `AVmenubarAddMenu` method.

Example: *Adding a menu command to a new menu*

```
//Declare menu variables
AVMenubar Themenubar = NULL;
AVMenu NewMenu = NULL;
AVMenuItem NewMenuCommand = NULL;

//Retrieve the menu bar in Adobe Reader or Acrobat
Themenubar = AVAppGetMenubar();

//Create a new menu
NewMenu = AVMenuNew("New Menu", "ADBE:NewMenu", gExtensionID);
if (NewMenu == NULL)
{
   AVAlertNote ("Unable to create the menu");
   AVMenuRelease (NewMenu);
   return ;
}

//Create a new menu command
NewMenuCommand = AVMenuItemNew("Show Message", "ADBE:ExternWin", NULL,
true, NO_SHORTCUT, 0, NULL, gExtensionID);
if (NewMenuCommand == NULL)
{
   AVAlertNote ("Unable to create the menu command");
   AVMenuItemRelease(NewMenuCommand);
   return;
}

//Attach the menu item to the menu and the menu to
//the menu bar
AVMenuAddMenuItem (NewMenu, NewMenuCommand, 0);
AVMenubarAddMenu (Themenubar, NewMenu, APPEND_MENU);

//Release the typedef instances
AVMenuItemRelease(NewMenuCommand);
AVMenuRelease(NewMenu);
```

Note: If you plan to add a submenu to a menu command, you must create the submenu before creating the menu command.

Creating menu callback functions

When creating menus, you must create menu callback functions that are invoked by Adobe Reader or Acrobat. These types of callback functions can be created:

Execute: Invoked by Adobe Reader or Acrobat in response to a user selecting a menu command. This callback is required.

Compute-enabled: This optional callback is invoked by Adobe Reader or Acrobat when determining whether to enable the menu command.

Compute-marked: This optional callback is invoked by Adobe Reader or Acrobat when determining whether the menu command should be checked.

For the purpose of this discussion, a simplistic user-defined function named `ShowMessage` is introduced. This method displays a message box by invoking the `AVAlertNote` method. The following code example shows the body of the `ShowMessage` function.

```
ACCB1 void ACCB2 ShowMessage (void* data)
{
   AVAlertNote ("A menu command was selected.");
}
```

The data parameter for this and the other callbacks can be used to maintain private data for the menu command. Notice that this user-defined function is declared using the `ACCB1` and `ACCB2` macros. (See "Using callback functions" on page 29.)

For each callback that you create, you declare pointers to callbacks that are defined in the Acrobat core API:

```
AVExecuteProc ExecProcPtr = NULL;
AVComputeEnabledProc CompEnabledProcPtr = NULL;
AVComputeMarkedProc CompMarkedProcPtr = NULL;
```

`AVExecuteProc` is a callback that you can create that is invoked by Acrobat or Adobe Reader when a user selects a menu item. `AVComputeEnabledProc` is a callback that you can create that is invoked by Adobe Reader or Acrobat when determining whether to enable the menu command. `AVComputeMarkedProc` is a callback that you can create that is invoked by Adobe Reader or Acrobat when determining whether the menu command should be checked.

After you create a pointer, such as a pointer that points to `AVExecuteProc`, you can invoke the `ASCallbackCreateProto` macro that is defined in the Acrobat core API to convert a user-defined function to an Acrobat callback function. For example, you can invoke `ASCallbackCreateProto` to convert `ShowMessage` to a callback function. The `ASCallbackCreateProto` macro requires the following arguments:

- The callback type. For example, you can pass `AVExecuteProc`.
- The address of the user-defined function to convert to a callback function.

The `ASCallbackCreateProto` macro returns a callback of the specified type that invokes the user-defined function whose address was passed as the second argument. The following code example shows the `ASCallbackCreateProto` macro converting the `ShowMessage` user-defined function to a `AVExecuteProc` callback.

```
AVExecuteProc ExecProcPtr = NULL;
ExecProcPtr = ASCallbackCreateProto(AVExecuteProc, &ShowMessage);
```

After you create an `AVExecuteProc` callback, invoke the `AVMenuItemSetExecuteProc` method to associate a menu command with a callback. That is, when a user selects a specific menu command, Acrobat or Adobe Reader will invoke the user-defined function whose address was passed to the `ASCallbackCreateProto` macro. The `AVMenuItemSetExecuteProc` method requires the following parameters:

- An `AVMenuItem` instance that represents the menu command.
- An `AVExecuteProc` that represents the callback function.
- The address of a user-defined data structure that can be passed to the user-defined function.

When you are done with a menu callback, you can invoke the `ASCallbackDestroy` method to release memory that it consumes. The following code example creates callback functions for menu commands.

Example: *Creating menu callback functions*

```
/* Display a message box */
ACCB1 void ACCB2 ShowMessage (void* data)
{
   AVAlertNote ("A menu command was selected.");
}
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)
{

//Declare menu callbacks
AVExecuteProc ExecProcPtr = NULL;
AVComputeEnabledProc CompEnabledProcPtr = NULL;
AVComputeMarkedProc CompMarkedProcPtr = NULL;

//Declare menu variables
AVMenu FileMenu = NULL;
AVMenuItem NewItem = NULL;

//Retrieve the menu bar in Adobe Reader or Acrobat
AVMenubar Themenubar = AVAppGetMenubar ();

//Create menu callbacks
ExecProcPtr = ASCallbackCreateProto (AVExecuteProc, &ShowMessage);
CompEnabledProcPtr = ASCallbackCreateProto (AVComputeEnabledProc,
&ComputeEnabledProc);
```

```
CompMarkedProcPtr = ASCallbackCreateProto (AVComputeMarkedProc,
&ComputeMarkedProc);

//Retrieve the File menu
FileMenu = AVmenubarAcquireMenuByName (Themenubar, "File");
if (FileMenu)
{
//Create a new menu item
   NewItem = AVMenuItemNew ("Show Message", "ADBE:ExternWin", NULL, true,
   NO_SHORTCUT, 0, NULL, gExtensionID);
   if (NewItem == NULL)
   {
      AVAlertNote ("Unable to create a menu item, not loading.");
      return false;
   }
   AVMenuItemSetExecuteProc (NewItem, ExecProcPtr, NULL);
   AVMenuItemSetComputeEnabledProc (NewItem,
   CompEnabledProcPtr,NULL);
   AVMenuItemSetComputeMarkedProc (NewItem,
   CompMarkedProcPtr,NULL);
   AVMenuAddMenuItem (FileMenu, NewItem, 1);
   AVMenuRelease (FileMenu);
   return true;
}
else return false;
}
ACCB1 ASBool ACCB2 PluginUnload (void)
{
ASCallbackDestroy (ExecProcPtr);
ASCallbackDestroy (CompEnabledProcPtr);
ASCallbackDestroy (CompMarkedProcPtr);
return true;
}
```

Note: Notice that the application logic that creates a menu command is located in the `PluginInit` procedure. (See "About plugin initialization" on page 26.)

Determining if a menu item can be executed

In previous versions of Adobe Reader and Acrobat, it was possible for a document to execute a menu item in the viewing application using a Named action or the `app.execMenuItem` JavaScript method. These two features, referred to as `ExecMenu` expose all menu items to the document, potentially allowing a malicious document to compromise a user's privacy or system. For example, it was possible to use the `app.execMenuItem` JavaScript method to obtain data from a document by creating the equivalent to a user selecting the menu sequence of Select All, Copy, and Paste.

Acrobat and Adobe Reader 8.0 and later contain a list of menu items that can be executed using `ExecMenu`. Any menu item not on the list cannot be programmatically executed.

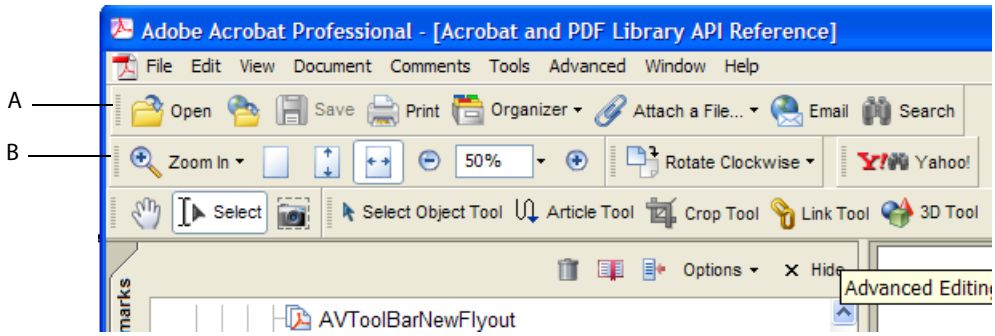
You can determine if a menu item can be programmatically executed by invoking the `AVMenuItemIsScriptable` method and passing an `AVMenuItem`. This method returns a Boolean value. That is, if the menu item that corresponds to the `AVMenuItem` argument can be executed, `True` is returned; otherwise, `False` is returned.

7 Creating Toolbars and Buttons

This chapter explains how to use the Acrobat core API to create new toolbars and toolbar buttons and to modify existing ones. For example, you can create a new button, attach it to an existing toolbar, and associate the button with a specified callback function that is executed when the user clicks the button.

About toolbars

Adobe Reader and Acrobat consists of various toolbars that enable a user to invoke specific functionality. For example, you can click the `Open` button that is located on the `File` toolbar to open an existing PDF document. The following illustration shows two toolbars located in Acrobat.



The following table describes the arrows in the previous illustration.

Letter	Description
A	The File toolbar
B	The Zoom toolbar.

Tip: You can obtain the display name of the toolbar by pointing the mouse to the left portion of a toolbar.

Note: The display name is the name that is displayed in Acrobat or Adobe Reader. However, toolbars also have internal names that may differ. That is, in some cases, the internal name and display name of a toolbar are different and in other cases they are the same. For example, consider the Help toolbar. The display name is Help and the internal name is HowTo. An internal name is used to programmatically retrieve a toolbar. (See "Retrieving toolbars" on page 91.)

About AVToolBar typedefs

An `AVToolBar` represents a toolbar that is located in Adobe Reader or Acrobat. A plugin can add buttons to and remove buttons from a toolbar, show or hide toolbars, and create new toolbars. Because screen space is limited on many monitors, it is recommended that you only create buttons that are necessary.

You can add a button to a toolbar by specifying the relative position of the button (before or after) to an existing button. A plugin controls the toolbar upon which a button will appear by placing the button next to an existing one already in the appropriate location.

About AVToolButton typedefs

An `AVToolButton` represents a button that is located within a toolbar. Like menu items, a callback function that is executed when the button is clicked must be defined. (See "Creating toolbar button callback functions" on page 98.)

A plugin can invoke a button as if a user clicked it. Buttons can be enabled (selectable) or disabled (grayed out), and can be marked (selected). Each button also has an icon that appears in the toolbar. Normally, all tools are persistent and remain selected indefinitely. The Control key (Windows) or Option key (Mac OS) can be used to select a tool for one-shot use. Plugins should follow this convention to add buttons.

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

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

Retrieving toolbars

You can use the Acrobat core API to retrieve an existing toolbar that appears in Adobe Reader or Acrobat. After you retrieve a toolbar, you can perform additional tasks such as attaching a button. (See "Attaching a button to a toolbar" on page 96.)

You retrieve a specific toolbar by invoking the `AVAppGetToolBarByName` method. This method requires a constant character pointer that specifies the internal name of a toolbar and returns an `AVToolBar` object that corresponds to the toolbar. If the name cannot be found, this method returns `NULL`.

The following table lists toolbar names that appear in Acrobat and Adobe Reader (Adobe Reader does not contain all the toolbars that Acrobat does). The Display Name column specifies the toolbar name that appears in Acrobat or Adobe Reader. The Internal Name column specifies the value that you must pass to the `AVAppGetToolBarByName` method to retrieve the toolbar.

Display Name	Internal Name	Description
Advanced Editing	Editing	Contains buttons that enable you to perform advanced editing operations such as using the Crop tool.
Basic	BasicTools	Contains buttons that enable you to perform basic operations such as using the Hand tool.
Commenting	Commenting	Contains buttons that enable you to perform commenting operations such as using the Notes tool.
Drawing Markups	AdvCommenting	Contains buttons that enable you to perform drawing operations such using the Arrow tool.
Edit	UndoRedo	Contains buttons that enable you to perform editing operations such as checking the spelling.
File	File	Contains buttons that enable you to perform file operations such as opening a PDF file.

Display Name	Internal Name	Description
Help	HowTo	Contains a button that enables you to access online help topics.
Measuring	Measuring	Contains buttons that enable you to perform measuring operations such as using the Distance tool.
Navigation	Navigation	Contains buttons that enable you to perform navigation operations such as skipping to the next page.
Print Production	PrintProduction	Contains buttons that enable you to perform print operations such as viewing an output preview.
Rotate View	Rotate	Contains a button that enables you to rotate a PDF document.
Tasks	Tasks	Contains buttons that enable you to perform operations such as digitally signing a document.
Typewriter	Typewriter	Enables you to type text on a PDF document.
Zoom	Viewing	Contains buttons that enable you to perform viewing operations such as zooming in on a document.

The following code example retrieves the Tasks toolbar.

Example: *Retrieving a toolbar by name*

```
//Retrieve the Tasks toolbar
const char * toolbarName= "Tasks" ;
AVToolBar myToolBar = AVAppGetToolBarByName(toolbarName);
```

Note: You can invoke the `AVAppGetToolBar` method to return an `AVToolbar` object that is based on the Advanced Editing toolbar.

Creating toolbar buttons

You can create a new button that you can attach to a new or existing toolbar. To create a new button, invoke the `AVToolButtonNew` method and pass the following arguments:

- An `ASAtom` object that specifies the button's name.
- An `AVIcon` object that represents the button's icon. If a button does not have an icon, the button appears with a gray background.
- An `ASBool` value that you can set to `true` or `false`. If `true`, the button is shown only when the user selects 'Full menus'. If `false`, shows in both 'Full menu' and 'Short menu' modes. This argument is ignored in Acrobat 3.0 or later.
- An `ASBool` value that you can set to `true` or `false`. If `true`, the new button is a separator used to leave space between groups of related buttons. Callback functions are ignored and a user cannot click on a separator. If `false`, the button is normal.

The `AVToolButtonNew` method returns an `AVToolButton` object. You must attach this button to a toolbar in order to view it. (See "Attaching a button to a toolbar" on page 96.)

It is strongly recommended that you create an `AVIcon` object when creating a new button. To create an `AVIcon` object, you must invoke platform specific APIs. That is, you do not invoke methods that belong to the Acrobat core API. If, for example, you are working on Windows, you can invoke a Win32 API method named `LoadBitmap`. Likewise, if you are working on Mac OS, you can invoke `SafeGetResource`.

The following code example shows how to create an `AVIcon` object that is based on a bitmap resource named `IDB_BITMAP1`.

```
AVIcon myIcon = (AVCursor)LoadBitmap(gHINSTANCE,
MAKEINTRESOURCE(IDB_BITMAP1));
```

The `gHINSTANCE` object is an instance of `HINSTANCE` and is declared in the PIMain.c file. In addition to creating a new icon, you can also retrieve an existing icon appearing on a toolbar button. (See "Retrieving existing toolbar buttons" on page 95.)

Once you create an `AVIcon` object, you can create a new toolbar button. The following code example creates a new toolbar button.

Example: *Creating a toolbar button*

```
//Declare an AVToolButton object
AVToolButton MyButton = NULL;

//Create an AVIcon object
AVIcon myIcon = (AVCursor)LoadBitmap(gHINSTANCE,
MAKEINTRESOURCE(IDB_BITMAP1));

//Create a new button
MyButton = AVToolButtonNew(ASAtomFromString("MyExtn:MyButton"), myIcon,
false, false);
```

Setting help text for a button

A button's help text appears when the mouse scrolls over and pauses on a toolbar button. A small pop-up window appears with a text message. To set a button's help text, invoke the `AVToolButtonSetHelpText` method and pass the following arguments:

- An `AVToolButton` object that represents a button for which the help text is set.
- A constant character pointer that specifies the button's help text value.

The following code example sets a button's help text.

Example: *Setting a button's help text*

```
//Declare an AVToolButton object
AVToolButton MyButton = NULL;

//Create an AVIcon object
AVIcon myIcon = (AVCursor)LoadBitmap(gHINSTANCE,
MAKEINTRESOURCE(IDB_BITMAP1));

//Create a new button
MyButton = AVToolButtonNew(ASAtomFromString("MyExtn:MyButton"), myIcon,
FALSE, FALSE);

//Set a button's help text
```

```
const char * helpText = "Open PDF in external window" ;
AVToolButtonSetHelpText (MyButton, helpText);
```

Setting label text

A button's label text is the text that is displayed beside the button. To set a button's label, invoke the `AVToolButtonSetLabelText` method and pass the following arguments:

- An `AVToolButton` object that represents a button for which the label text is set.
- An `ASConstText` object that specifies the button's label text (the following code example demonstrates how to create this object).
- An `AVToolButtonLabelPriority` value that specifies a set of priority values for a button's label text. This priority value determines the preference order in which labels are shown when a toolbar is too short to hold all of the button labels. The following values are valid:
  - `kAVButtonPriorityOffExtraLow`
  - `kAVButtonPriorityOffLow`
  - `kAVButtonPriorityOffNormal`
  - `kAVButtonPriorityOffHigh`
  - `kAVButtonPriorityOffExtraHigh`
  - `kAVButtonPriorityOnExtraLow`
  - `kAVButtonPriorityOnLow`
  - `kAVButtonPriorityOnNormal`
  - `kAVButtonPriorityOnHigh`
  - `kAVButtonPriorityOnExtraHigh`
  - `kAVButtonPriorityAlwaysOn`

The following code example sets a button's label text.

Example: *Setting a button's label text*

```
//Declare an AVToolButton object
AVToolButton MyButton = NULL;

//Create a AVIcon object
AVIcon myIcon = (AVCursor)LoadBitmap(gHINSTANCE,
MAKEINTRESOURCE(IDB_BITMAP1));

//Create a new button
MyButton = AVToolButtonNew (ASAtomFromString("MyExtn:MyButton"), myIcon,
FALSE, FALSE);

//Create an ASConstText object by using a ASText object
ASText tmpText = ASTextNew();
ASTextSetPDText(tmpText, "View PDF");
ASConstText labelText = tmpText;

//Set the button's label text with a kAVButtonPriorityOnNormal priority
AVToolButtonSetLabelText (MyButton, labelText, kAVButtonPriorityOnNormal);
```

Creating a sub-menu for a button

You can create a sub-menu that appears when a user clicks the button. A sub-menu contains menu comments that a user can select to invoke a specific action. To create a sub-menu for a button, invoke the `AVToolButtonSetMenu` method and pass the following arguments:

- An `AVToolButton` object that specifies a button to which the menu is attached.
- An `AVMenu` object that represents the menu. (See "Creating Menus and Menu Commands" on page 82.)

Tip: To view an example of a sub-menu, click the Help button that appears on the Help toolbar.

Retrieving existing toolbar buttons

Instead of creating a new button, you can retrieve an existing button. You can, for example, retrieve a button from one toolbar and attach it to another toolbar. (See "Attaching a button to a toolbar" on page 96.)

To retrieve an existing toolbar button, invoke the `AVToolBarGetButtonByName` method and pass the following arguments:

- An `AVToolBar` object that represents the toolbar from which the button is retrieved.
- An `ASAtom` object that represents the button name. For information about button names, see the Acrobat and PDF Library API Reference.

The `AVToolBarGetButtonByName` method returns an `AVToolButton` object that corresponds to the specified button. If the name is not found, then this method returns `NULL`. Once you obtain a button, you can perform various tasks such as attaching it to another toolbar or retrieving its icon by invoking the `AVToolButtonGetIcon` method and passing the `AVToolButton` object that contains the icon.

The following code example retrieves the SecureTask button located on the Tasks toolbar and gets its icon.

Example: *Retrieving existing toolbar buttons*

```
//Retrieve the Tasks toolbar
const char * toolbarName= "Tasks" ;
AVToolBar myToolBar = AVAppGetToolBarByName(toolbarName);

//Retrieve the SecureTask button located on the Tasks toolbar
AVToolButton mySecureButton =
AVToolBarGetButtonByName(myToolBar,ASAtomFromString("SecureTask"));

if (mySecureButton == NULL)
{
   AVAlertNote ("The button was not successfully retrieved");
   return;
}

//Get the icon located on the button
//Pass the AVToolButton object
AVIcon mySecureIcon = AVToolButtonGetIcon(mySecureButton);
```

Attaching a button to a toolbar

After you create a new button, you must attach it to a toolbar. A button must be attached to a toolbar before it is visible within Adobe Reader or Acrobat. To attach a button to a toolbar, invoke the `AVToolBarAddButton` method and pass the following arguments:

- An `AVToolBar` object that represents the toolbar to which the button is attached.
- An `AVToolButton` object that represents the button that is attached.
- An `ASBool` object that specifies the location of where the button is attached. If `true`, the button is attached before the button specified by the `otherButton` argument. If `false`, the button is attached after the button specified by the `otherButton` argument. If `otherButton` is NULL and this value is `true`, the button is attached at the beginning of the toolbar. If `otherButton` is NULL and this value is `false`, the button is attached at the end of the toolbar.
- An `AVToolButton` object (the name of this argument is `otherButton`) that is used in conjunction with the `ASBool` object that specifies the location of where the `AVToolButton` object is attached.

Before a button has functionality, you must create a callback function. (See "Creating toolbar button callback functions" on page 98.)

Acrobat 9 adds the `AVToolBarAddButtonEx` method for creating a new button. This method takes a structure that lets you specify where you want the button to appear and whether the button should be hidden by default.

The following code example attaches a newly created button to the File toolbar.

Example: *Attaching a button to a toolbar*

```
//Declare an AVToolButton object
AVToolButton MyButton = NULL;

//Create a AVIcon object
AVIcon myIcon = (AVCursor)LoadBitmap(gHINSTANCE,
MAKEINTRESOURCE(IDB_BITMAP1));

//Create a new button
MyButton = AVToolButtonNew (ASAtomFromString("MyExtn:MyButton"), myIcon,
FALSE, FALSE);

//Retrieve the File toolbar
const char * toolbarName= "File";
AVToolBar ToolBar = AVAppGetToolBarByName(toolbarName);

//Attach the button
AVToolBarAddButton(ToolBar, MyButton, FALSE, NULL);
```

Note: For information about creating a button, see "Creating toolbar buttons" on page 92.

Exposing a button in a web browser

You can expose an Acrobat or Adobe Reader toolbar button within a web browser by invoking the `AVToolButtonSetExternal` method. Pass the following arguments to the `AVToolButtonSetExternal` method:

- An `AVToolButton` object that represents the button to expose within a web browser.
- Both the `TOOLBUTTON_EXTERNAL` and `TOOLBUTTON_INTERNAL` values to ensure that the button is visible within Acrobat or Adobe Reader and a web browser.

The following code example exposes a button in a web browser.

Example: *Exposing a button in a web browser*

```
//Declare an AVToolButton object
AVToolButton MyButton = NULL;

//Create a AVIcon object
AVIcon myIcon = (AVCursor)LoadBitmap(gHINSTANCE,
MAKEINTRESOURCE(IDB_BITMAP1));

//Create a new button
MyButton = AVToolButtonNew (ASAtomFromString("MyExtn:MyButton"), myIcon,
FALSE, FALSE);

//Retrieve the File toolbar
const char * toolbarName= "File";
AVToolBar ToolBar = AVAppGetToolBarByName(toolbarName);

//Expose the button in a web browser
AVToolButtonSetExternal(MyButton, TOOLBUTTON_EXTERNAL | TOOLBUTTON_INTERNAL);

//Attach the button
AVToolBarAddButton(ToolBar, MyButton, FALSE, NULL);
```

Removing a button from a toolbar

You can use the Acrobat core API to remove a button from a toolbar. To remove a button from a toolbar, invoke the `AVToolButtonRemove` method and pass a `AVToolButton` object that represents the button to remove. Although the button is removed from the toolbar, it is not destroyed. At any time, you can attach the button to the same or different toolbar. (See "Attaching a button to a toolbar" on page 96.)

After you remove the button, invoke the `AVToolBarUpdateButtonStates` method to update the toolbar. This method requires an `AVToolBar` object that represents the toolbar to update. The following code example removes the SecureTask button located on the Tasks toolbar.

Example: *Removing a button from a toolbar*

```
//Retrieve the Tasks toolbar
const char * toolbarName= "Tasks" ;
AVToolBar myToolBar = AVAppGetToolBarByName(toolbarName);

//Retrieve the SecureTask button located on the Tasks toolbar
AVToolButton mySecureButton =
AVToolBarGetButtonByName(myToolBar,ASAtomFromString("SecureTask"));

if (mySecureButton == NULL)
{
   AVAlertNote ("The button was not successfully retrieved");
   return;
```

```
}

//Remove the SecureTask button from the Tasks toolbar
AVToolButtonRemove(mySecureButton);

//Update the toolbar
AVToolBarUpdateButtonStates(myToolBar);
```

Note: You can invoke the `AVToolButtonDestroy` method to destroy a button.

Creating toolbar button callback functions

You can create a toolbar button callback function which is invoked by Adobe Reader or Acrobat when a user clicks a button. For the purposes of this discussion, a simplistic user-defined function named `ShowButtonMessage` is introduced. This method displays a message box by invoking the `AVAlertNote` method. The following code shows the body of the `ShowButtonMessage` function.

```
ACCB1 void ACCB2 ShowButtonMessage (void* data)
{
   AVAlertNote ("A button was clicked.");
}
```

The data parameter for this and the other callbacks can be used to maintain private data that is used by the callback. Notice that this user-defined function is declared using the `ACCB1` and `ACCB2` macros. (See "Using callback functions" on page 29.)

To create a callback for a button, create an `AVExecuteProc` object:

```
AVExecuteProc ExecProcPtr = NULL;
```

`AVExecuteProc` is a callback that you can create that is invoked by Acrobat or Adobe Reader when a user clicks a button. After you create an `AVExecuteProc` object, you can invoke the `ASCallbackCreateProto` macro that is defined in the Acrobat core API to convert a user-defined function to an Acrobat callback. For example, you can invoke `ASCallbackCreateProto` to convert `ShowButtonMessage` to a callback function. The `ASCallbackCreateProto` macro requires the following arguments:

- The callback type. For example, you can pass `AVExecuteProc`.
- The address of the user-defined function to convert to a callback.

The `ASCallbackCreateProto` macro returns a callback of the specified type that invokes the user-defined function whose address was passed as the second argument. The following lines of code shows the `ASCallbackCreateProto` macro converting the `ShowButtonMessage` user-defined function to a `AVExecuteProc` callback.

```
AVExecuteProc ExecProcPtr = NULL;
ExecProcPtr= ASCallbackCreateProto(AVExecuteProc, &ShowButtonMessage);
```

After you create an `AVExecuteProc` callback, you can invoke the `AVToolButtonSetExecuteProc` method to associate a button with a callback. That is, when a user clicks a button, Acrobat or Adobe Reader will invoke the user-defined function whose address was passed to the `ASCallbackCreateProto` macro. The `AVToolButtonSetExecuteProc` method requires the following parameters:

- An `AVToolButton` object that represents the button to associate with the callback
- An `AVExecuteProc` object that represents the callback function

- The address of a user-defined data structure that can be passed to the user-defined function

When you are done with a button callback, invoke the `ASCallbackDestroy` method to release the memory that it consumes.

The following code example creates a callback function for a button.

Example: *Creating a toolbar button callback function*

```
//Define a toolbar button callback function
ACCB1 void ACCB2 ShowButtonMessage (void* data)
{
   AVAlertNote ("A button was clicked.");
}

ACCB1 ASBool ACCB2 PluginInit (void)
{

//Declare an AVToolButton object
AVToolButton MyButton = NULL;

//Create a AVIcon object
AVIcon myIcon = (AVCursor)LoadBitmap(gHINSTANCE,
MAKEINTRESOURCE(IDB_BITMAP1));

//Create a new button
MyButton = AVToolButtonNew (ASAtomFromString("MyExtn:MyButton"), myIcon,
FALSE, FALSE);

//Retrieve the File toolbar
const char * toolbarName= "File" ;
AVToolBar ToolBar = AVAppGetToolBarByName(toolbarName);

//Create toolbar button callback
AVExecuteProc ExecProcPtr = ASCallbackCreateProto (AVExecuteProc,
&ShowButtonMessage);
AVToolButtonSetExecuteProc (MyButton, ExecProcPtr, NULL);

//Attach the button
AVToolBarAddButton(ToolBar, MyButton, FALSE, NULL);

//Release the callback function
ASCallbackDestroy(ExecProcPtr);

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

Note: Notice that the application logic that creates a toolbar button is located in the `PluginInit` procedure. (See "About plugin initialization" on page 26.)

8 Creating Annotations

This chapter explains how to create new annotations and modify existing ones. An annotation associates an object such as a note, sound, or movie with a location on a page of a PDF document, or provides a way to interact with the user by means of the mouse and keyboard.

About annotations

The Acrobat core API provides methods for working with annotations in PDF documents. Annotations are represented by a `PDAnnot` typedef, which is the abstract superclass for all annotations.

Several annotation types exist, which are identified by their subtype. Each subtype can have additional properties that extend the basic ones. The subtype for text annotations (also called notes) is text. The subtype for link annotations is link. The Acrobat core API contains two built-in annotation typedefs `PDTextAnnot` and `PDLinkAnnot`. A `PDTextAnnot` object corresponds to a text annotation and a `PDLinkAnnot` object corresponds to a link annotation.

You can use `PDAnnot` methods to get and set various annotation properties, such as color, date, title, location, and subtype. For example, you can invoke the `PDAnnotSetColor` method to set the color of an annotation.

Note: This chapter does not discuss how to create 3D annotations. (See "Creating 3D Annotations" on page 194.)

Working with text annotations

The Acrobat API lets you create text annotations and retrieve and modify attributes of an existing text annotation. Acrobat displays text annotations as sticky notes.

Creating text annotations

You can create a text annotation by performing the following tasks:

1. Create a rectangle region that specifies the annotation's location. To create a rectangle region, create an `ASFixedRect` object.
2. Define the rectangle's borders by setting the `ASFixedRect` object's left, top, right, and bottom attributes.
3. Create a `PDPage` object that represents the page that will contain the new annotation by invoking the `PDDocAcquirePage` method. The first argument passed to this method is a `PDDoc` object that represents the PDF document and the second is an `ASInt32` object that represents the page number on which the annotation is applied. This method returns a `PDPage` object.
4. Create a `PDAnnot` object by invoking the `PDPageCreateAnnot` method and passing the following arguments:
   - A `PDPage` object that represents the page that will contain the new annotation.

- An `ASAtom` object that represents the annotation's subtype. Because a text annotation is created, specify `Text` as the annotation's subtype. (See the Acrobat and PDF Library API Reference.)
- The address of the `ASFixedRect` object.

5. Cast the `PDAnnot` object to a `PDTextAnnot` object by invoking the `CastToPDTextAnnot` method. This method requires a `PDAnnot` object and returns a `PDTextAnnot` object.

6. Open the text annotation by invoking the `PDTextAnnotSetOpen` method. Opening an annotation enables you to set its content. This method requires a `PDTextAnnot` object and an `ASBool` value that specifies `true`.

7. Set the text of the annotation by invoking the `PDTextAnnotSetContents` method and passing the following arguments:
   - A `PDTextAnnot` object that represents the annotation for which text is set.
   - A character pointer that specifies the text to set.
   - An `ASInt32` object that specifies the length of the character pointer.

8. Add the text annotation to the page by invoking the `PDPageAddAnnot` method and passing the following arguments:
   - A `PDPage` object that represents the page that will contain the new annotation.
   - An `ASInt32` object that specifies the index that controls where the annotation is added. The first annotation in the array has an index of zero. Passing a value of -2 adds the annotation to the end of the array.
   - A `PDTextAnnot` object that represents the annotation.

The following code example adds a text annotation to a PDF document page. In this code example, a `PDDoc` object named `myPDDoc` exists. (See "Creating a PDDoc object" on page 76.)

Example: *Creating text annotations*

```
PDPage page = NULL;
PDAnnot annot,textannot;
char* ptr = "This is initial text";

//Create an ASFixed object and define its borders
ASFixedRect fr;
fr.left = ASInt32ToFixed(36);
fr.top = ASInt32ToFixed(792-36);
fr.right = ASInt32ToFixed(136);
fr.bottom = ASInt32ToFixed(792-136);

//Create a PDPage object
page = PDDocAcquirePage(myPDDoc, 0);

//Create a PDAnnot object
annot = PDPageCreateAnnot (page, ASAtomFromString("Text"),&fr);

//Cast the PDAnnot object to a PDTextAnnot object
textannot = CastToPDTextAnnot(annot);

//Open the annotation, set the text, and add it to a page
PDTextAnnotSetOpen (textannot, true);
```

```
PDTextAnnotSetContents (textannot, ptr, strlen (ptr));
PDPageAddAnnot(page,-2, textannot);
```

Retrieving existing annotations

You can use the Acrobat core API to retrieve existing annotations located within a PDF document by performing the following tasks:

1. Create a `PDDoc` object that represents the PDF document that contains annotations. (See "Creating a PDDoc object" on page 76.)
2. Search for existing annotations by iterating through the PDF document page by page. One way to perform this task is to create a for loop structure and invoke the `PDDocGetNumPages` method. This method requires a `PDDoc` object as an argument and returns the number of pages within the document.
3. For each page within the PDF document, obtain a `PDPage` object by invoking the `PDDocAcquirePage` method and passing the following arguments:
   - A `PDDoc` object that represents the PDF document that contains the page.
   - An `ASInt32` object that represents the page number.
4. After you obtain a `PDPage` object, get the number of annotations located on the page by invoking the `PDPageGetNumAnnots` method. This method requires a `PDPage` object as an argument and returns an `ASInt32` object representing the number of annotations located on the page.
5. For each annotation on a page, invoke the `PDPageGetAnnot` method. This method requires a `PDPage` object and an `ASInt32` object that represents the index of the annotation. This method returns a `PDAnnot` object.

The following code example retrieves existing annotations located within a PDF document. After an annotation is retrieved, information about the annotation is displayed within an alert box. Information about the annotation is retrieved by invoking the `PDAnnotGetSubtype` method. This method returns an `ASAtom` object representing the annotation's subtype. For example, if the annotation is a stamp, then an `ASAtom` object storing the value `Stamp` is returned. You can get the string value from an `ASAtom` object by invoking the `ASAtomGetString` method and passing the `ASAtom` object.

Example: *Retrieving existing annotations*

```
PDPage page;
ASInt32 i,i2;
PDAnnot annot;
char* ptr;
char buf[200];
ASAtom atom;

//Iterate through the PDF document page by page
for (i = 0; i < PDDocGetNumPages(myPDDoc); i ++){
   //Get each page within the document
   page = PDDocAcquirePage(myPDDoc, i);

   //Get each annotation on the page
   for (i2 = 0; i2 < PDPageGetNumAnnots(page); i2++) {
      //Get a specific annotation
      annot = PDPageGetAnnot(page,i2);
```

```
      if (PDAnnotIsValid(annot)){

         //Display subtype information about the annotation
         atom = PDAnnotGetSubtype(annot);

         //Cast the ASAtom object to a character pointer
         ptr = (char*) ASAtomGetString(atom);
         sprintf(buf, "The annotation's subtype is %s", ptr);
         AVAlertNote (buf);
      }
   }
}
```

Note: In the previous code example, assume a `PDDoc` object named `myPDDoc` exists. (See "Creating a PDDoc object" on page 76.)

Modifying text annotations

You can modify an annotation after you retrieve it. For example, you can retrieve an existing text annotation and modify its text. For information about retrieving an annotation, see "Retrieving existing annotations" on page 102.

Before you modify an annotation, determine whether the annotation is the correct subtype. That is, before modifying a text annotation, ensure that the annotation is a `Text` annotation. You can determine whether an annotation is the correct subtype by invoking the `PDAnnotGetSubtype` method. This method requires a `PDAnnot` object and returns an `ASAtom` object that specifies the annotation's subtype.

When modifying a text annotation, it is recommended that you check its contents. For example, you can retrieve all text annotations in a PDF document, retrieve the annotation's text, and modify annotations that contain specific text. To retrieve the text of an annotation, invoke the `PDTextAnnotGetContents` method and pass the following arguments:

- A `PDTextAnnot` object that contains text to retrieve.
- A character pointer that is populated with the annotation's text.
- An `ASInt32` object that represents the size of the character pointer.

The following code example iterates through all annotations located in a PDF document. Each valid annotation is checked to determine whether it is a `Text` annotation. This task is performed by invoking the `PDAnnotGetSubtype` method. If the annotations is a `Text` annotation, then the annotation's text is retrieved by invoking the `PDTextAnnotGetContents` method.

Because the size of the annotation's text is unknown, the `PDTextAnnotGetContents` is invoked twice. The first time it is invoked, `NULL` is passed as the buffer address (second argument) and `0` is specified as the buffer size (third argument). The text length is returned to an `ASInt32` object named `bufSize`. The `ASmalloc` method is invoked which allocates `bufSize` bytes to the character pointer.

The second time `PDTextAnnotGetContents` is invoked, the allocated character pointer is passed as well as the `ASInt32` object named `bufSize`. The character pointer is populated with the annotation's text.

Next the `strcmp` method is invoked to compare the annotation's text with a specific string value. If the annotation's text matches the string value, then the `PDTextAnnotSetContents` method is invoked, which replaces the annotation's text with new text.

Example: *Modifying a text annotation*

```
PDPage page;
ASInt32 i,i2;
PDAnnot annot;
char setbuf[200];
ASAtom atom;

//Iterate through the PDF document page by page
for (i = 0; i < PDDocGetNumPages(myPDDoc); i ++){
   //Get each page within the document
   page = PDDocAcquirePage(myPDDoc, i);

   //Get each annotation on the page
   for (i2 = 0; i2 < PDPageGetNumAnnots(page); i2++) {
      //Get a specific annotation
      annot = PDPageGetAnnot(page,i2);
      if (PDAnnotIsValid(annot)){
         //Determine if the annotation is a Text annotation
         if (PDAnnotGetSubtype(annot) == ASAtomFromString("Text")) {
            //Create a character pointer to store the annotation's text
            char * annBuf;
            ASInt32 bufSize = PDTextAnnotGetContents(annot, NULL, 0) +1;

            //Allocate the size of bufSize to the character pointer
            annBuf = (char*)ASmalloc((os_size_t)bufSize);

            //Populate annBuf with the annotation's text
            PDTextAnnotGetContents(annot, annBuf, bufSize);

            //Compare the contents of annBuf with a string
            if (strcmp(annBuf,"This is initial text") == 0){
               //Modify the annotation's text
               sprintf (setbuf, "This is the new text for the annotation.");
               PDTextAnnotSetOpen (annot, true);
               PDTextAnnotSetContents (annot, setbuf, strlen(setbuf));
            }
      }
   }
}
```

Note: In the previous code example, assume a `PDDoc` object named `myPDDoc` exists. (See "Creating a PDDoc object" on page 76.)

Working with redaction annotations

The Acrobat API lets you create redaction annotations and access and modify the attributes in an existing redaction annotation. It also lets you apply an existing redaction annotation, which permanently removes the redacted material from the PDF document.

A redaction annotation identifies content to be removed from the document. The intent of redaction annotations is to enable the following process:

1. Create redaction annotations that identify the content to be removed from the document. The redaction annotation specifies a rectangle that covers the content to be removed and specifies the appearance of the rectangle and associated information.

2. Apply redaction annotations, which remove the content in the area specified by a set of redaction annotations. In the removed content's place, some marking appears to indicate that the area was redacted. Also, the redaction annotations are removed from the PDF document.

   A single content removal operation can remove the content specified by multiple redaction annotations.

Creating a redaction annotation

To create a redaction annotation that identifies the content to be removed from the document and the appearance of the redaction annotation, perform the following tasks:

1. Create a `PDRedactParamsRec` structure and populate it with values that describe characteristics of the redaction annotation, such as the page number on which the redaction is to be applied and the regions on the page being redacted.

2. Apply the redaction annotation to the document, by invoking the `PDDocCreateRedaction` method.

The `PDDocCreateRedaction` function automatically sets common annotation properties, such as the annotation rectangle (`Rect`) and the annotations unique name (`NM`). Further changes to the annotation are unnecessary. However, if you want to set values for annotation properties not set by the `PDDocCreateRedaction` method, convert the `PDAnnot` object returned from that method into a Cos object and set its dictionary values. (See Working with Cos Objects and Creating the attributes dictionary.)

Modifying an existing redaction annotation

To modify the attributes of an existing redaction annotation, perform the following tasks:

1. Obtain a pointer to the redaction annotation. (See Retrieving existing annotations.) Ensure that the pointer references an annotation with the `Subtype` value `Redact`.

2. Get the redaction annotation's properties, by invoking the `PDRedactionGetProps` function. The first argument is the pointer to the redaction annotation obtained in Step 1 and the second argument is a pointer to the `PDRedactParamsRec` structure.

3. Modify the members of the `PDRedactParamsRec` structure supplied by the `PDRedactionGetProps` function. The first argument is the pointer to the redaction annotation obtained in Step 1, and the second argument is a pointer to the `PDRedactParamsRec` structure.

4. Set the redaction annotation's properties by invoking the `PDRedactionSetProps` function.

Applying redaction annotations (removing redacted content)

To apply previously created redaction annotations, perform the following tasks:

1. Create a `PDApplyRedactionParamsRec` structure that specifies the redaction annotations to apply. The structure also lets you provide callback functions that Acrobat invokes as it applies the redaction annotations.

2. Apply the redaction annotations by invoking the `PDDocApplyRedactions` function. This function name is overloaded with two forms. The simpler form specifies a first argument that is the pointer to the document, and the second argument that is the structure created in Step 1. The more complex forms let you specify progress callbacks.

9 Working with Bookmarks

You can use the core API to create new bookmarks and search for existing ones. A bookmark is a link with representative text on the Bookmarks tab in the navigation pane. Each bookmark navigates to a different view or page within a PDF document. You can also use a bookmark to navigate to a specific destination within a PDF document, to another document (PDF or other), or to a web page. Bookmarks can also perform actions, such as executing a menu item or displaying a graphic file.

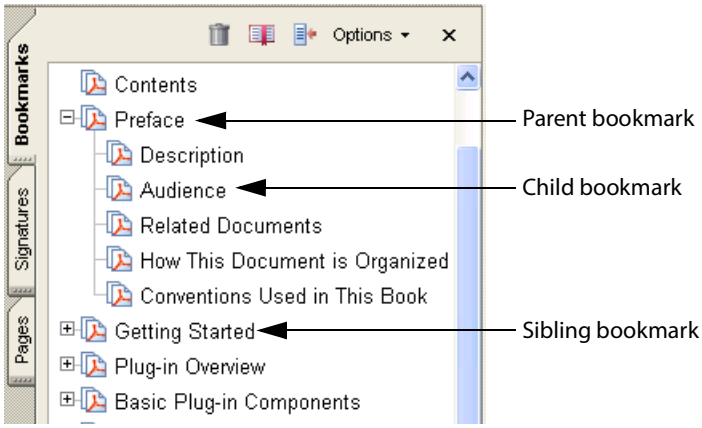
About bookmarks

Bookmarks are represented by a `PDBookmark` object. All bookmarks have the following attributes:

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

Every document has a root bookmark. The root bookmark does not represent a physical bookmark that appears in Adobe Reader or Acrobat, but is the root from which all bookmarks in the tree are descended. Bookmarks are organized in a tree structure in which each bookmark has zero or more children that appear indented, 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 a 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.

The following image shows how bookmarks appear in Adobe Reader or Acrobat.



The Acrobat core API contains methods that operate on bookmarks. Using these methods, you can perform the following tasks:

- Create new bookmarks
- Get and set various attributes of a bookmark (such as its title or action or whether it is open)
- Search for a bookmark

Creating bookmarks

Before you can create a bookmark, you must create a `PDDoc` object that represents the PDF document to which the bookmark is added. (See "Creating a PDDoc object" on page 76.)

To create bookmarks for a PDF document, perform the following tasks:

1. Get the root of the PDF document's bookmark tree by invoking the `PDDocGetBookmarkRoot` method. This method requires a `PDDoc` object and returns a `PDBookmark` object that represents the document's root bookmark. The document's root bookmark does not appear in Adobe Reader or Acrobat.

2. Create another `PDBookmark` object that represents the bookmark to add to the document's root bookmark by invoking the `PDBookmarkAddNewChild` method. This method requires a `PDBookmark` object that represents the parent bookmark (in this case the parent bookmark is also the document's root bookmark) and an `ASAtom` object that represents the bookmark's title.

3. Create a `PDBookmark` object that represents a sibling bookmark to the bookmark that was added to the document's root bookmark (the sibling bookmark is also a child of the document's root bookmark). You can perform this task by invoking the `PDBookmarkAddNewSibling` method. This method requires a `PDBookmark` object that represents the new bookmark's sibling bookmark and an `ASAtom` object that represents the bookmark's title.

The following code example adds two new bookmarks to a PDF document. After each bookmark is created, the `PDBookmarkIsValid` method is invoked to determine whether the bookmark is valid. The name of the `PDDoc` object used in this code example is `myPDDoc`. (See "Creating a PDDoc object" on page 76.)

Example: *Creating bookmarks*

```
//Declare a bookmark object
PDBookmark rootBookmark;
PDBookmark childBookmark ;
PDBookmark siblingBookmark;

//Get the root bookmark
rootBookmark = PDDocGetBookmarkRoot(myPDDoc);

if (PDBookmarkIsValid(rootBookmark)){

   //Add a child bookmark to the root bookmark
   childBookmark = PDBookmarkAddNewChild(rootBookmark, "Bookmark1");

   if (PDBookmarkIsValid(childBookmark)){

      //Add a sibling bookmark to the child bookmark
      siblingBookmark = PDBookmarkAddNewSibling(childBookmark, "Bookmark2");
   }
}
```

Defining bookmark actions

After you create a new bookmark, you must define an action that occurs when a user clicks on the bookmark. Otherwise, nothing occurs when a user clicks on the bookmark.

To create an action for a bookmark, you must create a `PDAction` object that represents the action that occurs when a user clicks on a bookmark. Once you create a `PDAction` object, you can assign it to a bookmark. (See "Assigning an action to a bookmark" on page 110.)

As specified earlier in this chapter, a typical bookmark action is to move to another location in the current document. To illustrate how to create a bookmark action, this section defines a bookmark action that displays a specific page in a PDF document when a user clicks the bookmark.

To define a bookmark action that generates a specific view of a PDF document, you create a `PDAction` object by invoking the `PDActionNewFromDest` method. This method creates a new action that directs the user to the specified destination view and requires the following arguments:

- A `PDDoc` that represents the PDF document for which the action is created.
- A `PDViewDestination` object that represents a specific view in the PDF document. (See "Creating a PDViewDestination object" on page 109.)
- A `PDDoc` that represents the destination document. This object is the same object that is specified as the first parameter.

The `PDActionNewFromDest` method returns a `PDAction` method.

Creating a PDViewDestination object

You must create a `PDViewDestination` object in order to invoke the `PDActionNewFromDest` method. To create a `PDViewDestination` object, invoke the `AVPageViewToViewDest` method and pass the following arguments:

- An `AVPageView` object that represents the page view from which the destination is created. For information about creating this object, see "Creating a AVPageView object" on page 110.
- An `ASAtom` object that specifies the fit type of the view destination (see the table that follows this list).
- A `PDDoc` object that represents the PDF document for which the view is established.

The `AVPageViewToViewDest` method returns a `PDViewDestination` object. The following table specifies the fit type values that you pass to the `AVPageViewToViewDest` method as the second argument.

Value	Description
`XYZ`	Destination specified as upper-left corner point and a zoom factor.
`Fit`	Fits the page into the window, corresponding to the Acrobat viewer's Fit Page menu item.
`FitH`	Fits the width of the page into the window, corresponding to the Acrobat viewer's Fit Width menu item.
`FitV`	Fits the height of the page into a window.
`FitR`	Fits the rectangle specified by its upper-left and lower-right corner points into the window.
`FitB`	Fits the rectangle containing all visible elements on the page (known as the bounding box) into the window (corresponds to the Acrobat viewer's Fit Visible menu item).
`FitBH`	Fits the width of the bounding box into the window.
`FitBV`	Fits the height of the bounding box into the window.

Creating a AVPageView object

You must create an `AVPageView` object that represents the page view in order to invoke the `AVPageViewToViewDest` method. You can create an `AVPageView` object by invoking the `AVDocGetPageView` method. This method requires an `AVDoc` that represents a PDF document whose page view is obtained and returns an `AVPageView` object.

You can create an `AVDoc` object that is based on the `PDDoc` object that already exists by invoking the `AVDocFromPDDoc` method and passing the `PDDoc` object. You can invoke this method as an argument for the `AVDocGetPageView` method.

```
AVPageView myPageView = AVDocGetPageView(AVDocFromPDDoc(myPDDoc));
```

Once you create an `AVPageView` object, you can specify a specific PDF document page number by invoking the `AVPageViewGoTo` method and passing the `AVPageView` object and an `ASInt32` object that represents the page number:

```
ASInt32 pNum = 2;
AVPageViewGoTo(myPageView,pNum);
```

Assigning an action to a bookmark

After you create both an `AVPageView` object and an `PDViewDestination` object, you can create a `PDAction` object and assign it to a specific bookmark by invoking the `PDBookmarkSetAction` method and passing the `PDBookmark` object and the `PDAction` object as arguments.

The following code example creates a `PDAction` object and assigns it to a bookmark that is represented by a `PDBookmark` object named `childBookmark`.

Example: *Assigning an action to a bookmark*

```
//Create a PDDoc object based on the current PDF document
AVDoc avDoc = AVAppGetActiveDoc();
AVPageView pageView = AVDocGetPageView(avDoc);
PDPageNumber pageNum = AVPageViewGetPageNum(pageView);
PDDoc myPDDoc = AVDocGetPDDoc(avDoc);

//Create a AVPageView object that represents the page view of a document
AVPageView myPageView = AVDocGetPageView(AVDocFromPDDoc(myPDDoc));

//Set the page view to the second page
ASInt32 pNum = 2;
AVPageViewGoTo(myPageView,pNum);

//Create an PDViewDestination object that is used to create a PDAction object
PDViewDestination pdvDes =
AVPageViewToViewDest(myPageView,ASAtomFromString("Fit"),myPDDoc);

//Create a PDAction object
PDAction myAction = PDActionNewFromDest(myPDDoc,pdvDes,myPDDoc);

//Attach an action to the bookmark
PDBookmarkSetAction(childBookmark,myAction);
```

Caution: When running this code example, you must have the PDF document on which the `PDDoc` object is based open. Otherwise, a run-time error occurs. Also, you must create a `PDBookmark` object named `childBookmark`. (See "Creating bookmarks" on page 108.)

Removing bookmark actions

You can remove an action from a bookmark by invoking the `PDBookmarkRemoveAction` method. After you remove a bookmark, you can add a new action. The `PDBookmarkRemoveAction` method requires a `PDBookmark` object that represents the bookmark from which the action is removed.

Opening and closing bookmarks

You can programmatically open and close a bookmark. To open and close a bookmark, invoke the `PDBookmarkSetOpen` method and pass the following arguments:

- A `PDBookmark` object to open or close.
- An `ASBool` value that specifies whether to open or close the bookmark. The value `true` specifies to open the bookmark and the value `false` specifies to close the bookmark.

Before you invoke the `PDBookmarkSetOpen` method, it is recommended that you invoke the `PDBookmarkIsOpen` method to determine whether the bookmark is open. This method requires a `PDBookmark` object and returns an `ASBool` value. If the bookmark is open, then `true` is returned.

The following code example retrieves and opens a bookmark whose title is *Samples*. For information about retrieving a specific bookmark, see "Retrieving a specific bookmark" on page 112.

Example: *Opening a bookmark*

```
//Retrieve a bookmark whose title is Samples
PDBookmark rootBookmark, myBookmark;
char* bookmarkTitle = "Samples";

//Get the root bookmark
rootBookmark = PDDocGetBookmarkRoot(myPDDoc);

//Get the bookmark whose title is Samples
myBookmark = PDBookmarkGetByTitle (rootBookmark, bookmarkTitle,
strlen(bookmarkTitle), -1);
if (PDBookmarkIsValid (myBookmark)){

   //Determine whether the bookmark is open
   if (!PDBookmarkIsOpen(myBookmark)){
      //Open the bookmark
      PDBookmarkSetOpen(myBookmark,true);
      AVAlertNote("The bookmark was opened");
   }
}
else
   AVAlertNote("The bookmark was not retrieved");
```

Retrieving bookmarks

You can retrieve the root bookmark, retrieve a specific bookmark, or retrieve all bookmarks that are located within a PDF document.

Retrieving the root bookmark

Every PDF 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 example shows how to get a PDF document's root bookmark by creating application logic within a user-defined function named `GetFirstBookmark`. First, the `PDDocGetBookmarkRoot` method is invoked to get the bookmark root. This method requires a `PDDoc` object that represents the PDF document from which the root bookmark is retrieved and returns a `PDBookmark` object that represents the root bookmark. (See "Creating a PDDoc object" on page 76.)

Next, the `PDBookmarkGetFirstChild` method is invoked to get the first child of the root. If there are no bookmarks, `PDBookmarkGetFirstChild` returns `NULL`.

Example: *Retrieving the root bookmark*

```
PDBookmark GetFirstBookmark(PDDoc doc)
{
   PDBookmark theroot, childBookmark;
   theroot = PDDocGetBookmarkRoot(doc);
   childBookmark = PDBookmarkGetFirstChild(theroot);
   return childBookmark;
}
```

Retrieving a specific bookmark

You can retrieve a specific bookmark by specifying its title. The following code example retrieves a specific bookmark by invoking the `PDDocGetBookmarkRoot` method to get the document's root bookmark as a starting point for the search. It then invokes the `PDBookmarkGetByTitle` method to retrieve the first bookmark whose title matches the specified title. This method requires the following arguments:

- The root of the bookmark tree that is searched.
- A character pointer that specifies the title of the bookmark.
- An `ASInt32` object that specifies the length of the character pointer.
- An `ASInt32` object that specifies the number of bookmark levels to search. The value `-1` specifies to search the entire sub-tree. The value `1` specifies to search only child bookmarks of the current bookmark. The value `0` specifies to look at the current bookmark.

Example: *Retrieving a specific bookmark*

```
//Retrieve a bookmark whose title is Samples
PDBookmark rootBookmark, myBookmark;
char* bookmarkTitle = "Samples";

//Get the root bookmark
rootBookmark = PDDocGetBookmarkRoot(myPDDoc);

//Retrieve a specific bookmark
```

```
myBookmark = PDBookmarkGetByTitle (rootBookmark, bookmarkTitle,
strlen(bookmarkTitle), -1);
if (PDBookmarkIsValid (myBookmark))
   AVAlertNote("The bookmark was retrieved");
else
   AVAlertNote("The bookmark was not retrieved");
```

Note: In the previous code example, a `PDDoc` object named `myPDDoc` is passed to the `PDDocGetBookmarkRoot` method. For information about creating this object, see "Creating a PDDoc object" on page 76.

Retrieving all bookmarks

You can use the Acrobat core API to retrieve all bookmarks located within a PDF document. For example, you can retrieve the title of every bookmark that is located within a PDF document.

The following code example creates a recursive user-defined function named `VisitAllBookmarks`. First it invokes the `PDBookmarkIsValid` method to ensure that the bookmark that is passed is valid (the root bookmark is always valid.)

Second, this user-defined function retrieves the title of the bookmark by invoking the `PDBookmarkGetTitle` method. This method requires the following arguments:

- A `PDBookmark` object that contains the title to retrieve.
- A character pointer that is populated with the bookmarks title.
- An `ASInt32` object that represents the size of the character pointer.

Because the size of the bookmark's title is unknown, the `PDBookmarkGetTitle` is invoked twice. The first time it is invoked, `NULL` is passed as the buffer address (second argument) and `0` is specified as the buffer size (third argument). The text length is returned to an `ASInt32` object named `bufSize`. The `ASmalloc` method is invoked which allocates `bufSize` bytes to the character pointer.

The second time `PDBookmarkGetTitle` is invoked, the allocated character pointer is passed as well as the `ASInt32` object named `bufSize`. The character pointer is populated with the bookmark's title. The `AVAlertNote` method is invoked and the character pointer is passed as an argument that results in the bookmark's title being displayed within a message box.

The `PDBookmarkHasChildren` method is invoked to determine whether there are any child bookmarks under the current bookmark. If there are child bookmarks, the `PDBookmarkGetFirstChild` method is invoked to retrieve the first child bookmark. A recursive call is made to `VisitAllBookmarks` (that is, the user-defined method is invoking itself) until there are no more children bookmarks. Then the `PDBookmarkGetNext` method is invoked to get a sibling bookmark and the process continues until there are no more bookmarks within the PDF document.

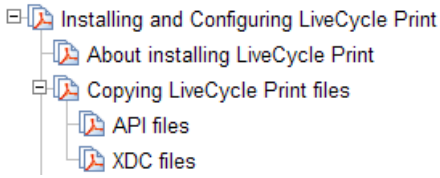
Example: *Retrieving existing bookmarks*

```
//Recursively go through bookmark tree to visit each bookmark
void VisitAllBookmarks (PDBookmark aBookmark)
{
   PDBookmark treeBookmark;
   DURING

   //Ensure that the bookmark is valid
   if (!PDBookmarkIsValid(aBookmark))
```

```
      E_RTRN_VOID

   //Get the title of the bookmark
   char * bmBuf;
   ASInt32 bufSize = PDBookmarkGetTitle(aBookmark, NULL, 0) +1;

   //Allocate the size of bufSize to the character pointer
   bmBuf = (char*)ASmalloc((os_size_t)bufSize);

   //Populate bmBuf with the bookmark's title
   PDBookmarkGetTitle(aBookmark, bmBuf, bufSize);

   //Display the title of the bookmark within a message box
   AVAlertNote(bmBuf);

   //Determine if the current bookmark has children bookmark
   if (PDBookmarkHasChildren (aBookmark))
   {
   //Get the first child of the bookmark
      treeBookmark = PDBookmarkGetFirstChild(aBookmark);
      while (PDBookmarkIsValid (treeBookmark)){
      VisitAllBookmarks (treeBookmark);
      treeBookmark = PDBookmarkGetNext(treeBookmark);
      }
   }
   HANDLER

   END_HANDLER
}
```

Deleting bookmarks

You can use the Acrobat core API to delete an existing bookmark. Deleting a bookmark deletes child bookmarks; however, PDF document content is not affected. To delete a bookmark, you must invoke the `PDBookmarkDestroy` method and pass a `PDBookmark` object that represents the bookmark to delete. For example, consider the bookmark structure shown in the following diagram.



Assume, for example, that you want to delete the bookmark titled Copying Print files. Once you delete this bookmark, the API files and XDC files bookmarks are also deleted. To delete the Copying Print files bookmark, you must create a `PDBookmark` object that represents this bookmark and pass this object to the `PDBookmarkDestroy` method.

The following code example deletes a bookmark. Included in this code example is application logic that retrieves a specific bookmark. (See "Retrieving a specific bookmark" on page 112.)

Example: *Deleting a bookmark*

```
//Retrieve a bookmark whose title is Samples
```

```
PDBookmark rootBookmark, myBookmark;
char* bookmarkTitle = "Copying Print files";

//Get the root bookmark
rootBookmark = PDDocGetBookmarkRoot(myPDDoc);

//Retrieve a specific bookmark
myBookmark = PDBookmarkGetByTitle(rootBookmark, bookmarkTitle,
strlen(bookmarkTitle), -1);
if (PDBookmarkIsValid (myBookmark))
   AVAlertNote("The bookmark was retrieved");
else
   AVAlertNote("The bookmark was not retrieved");

//Delete this bookmark
PDBookmarkDestroy(myBookmark);
```

Note: In the previous code example, a `PDDoc` object named `myPDDoc` is passed to the `PDDocGetBookmarkRoot` method. For information about creating this object, see "Creating a PDDoc object" on page 76.

10 Working with Page Views and Contents

This chapter explains how to display page views and modify page contents. A page view is the area of the Acrobat or Adobe Reader window that displays the visible content of a document page. An example of a page view is a PDF document page displayed within Adobe Reader or Acrobat at a 120% magnification.

About page coordinates

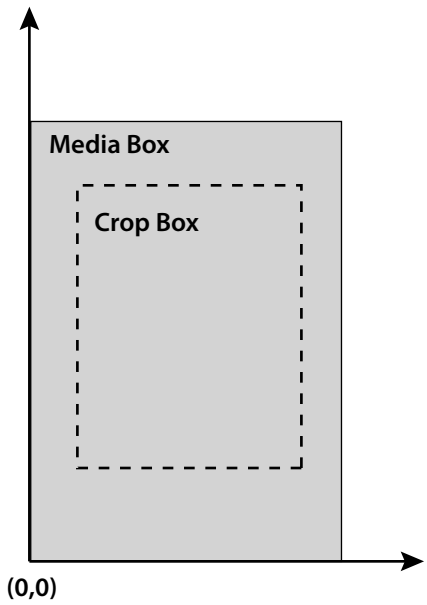
When working with page views and page contents, most times it is necessary to specify page coordinates. Two coordinate systems are applicable to the Acrobat core API:

- User space
- Device space

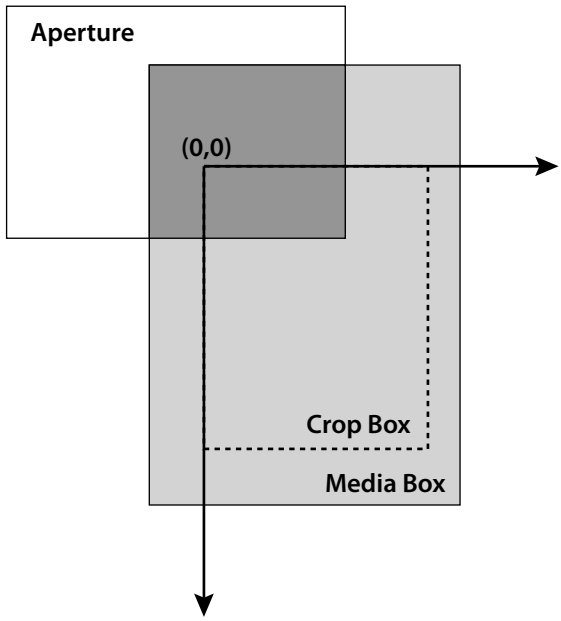
User space is the coordinate system used within PDF documents. It specifies coordinates for most objects in the PD layer. (See "Portable Document layer" on page 16.)

When working with the user space coordinate system, you use an `ASFixedRect` object to represent a rectangle. For example, to place an annotation on a page, create an `ASFixedRect` object and specify its coordinates. To specify a `ASFixedRect` object's coordinates, you must define its `top`, `right`, `bottom`, and `left` attributes. (See "Creating text annotations" on page 100.)

The following diagram shows a user space coordinate system.



Device space specifies coordinates in screen pixels and is used in the AV layer to specify screen coordinates of windows. In device space, you use an `AVRect` object to represent a specific coordinate. The following diagram shows a device space coordinate system.



The `AVPageViewRectToDevice` method can transform a rectangle's coordinates from user space to device space. For example, you can use PD methods to determine user space coordinates of a rectangle. However, to display an outline around the rectangle, you must convert user space coordinates to device space coordinates. (See "Modifying page contents" on page 118.)

About page views

A page view is represented by an `AVPageView` object. To create an `AVPageView` object, invoke the `AVDocGetPageView` method. (See Displaying page views.)

Using `AVPageView` methods, you can perform page-view tasks such as:

- Displaying a page.
- Selecting a zoom factor.
- Scrolling the displayed page.
- Highlighting one or more words.
- Controlling screen redrawing.
- Traversing the view stack that records where users have been in a document.

Note: To control the size of the page view, you can invoke the `AVWindowSetFrame` and `AVDocSetSplitterPosition` methods. (See the Acrobat and PDF Library API Reference.)

Displaying page views

Using the Acrobat core API, you can programmatically display a page view by performing the following tasks:

1. Create an `AVDoc` object that represents the PDF document that contains the page on which the page view is based. (See "Opening PDF documents" on page 64.)

2. If desired, ensure that the specified page number does not exceed the number of pages located in the document. Convert the `AVDoc` object to a `PDDoc` object by invoking the `AVDocGetPDDoc` method. This method requires an `AVDoc` and returns a `PDDoc` object. Get the total number of pages located within the document by invoking the `PDDocGetNumPages` method. This method requires a `PDDoc` object and returns an `ASInt32` object representing the number of pages within the document.

3. Create an `AVPageView` object that represents the page view by invoking the `AVDocGetPageView` method. This method requires an `AVDoc` object and returns an `AVPageView` object.

4. Specify the page view's page number by invoking the `AVPageViewGoTo` method. This method requires an `AVPageView` object that represents the page view and an `ASInt32` object that specifies the page number. The page number uses a zero-based index value. Therefore the value of the first page is `0`.

5. Display the page view by invoking the `AVPageViewDrawNow` method and passing the `AVPageView` object. When you invoke a method that affects a page view, you must invoke this method to redraw the page and display the page view. Otherwise, changes to a page view are not displayed.

The following code example displays a page view that is based on an `AVDoc` object named `myDocument`. The specified page of the page view is 40 (which displays page 41). For information about creating an `AVDoc` object, see "Opening PDF documents" on page 64.

Example: *Displaying a page view*

```
//Create local variables
ASInt32 totalPages;
ASInt32 pageNum= 40;
AVPageView pageView;

//Convert the AVDoc to a PDDoc and get the page count
PDDoc pdDoc = AVDocGetPDDoc(myDocument);
totalPages= PDDocGetNumPages(pdDoc);

if (pageNum < totalPages){

   //Get a AVPageView object and display it
   pageView = AVDocGetPageView(myDocument);
   AVPageViewGoTo(pageView,pageNum);
   AVPageViewDrawNow (pageView);
}
```

Modifying page contents

Pages in a PDF document are represented by a `PDPage` object. Pages can contain properties such as annotations, contents, resources, bounding box, media box, crop box, rotation state, and Cos objects. These properties make up the page's content. `PDPage` methods enable you to perform tasks such as retrieving objects such as annotations, adding new objects, removing objects, and modifying existing objects.

To access a PDPage object, you must obtain the applicable PDDoc object, either directly or by invoking the AVDocGetPDDoc method. You can then invoke the PDDocAcquirePage method to acquire the page (the PDPage object).

To access the contents of PDF pages, you use PD layer methods. In addition, the Acrobat core API provides the PDFEdit typedef, which provides an easy way to access to the contents of a PDF page. PDFEdit methods are useful when working with page items such as images, paths, graphics, and text objects. This API treats the contents of a page as a list of objects whose values and attributes can be modified.

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. For information see "Portable Document layer" on page 16.

Creating a PDEContent object

A PDEContent object is a container object that contains modifiable contents of a PDPage including PDEText, PDEPath, and PDEImage objects. After you create a PDEContent object, you can access and modify objects that it contains.

Create a PDEContent object by performing the following tasks:

1. Create an AVDoc object by invoking the AVAppGetActiveDoc method. This method gets the frontmost document located within Adobe Reader or Acrobat.
2. Create an AVPageView object by invoking the AVDocGetPageView method. This method requires an AVDoc object and returns an AVPageView object.
3. Get the current page number of the PDF document by invoking the AVPageViewGetPageNum method. This method requires an AVPageView object and returns a PDPageNumber object that represents the current page number.
4. Create a PDDoc object by invoking the AVDocGetPDDoc method. This method requires an AVDoc object and returns a PDDoc object.
5. Create a PDPage object by invoking the PDDocAcquirePage method. This method requires a PDDoc object and a PDPageNumber object as arguments and returns a PDPage object.
6. Create a PDEContent object by invoking the PDPageAcquirePDEContent method and passing a PDPageNumber object and an ASExtension object that represents the identity of the caller. For plugins, you can use the gExtensionID extension (this is defined in the PIMain.c file).

The following code example creates a PDEContent object.

Example: *Creating a PDEContent object*

```
AVDoc avDoc = AVAppGetActiveDoc();
AVPageView pageView = AVDocGetPageView(avDoc);
PDPageNumber pageNum = AVPageViewGetPageNum(pageView);

/* Bridge method to PD doc*/
PDDoc pdDoc = AVDocGetPDDoc(avDoc);

/* Acquire current page */
PDPage pdPage = PDDocAcquirePage(pdDoc, pageNum);
```

```
PDEContent pdeContent = PDPageAcquirePDEContent(pdPage, gExtensionID);
```

Accessing page contents

Before you can modify page contents, you must access them by using a `PDEContent` object, which is a collection object that stores page contents. Each page content is stored as an element within the `PDEContent` object. An element is represented by a `PDEElement` object. For example, a `PDEElement` object can contain an annotation.

To access page contents, perform the following tasks:

1. Create a `PDEContent` object. (See "Creating a PDEContent object" on page 119.)
2. Invoke the `PDEContentGetNumElems` method to retrieve the number of elements located within the `PDEContent` object. This method requires a `PDEContent` object and returns an `ASInt32` object that specifies the number of elements.
3. Iterate through the `PDEContent` object and retrieve each element by invoking the `PDEContentGetElem` method. This method requires a `PDEContent` object and an `ASInt32` object that specifies the element index (this is a zero-based value) and returns a `PDEElement` object that represents a specific page property.

The following code example accesses each element located in a `PDEContent` object.

Example: *Accessing page contents*

```
//Declare a PDEElement object
PDEElement pdeElement;

//Get the number of elements in the PDEContent object
ASInt32 eleNum =  PDEContentGetNumElems(pdeContent);

//Get each element in the PDEContent object
for (int j=0; j<eleNum; j++){

   pdeElement = PDEContentGetElem(pdeContent, j);
}
```

Determining page element types

You can determine the element type that a `PDEElement` object represents by invoking the `PDEObjectGetType` method. This method requires a `PDEObject`; however, you can pass a `PDEElement` object and cast it to a `PDEObject`. This method returns an `ASInt32` object that specifies the element type. For example, if the element is a text element, this method returns `kPDEText`. For a listing of all element types, see the Acrobat and PDF Library API Reference.

The following code example determines the element type by adding application logic to the application logic introduced in "Accessing page contents" on page 120.

Example: *Determining page element types*

```
//Declare a PDEElement object
PDEElement pdeElement;
```

```
//Get the number of elements located in the PDEContent object
ASInt32 eleNum =  PDEContentGetNumElems(pdeContent);

//Get each element located within the PDEContent object
for (int j=0; j<eleNum; j++){

   pdeElement = PDEContentGetElem(pdeContent, j);

   //Determine if the element is a text element
   if (PDEObjectGetType((PDEObject)pdeElement) == kPDEText){

   //Perform an action
}
```

Modifying text elements

You can modify elements located within a `PDEContent` object. This section discusses modifying text elements by placing a red border around them. To place a red border around a text element, you must create a `PDColorValueRec` object and define its attributes.

```
//Create a PDColorValue and define red
PDColorValueRec red;
red.space = PDDeviceRGB;
red.value[0] = ASInt32ToFixed(1);
red.value[1] = 0;
red.value[2] = 0;
```

After you create a `PDColorValueRec` object, you can attach it to an `AVPageView` object by invoking the `AVPageViewSetColor` method. This method requires an `AVPageView` object and a `PDVColorValueRec` object. Once set, this color is used in drawing operations.

A text element is represented by a `PDEText` object, which is a container of text runs. A text run can be a single character or multiple characters having the same attributes in a PDF file. You can get the number of text runs located within a `PDEElement` object by invoking the `PDETextGetNumRuns` method and passing the `PDEElement` object and casting it as a `PDEText` object.

```
int numRuns = PDETextGetNumRuns((PDEText) pdeElement);
```

You can draw a red border around each `PDEText` object by performing the following tasks:

1. Get the bounding box of the `PDEText` object by invoking the `PDETextGetBBox` method and passing the following arguments:
   - A `PDEText` object that represents the text element whose bounding box is obtained.
   - A `PDETextFlags` value that specifies whether index refers to the character offset from the beginning of the text object or the index of the text run. Must be either `kPDETextChar` for a text character or `kPDETextRun` for a text run.
   - An `ASInt32` value that specifies the index of the character or text run.
   - The address of an `ASFixedRect` object that is populated with the coordinates of the bounding box of a specified character or text run.
2. Transform the bounding box's coordinates from user space to device space by invoking the `AVPageViewRectToDevice` method and passing the following arguments:

- An `AVPageView` object that represents the page view for which the coordinates are transformed. Use the same `AVPageView` object that was used to create a `PDEContent` object. (See "Creating a PDEContent object" on page 119.)
- The address of the `ASFixedRect` object that was passed to the `PDETextGetBBox` method. This object contains coordinate data that is transformed.
- The address of an `AVRect` object that is populated with device space coordinates.

3. Draw a stroked, but not filled, rectangle by invoking the `AVPageViewDrawRectOutline` method and passing the following arguments:
   - An `AVPageView` object that represents the page view in which the rectangle is drawn.
   - The address of the `AVRect` object that specifies device space coordinates. You can pass the address of the same `AVRect` object that was passed to the `AVPageViewRectToDevice` method.
   - An `AVDevSize` object that specifies the border width in pixels.
   - The address of an `ASFixed` object whose elements specify the length of dashes and gaps. You can specify `NULL` to draw a solid outline.
   - An `AVTArraySize` object that specifies the number of elements in the `ASFixed` object. This value is ignored if you specified `Null` as the previous argument. As a result, you can specify `0` for this argument. The maximum allowed number of elements is currently 10.

The following code example modifies page contents by placing a red border around text elements and places a blue border around other elements that are located within a `PDEContent` object.

Example: *Modifying page contents*

```
//Declare objects used in this code example
PDEElement pdeElement;
ASFixedRect bbox;
AVRect rect;
PDColorValueRec red, blue;

//Define red
red.space = PDDeviceRGB;
red.value[0] = ASInt32ToFixed(1);
red.value[1] = 0;
red.value[2] = 0;

//Define blue
blue.space = PDDeviceRGB;
blue.value[0] = 0;
blue.value[1] = 0;
blue.value[2] = ASInt32ToFixed(1);

//Create a PDEContent object based on the current page view
AVDoc avDoc = AVAppGetActiveDoc();
AVPageView pageView = AVDocGetPageView(avDoc);
PDPageNumber pageNum = AVPageViewGetPageNum(pageView);
PDDoc pdDoc = AVDocGetPDDoc(avDoc);
PDPage pdPage = PDDocAcquirePage(pdDoc, pageNum);
PDEContent pdeContent = PDPageAcquirePDEContent(pdPage, gExtensionID);

//Get the number of elements located in the PDEContent object
ASInt32 eleNum = PDEContentGetNumElems(pdeContent);
```

```
//Retrieve each element in the PDEContent object
for (int  j=0; j < eleNum; j++){

   //Get a specific element
   pdeElement = PDEContentGetElem(pdeContent, j);

   //Determine if the object is of type text
   if (PDEObjectGetType((PDEObject) pdeElement) == kPDEText){

      //Get the number of text runs in the text element
      int numTextRuns = PDETextGetNumRuns((PDEText) pdeElement);

      //Assign red to the page view
      AVPageViewSetColor(pageView, &red);

   for (int i = 0; i < numTextRuns; 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);

      //Draw the rect
      AVPageViewDrawRectOutline (pageView,&rect,  1, NULL, 0);
      }
   }
   else
   {
      //Assign blue to the page view
      AVPageViewSetColor(pageView, &blue);

      //Get the bounding box of the non-text element
      PDEElementGetBBox (pdeElement, &bbox);
      //Convert from user space bbox to device space rect
      AVPageViewRectToDevice (pageView, &bbox, &rect);
      //Draw the rect
      AVPageViewDrawRectOutline (pageView, &rect, 1, NULL, 0);
   }
}

//Release objects
PDPageRelease(pdPage);
PDPageReleasePDEContent (pdPage, gExtensionID);
```

11 Working with Words

You can use the Acrobat core API to search for words, extract and display words, and highlight words. Using the Acrobat core API, you can, for example, create application logic that extracts words from a PDF document and places each word in a repository.

About searching for words

The Acrobat core API provides typedefs and methods for working with words. Two primary typedefs that you will use when working with words located in a PDF document are `PDWord` and `PDWordFinder`. The following are two word-finding indicators:

- 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 the Acrobat or Adobe Reader word finder algorithm).

About PDWord typedefs

A `PDWord` object represents a word in a PDF file. Each word contains a sequence of characters in one or more styles. All characters in a word are not necessarily physically adjacent. For example, words can be hyphenated across line breaks on a page.

Each character in a word has a character type. Character types include: control code, lowercase letter, uppercase letter, digit, punctuation mark, hyphen, soft hyphen, ligature, white space, comma, period, unmapped glyph, end-of-phrase glyph, wildcard, word break, and glyphs that cannot be represented in the destination font encoding. (See the Acrobat and PDF Library API Reference.)

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

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

A word's location is specified by the offset of its first character from the beginning of the page (known as the character offset). The characters are enumerated in the order in which they appear in page's content stream in the PDF file (which is not necessarily the order in which the characters are read when displayed or printed).

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

About PDWordFinder typedefs

A `PDWordFinder` extracts words from a PDF file, and enumerates the words on a single page or on all pages in a document. The Acrobat core API provides methods to extract words from a document, obtain information on the word finder, and to release a list of words.

Two primary methods of working with word finders are:

- Invoking the `PDWordFinderEnumWords` method, which invokes a user-defined callback function each time a word is recognized on a page. (See "Extracting and displaying words" on page 127.)
- Using `PDWordFinderAcquireWordList`, which builds a word list for an entire page before it returns. This method can return the recognized words in two possible orders:
  - The order in which the words are encountered in the PDF file.
  - According to word location on the page. For a page containing a single column of text, this generally is the same as reading order. For a page containing multiple columns of text, this is not true.

Creating a PDWordFinder object

To perform word operations, such as extracting and displaying words located in a PDF document, you must create a `PDWordFinder` object. You can create a `PDWordFinder` object by getting the active document (the frontmost document in Acrobat or Adobe Reader).

Optionally, you can create a `PDWordFinderConfigRec` object when creating a `PDWordFinder` object. A `PDWordFinderConfigRec` object enables you to customize how text is extracted. After you create an `PDWordFinderConfigRec` object, allocate its buffer size and set the following attributes:

recsize: The size of the data structure. This attribute must be set to `sizeof(PDWordFinderConfigRec)`.

ignoreCharGaps: If `true`, this attribute disables the conversion of large character gaps to space characters, so that the word finder reports a character space only when a space character appears in the PDF content.

ignoreLineGaps: If true, this attribute disables the handling of vertical movements as line breaks, so that the word finder determines a line break only when a line break character or special tag information appears in the PDF content.

noAnnots: If true, this attribute disables the extraction of text from text annotations. Normally, the word finder extracts text from the normal appearances of text annotations that are inside the page crop box.

noEncodingGuess: If true, disables the guessing of the encoding of fonts that have unknown or custom encoding, when there is no ToUnicode table. Inappropriate encoding conversions can cause the word finder to mistakenly recognize non-Roman single-byte fonts as Standard Roman encoding fonts and extract the text in an unusable format. When this option is selected, the word finder avoids such unreliable encoding conversions and tries to provide the original characters without any encoding conversion for a client with its own encoding handling.

Note: For a complete list of attributes that belong to a `PDWordFinderConfigRec` object, see the Acrobat and PDF Library API Reference.

Create a `PDWordFinder` object that is based on an active document by performing the following tasks:

1. Create an `AVDoc` object by invoking the `AVAppGetActiveDoc` method. (See "Opening PDF documents" on page 64.)
2. Create a `PDDoc` object by invoking the `AVDocGetPDDoc` method and passing the `AVDoc` object.
3. If desired, create a `PDWordFinderConfigRec` object. If you do not create a `PDWordFinderConfigRec` object, then default configuration is used. That is, all attributes that belong to an `PDWordFinderConfigRec` object are false.
4. Create a `PDWordFinder` object by invoking the `PDDocCreateWordFinderEx` method and passing the following arguments:
   - A `PDDoc` that represents the PDF document for which the word finder is applicable.
   - An `ASInt16` value that specifies the version of the word-finding algorithm to use. You can specify `WF_LATEST_VERSION` to use the latest version. For information about other values for this argument, see the Acrobat and PDF Library API Reference.
   - An `ASBool` value that specifies whether to return Unicode. When `true`, the word finder encodes the extracted text in Unicode format. Otherwise, the word-finding algorithm extracts the text in the host encoding format.
   - The address of the `PDWordFinderConfigRec` object to use. You can pass `NULL`, which results in the default configuration being used.

The following code example creates a `PDWordFinder` object.

Example: *Creating a PDWordFinder object that is based on the current PDF document*

```
//Get the current PDF document
AVDoc currentAVDoc = AVAppGetActiveDoc();
PDDoc currentPDDoc = AVDocGetPDDoc(currentAVDoc);

//Create a PDWordFinderConfigRec object;
PDWordFinderConfigRec pConfig;

//Set the DWordFinderConfigRec object's attributes
memset(&pConfig, 0, sizeof(PDWordFinderConfigRec));
pConfig.recSize = sizeof(PDWordFinderConfigRec);
pConfig.ignoreCharGaps = true;
pConfig.ignoreLineGaps = true;
pConfig.noAnnots = true;
pConfig.noEncodingGuess = true;

//Create a PDWordFinder object
PDWordFinder pdWordFinder = PDDocCreateWordFinderEx(currentPDDoc,
WF_LATEST_VERSION, false, &pConfig);
```

Extracting and displaying words

You can use a `PDWordFinder` object to extract and display all words that are located either in the entire document or the current page by creating a callback function that is invoked for each word found. To create a callback function that is invoked when a word is found, declare a `PDWordProc` object that represents the callback:

```
PDWordProc wordProc;
```

`PDWordProc` is a callback that is invoked when a word is located. After you create a `PDWordProc` object, you can invoke the `ASCallbackCreateProto` macro to convert a user-defined function to an Acrobat callback. For example, you can invoke `ASCallbackCreateProto` to convert a user-defined function named `wordEnumerator` to a callback function. The `ASCallbackCreateProto` macro requires the following arguments:

- The callback type. In this situation, specify `PDWordProc`.
- The address of the user-defined function to convert to a callback function.

The `ASCallbackCreateProto` macro returns a callback of the specified type that invokes the user-defined function whose address was passed as the second argument. The following lines of code shows the `ASCallbackCreateProto` macro converting the `wordEnumerator` user-defined function to a `PDWordProc` callback.

```
PDWordProc wordProc;
wordProc= ASCallbackCreateProto(PDWordProc, &wordEnumerator);
```

After you create a callback function, invoke the `PDWordFinderEnumWords` method to extract all words from the specified page and pass the following arguments:

- A `PDWordFinder` object that is responsible for finding and extracting words.
- An `ASInt32` value that represents the page number from which to extract words.
- A `PDWordProc` object that represents the callback function to invoke when a word is located.
- A pointer to user-supplied data to pass to the callback function. Pass `NULL` if you do not want to pass user-supplied data.

To illustrate how to display words that are located on a page, this section contains a code example that creates a callback function named `wordEnumerator` that performs the following tasks:

- Removes punctuation characters from the word by invoking the `PDWordFilterWord` method. The encoding information passed to the `PDDocCreateWordFinderEx` method determines which characters are removed.
- Invokes the `PDWordGetString` method to get the word as a string.
- Displays the string in an alert box by invoking the `AVAlertConfirm` method. If the user clicks OK, the next word is displayed until all words for the document page have been displayed. If the user clicks Cancel, this callback function returns `false`.

The following code example extracts and displays all words that are located on the current PDF document page. Included in this code example is application logic that creates a `PDWordFinder` object. (See "Creating a PDWordFinder object" on page 125.)

Example: *Extracting and displaying words*

```
ACCB1 void ACCB2 DisplayWords(void *data)
{
//Get the current PDF document page number
```

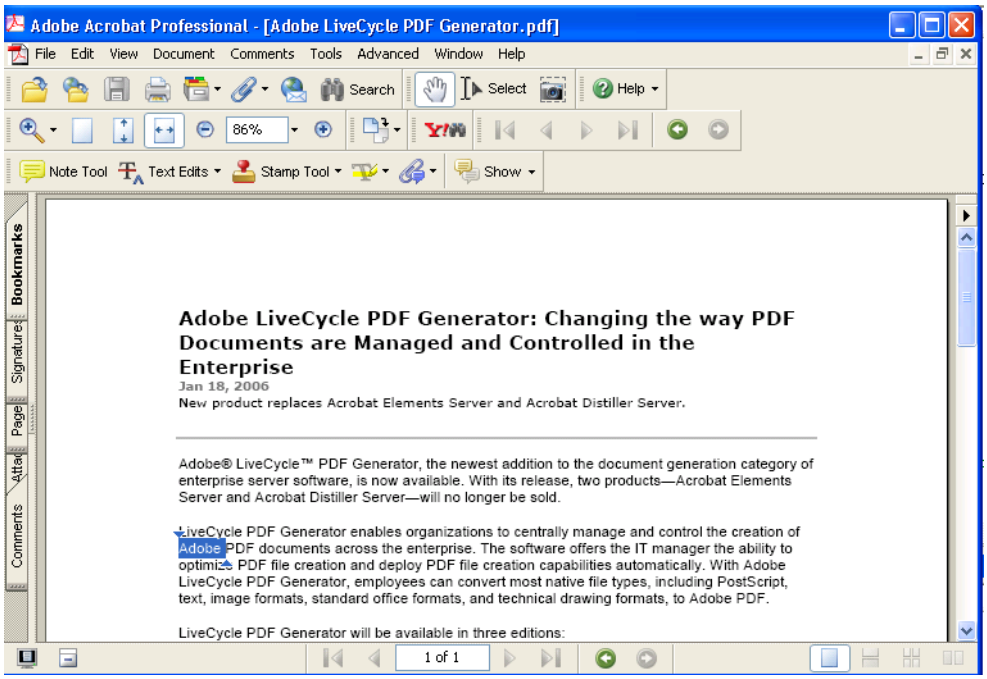
```
AVDoc currentAVDoc = AVAppGetActiveDoc();
PDDoc currentPDDoc = AVDocGetPDDoc(currentAVDoc);
AVPageView currentPageView = AVDocGetPageView (currentAVDoc);
ASInt32 pageNum = AVPageViewGetPageNum(currentPageView);

//Create a PDWordFinderConfigRec object;
PDWordFinderConfigRec pConfig;

//Set the DWordFinderConfigRec object's attributes
memset(&pConfig, 0, sizeof(PDWordFinderConfigRec));
pConfig.recSize = sizeof(PDWordFinderConfigRec);
pConfig.ignoreCharGaps = true;
pConfig.ignoreLineGaps = true;
pConfig.noAnnots = true;
pConfig.noEncodingGuess = true;

//Create a PDWordFinder object
PDWordFinder pdWordFinder = PDDocCreateWordFinderEx(currentPDDoc,
WF_LATEST_VERSION, false, &pConfig);

//Create a callback function
PDWordProc wordProc = NULL;
wordProc= ASCallbackCreateProto(PDWordProc, &wordEnumerator);

//Extract and display words
PDWordFinderEnumWords(pdWordFinder, pageNum, wordProc, NULL);
PDWordFinderDestroy(pdWordFinder);
}
ACCB1 ASBool ACCB2 wordEnumerator(PDWordFinder wObj, PDWord wInfo, ASInt32
pgNum, void *clientData)
{
char stringBuffer[100];
ASInt16 wordLength;

//Remove punctuation
PDWordFilterWord(wInfo, stringBuffer, 99, &wordLength);
stringBuffer[wordLength] = 0;

//Populate the char array with text that represents the word
PDWordGetString (wInfo, stringBuffer, 99);
return AVAlertConfirm(stringBuffer);
}
```

Note: In the previous code example, assume that the `DisplayWords` function was invoked from a menu item. (See "Creating Menus and Menu Commands" on page 82.)

Caution: If you pass `true` as the `PDDocCreateWordFinderEx` method's third argument, then the word finder encodes the extracted text in Unicode format. As a result, words will not be displayed within the alert box. Notice in this code example, the value `false` is passed as the `PDDocCreateWordFinderEx` method's third argument.

Highlighting words

You can use the Acrobat core API to highlight a word or a group of words located within a PDF document. By highlighting a word, you can make a specific word or group of words stand out. The following illustration shows the word *Adobe* highlighted.



To highlight a word you must create a `HiliteEntry` object and set its `offset` and length `attributes`. The `offset` attribute specifies the location of the word from the beginning of the document. For example, if you specify 1, then the second word in the document is highlighted (this value is a 0-based index). The `length` attribute specifies the number of words that are highlighted. If you specify 1, then a single word is highlighted.

You can highlight a word that is located in the current page by performing the following tasks:

1. Create a `HiliteEntry` object and set its `offset` and length `attributes`.
2. Create an `AVDoc` object by invoking the `AVAppGetActiveDoc` method. (See "Opening PDF documents" on page 64.)
3. Create a `PDDoc` object by invoking the `AVDocGetPDDoc` method and passing the `AVDoc` object.
4. Get the page view by invoking the `AVDocGetPageView` method and passing the `AVDoc` object. This method returns an `AVPageView` object. (See "Displaying page views" on page 117.)
5. Get the current page number by invoking the `AVPageViewGetPageNum` method and passing the `AVPageView` object. This method returns the page number of the current page view, which is required to highlight a word or group of words.
6. Create a `PDPage` object by invoking the `PDDocAcquirePage` method and passing the following arguments:
   - A `PDDoc` object.

- The page number of the current page view.

The `PDDocAcquirePage` method returns a `PDPage` object.

7. Highlight a word or group of words by invoking the `PDTextSelectCreateWordHilite` method and passing the following arguments:
   - A `PDPage` object that represents the page that will contain the highlighted word(s).
   - The address of the `HiliteEntry` object.
   - An `ASInt32` value that specifies the number of highlight entries.

   This method returns a `PDTextSelect` object.
8. Set the PDF document's text selection type by invoking the `AVDocSetSelection` method. This method does not have a return value and requires the following arguments:
   - An `AVDoc` object that represents the PDF document in which the highlighted words appear.
   - An `ASAtom` object that specifies the selection type. Because words are highlighted, you can specify `text`.
   - A `PDTextSelect` object that represents the text selection. Cast the `PDTextSelect` object as a void pointer.
   - An `ASBool` object that specifies whether to highlight the selection. Pass the value `true` to highlight the specified word(s).
9. Display the current selection by invoking the `AVDocShowSelection` method and passing the `AVDoc` that represents the PDF document that contains the highlighted word(s).
10. Release the `PDPage` object by invoking the `PDPageRelease` method and passing the `PDPage` object.

The following code example highlights the tenth word that is located in the page of the current PDF document.

Example: *Highlighting a word in a PDF document*

```
//Create a HiliteEntry object and set its attributes
HiliteEntry hilite;
hilite.offset = 10;
hilite.length = 1;

//Get the page number of the current page view
AVDoc currentAVDoc = AVAppGetActiveDoc();
PDDoc currentPDDoc = AVDocGetPDDoc(currentAVDoc);
AVPageView currentPageView = AVDocGetPageView(currentAVDoc);
ASInt32 pageNum = AVPageViewGetPageNum(currentPageView);

//Highlight the tenth word
PDPage pdPage = PDDocAcquirePage (currentPDDoc, pageNum);
PDTextSelect textSelection = PDTextSelectCreateWordHilite(pdPage,
&hilite, 1);
AVDocSetSelection(currentAVDoc, ASAtomFromString("Text"),
(void *)textSelection, true);
AVDocShowSelection (currentAVDoc);
PDPageRelease (pdPage);
```

12 Creating Handlers

Adobe Reader and Acrobat plugins and PDF Library applications can add new types of tools, annotations, actions, file systems, and so on, thereby expanding the number of supported object types. To accomplish this task, the Acrobat core API provides a collection of callback routines called handlers that support objects. Handlers perform operations, such as creating and destroy objects, handling mouse clicks, handling keyboard events, and so on.

About handlers

To add a new handler, you must write callback functions, create the appropriate data structure containing the callbacks and other data, and pass the structure to Acrobat by invoking the appropriate method. Subsequently, Acrobat automatically invokes the correct callback when it encounters an object of the type handled by the handler.

It is possible to subclass existing handlers or to create entirely new handler types. For example, a plugin can subclass the built-in text annotation handler by adding the ability to hide annotations. To accomplish this task, perform the following tasks:

1. Obtain the built-in text annotation handler structure by invoke the `AVAppGetAnnotHandlerByName` method.
2. Copy the structure before modifying it (not modifying the original).
3. Replace the handler's `Draw` callback with one that invokes the built-in `Draw` callback (obtained from the structure) if annotations are visible, or simply return without drawing anything if annotations are hidden.
4. Register the new handler by invoking the `AVAppRegisterAnnotHandler` method with a new type.

If a handler requires more data than provided in the predefined structures that are described in this section, you can append additional data to the predefined structures. To do this, create a new structure type with the predefined structure as its first member and the additional data as subsequent members. Before passing the expanded structure to an Acrobat method, cast the structure to the predefined structure type. Upon return of the structure from Acrobat, re-cast the structure to its expanded type to access the appended data.

Each handler data structure contains a size field, which specifies the structure's size. This field provides future compatibility. Different versions of the structure have different sizes, allowing Acrobat to determine which version your plugin was written to use.

Note: Regardless of whether your plugin adds data to the predefined structures, it must pass the size of the predefined structure (rather than the size of its expanded structure) in the size field.

Action handlers

Support for action types can be added by defining and registering an action handler. For example, the Acrobat Weblink plugin uses this ability to add support for URL links.

To add a new action type, you must provide a set of callbacks. Specify them in the `AVActionHandlerProcs` structure, and invoke the `AVAppRegisterActionHandler` method to register them.

By using an action handler, you can perform the following tasks:

- Perform an action, such as setting a specific view, by invoking the `AVActionPerformProc` method.
- Allow the user to set the action's properties (if the properties can be set) by invoking the `AVActionDoPropertiesProc` method.
- Initialize an action's dictionary with default values by invoking the `AVActionFillActionDictProc` method.
- Display a string containing brief instructions for the action by invoking the `AVActionGetInstructionsProc` method.
- Display various text strings to be used in dialog boxes by invoking one of the following methods: `AVActionGetButtonTextProc`, `AVActionGetStringOneTextProc`, or `AVActionGetStringTwoTextProc`.
- Copy the action by invoking the `AVActionCopyProc` method.

Annotation handlers

Support for annotation types in Acrobat can be added by defining and registering an annotation handler. For example, the Acrobat movie plugin uses an annotation handler to support video annotations.

To add an annotation type, you must provide a set of callbacks, specify them in the `AVAnnotHandler` structure, and register them with `AVAppRegisterAnnotHandler`.

By using an annotation handler, you can perform the following tasks:

- Draw the annotation by using the `AVAnnotHandlerDrawProc` callback method.
- Handle mouse clicks in the annotation by using the `AVAnnotHandlerDoClickProc` callback method.
- Control the cursor shape when the cursor is over the annotation by using the `AVAnnotHandlerAdjustCursorProc` callback method.
- Determine whether or not a specified point is within the annotation boundary by using the `AVAnnotHandlerPtInAnnotViewBBoxProc` callback method.
- Return the rectangle bounding that the annotation occupies by using the `AVAnnotHandlerGetAnnotViewBBoxProc` callback method.
- Highlight (unhighlight) the annotation when it is added to (removed from) the selection by using the following callback methods: `AVAnnotHandlerNotifyAnnotAddedToSelectionProc` or `AVAnnotHandlerNotifyAnnotRemovedFromSelectionProc`.
- Return the annotation's subtype by using the `AVAnnotHandlerGetTypeProc` callback method.
- Get the annotation's layer by using the `AVAnnotHandlerGetLayerProc` callback method.

Note: For information about working with annotations, see "Creating Annotations" on page 100.

AVCommand handlers

An AVCommand represents an action that a user can perform on the current document or the current selection in the current document. AVCommands are exposed to Adobe Reader or Acrobat through AVCommand handlers. You can add new command types by defining and registering an AVCommand handler. Commands can be executed interactively, programmatically, or through batch processing.

Creating an AVCommand handler

AVCommand handlers consist of a series of callback functions contained in the AVCommandHandlerRec structure (see AVExpt.h). To create a command handler, perform the following tasks:

1. Initialize an instance of the AVCommandHandlerRec structure.
2. Register the AVCommandHandlerRec structure by invoking the AVAppRegisterCommandHandler method.

The following code example creates an AVCommand handler.

Example: *Creating an AVCommand handler*

```
static AVCommandHandlerRec gAVCmdHandler;
const char *kCmdName = "MinimalCommand";
static ACCB1 AVCommandStatus ACCB2 DoWorkImpl (AVCommand cmd)
{
   AVAlertNote("The DoWorkImpl method was invoked");
   return kAVCommandDone;
}
void InitializeCommandHandler()
{
   memset (&gAVCmdHandler, 0, sizeof(AVCommandHandlerRec));
   gAVCmdHandler.size = sizeof(AVCommandHandlerRec);
   gAVCmdHandler.Work = ASCallbackCreateProto (AVCommandWorkProc,
   &DoWorkImpl);
   AVAppRegisterCommandHandler (ASAtomFromString(kCmdName),
   &gAVCmdHandler);
}
```

Note: To view a complete example, see "Running commands" on page 135.

Invoking AVCommands

To programmatically invoke AVCommands using AVCommand methods, perform the following tasks:

1. Instantiate the command by invoking the AVCommandNew method, providing the registered name of the command:

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

2. Configure the command by setting required and optional parameters.
3. Run the command by invoking the AVCommandExecute or AVCommandWork method.

Configuring AVCommands

Prior to executing an `AVCommand`, you configure three categories of properties:

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

Setting input parameters

At minimum, you must configure input parameters. The command must be provided with a `PDDoc` object that represents the PDF document on which to operate, as shown in the following example. For information about a `PDDoc` object, see "Creating a PDDoc object" on page 76.

Example: *Setting input parameters*

```
//Create a PDDoc object based on the current PDF document
AVDoc avDoc = AVAppGetActiveDoc();
AVPageView pageView = AVDocGetPageView(avDoc);
PDPageNumber pageNum = AVPageViewGetPageNum(pageView);
PDDoc pdDoc = AVDocGetPDDoc(avDoc);

//Create an ASCab object to store input parameters
ASCab inputs = ASCabNew();
ASCabPutPointer (inputs, kAVCommandKeyPDDoc, PDDoc, pdDoc, NULL);

//Set the input parameters
if (kAVCommandReady != AVCommandSetInputs(cmd, inputs)) {

   // Handle error

//Destroy the ASCab container
ASCabDestroy (inputs);
```

Note: For more information about the `AVCommandSetInputs` method, see the Acrobat and PDF Library API Reference.

Setting configuration parameters

Optionally you can set configuration parameters. The default UI policy is for commands to be fully interactive. To invoke the command programmatically, create an `ASCab` object and populate it with the appropriate parameters, as shown in the following example.

Example: *Setting configuration parameters*

```
// Create an ASCab object to store config parameters
ASCab config = ACabNew();
ASCabPutInt (config, "UIPolicy", kAVCommandUISilent);

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

Setting AVCommand parameters

An `AVCommand` parameter set is specific to each command. For example, the Document Summary command accepts values for these parameters: `Title`, `Subject`, `Author`, `Keywords`, `Binding`, and `LeaveAsIs`. (See the Acrobat and PDF Library API Reference.)

You can create an `ASCab` object to store the appropriate parameters; then create empty `ASText` objects to hold the parameter values and place these values into the `ASCabs` object. The following example uses this approach to set the Document Summary `Title` and `Subject` values.

Example: *Setting AVCommand parameters*

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

//Create an ASCab object to hold command parameters
ASCab params = ASCabNew();
ASText text = ASTextNew();
ASTextSetEncoded(text, docTitleValue,(ASHostEncoding)PDGetHostEncoding());
ASCabPutText (params, docTitleValue, text);

//Clear the ASText object
text = ASTextNew();
ASTextSetEncoded(text, docSubjectValue,(ASHostEncoding)PDGetHostEncoding());
ASCabPutText(params, docSubjectValue, text);
```

Running commands

The following code example shows an entire example of creating an `AVCommand` and running it.

Example: *Running an AVCommand*

```
void InitializeCommandHandler()
{
//Declare local variables
static AVCommandHandlerRec gAVCmdHandler;
const char *kCmdName = "MinimalCommand";
ASAtom cmdName;
AVCommand cmd;
const char *docTitleValue = "Document Title";
const char *docSubjectValue = "Document Subject";

//Create a PDDoc object based on the current PDF document
AVDoc avDoc = AVAppGetActiveDoc();
AVPageView pageView = AVDocGetPageView(avDoc);
PDPageNumber pageNum = AVPageViewGetPageNum(pageView);
PDDoc pdDoc = AVDocGetPDDoc(avDoc);

//Create an AVCommandHandlerRec object
memset (&gAVCmdHandler, 0, sizeof(AVCommandHandlerRec));
gAVCmdHandler.size = sizeof(AVCommandHandlerRec);
gAVCmdHandler.Work = ASCallbackCreateProto (AVCommandWorkProc, DoWorkImpl);
AVAppRegisterCommandHandler (ASAtomFromString(kCmdName),&gAVCmdHandler);

//Invoke the AVCommand
```

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

//Set the input parameters
ASCab inputs = ASCabNew();
ASCabPutPointer (inputs, kAVCommandKeyPDDoc, PDDoc, pdDoc, NULL);

//Set the input parameters and destroy the container ASCab
if (kAVCommandReady != AVCommandSetInputs (cmd, inputs)) {
// Handle error
}

//Create an ASCab object to hold command parameters
ASCab params = ASCabNew();
ASText text = ASTextNew();
ASTextSetEncoded(text, docTitleValue,(ASHostEncoding)PDGetHostEncoding());
ASCabPutText (params, docTitleValue, text);

//Clear the ASText object
text = ASTextNew();
ASTextSetEncoded(text, docSubjectValue,(ASHostEncoding)PDGetHostEncoding());
ASCabPutText(params, docSubjectValue, text);

//Invoke the command
AVCommandExecute(cmd);
}

static ACCB1 AVCommandStatus ACCB2 DoWorkImpl (AVCommand cmd)
{
AVAlertNote("The DoWorkImpl method was invoked");
return kAVCommandDone;
}
```

Exposing AVCommands to the batch framework

Acrobat or Adobe Reader builds the list of commands that users see in the `Batch Sequences` and `Batch Edit Sequence` dialog boxes from an internal list of `AVCommands` referred to as the global command list.

Adding a handler to the global command list

To expose a command to the batch framework, the `AVCommand` handler must first add an instance of the command to this global list by invoking the `AVAppRegisterGlobalCommand` method.

```
AVCommand cmd = AVCommandNew(ASAtomFromString(kCmdName));
AVAppRegisterGlobalCommand(cmd);
```

Although this step can be performed at any time once the command handler is registered, handlers commonly register commands from within the `AVCommandRegisterCommandsProc` callback (of the `AVCommandHandlerRec` structure).

Supporting properties

When building a list of batchable commands, Adobe Reader or Acrobat iterates through its internal command list, querying each command for the `CanBatch` and `GroupTitle` properties. To be exposed through the batch framework user interface, a command must support these properties (that is, return `true` and a valid `ASText` object, respectively). The `AVCommand` handler must implement the `GetProps` callback of the `AVCommandHandlerRec` structure.

If an `AVCommand` supports these properties, Adobe Reader or Acrobat queries a number of additional properties as the user interacts with the batch framework. Of these additional properties, only two are required: `Title` and `Generic Title`. A command must provide the title strings that will be displayed in the Batch Sequences and Batch Edit Sequence dialog boxes.

Example: *Exposing AVCommands to the batch framework*

```
const char *kCmdTitle = "Command Title";
const char *kGroupTitle = "Group Title";
const char *kCmdGenericTitle = "Generic Title";

ASBool doItAll = false;
if (ASCabNumEntries(params) == 0)
doItAll = true;
if (doItAll || ASCabKnown (params, kAVCommandKeyGroupTitle))
{
// Create a new text object and insert it into the ASCab
text = ASTextNew();
ASTextSetEncoded (text, kGroupTitle, (
ASHostEncoding)PDGetHostEncoding());
ASCabPutText(params, kAVCommandKeyGroupTitle, text);
}
if (doItAll || ASCabKnown (params, kAVCommandKeyCanBatch))
ASCabPutBool (params, kAVCommandKeyCanBatch, true );
if (doItAll || ASCabKnown (params, kAVCommandKeyGenericTitle))
{
//Create a new text object and insert it into the ASCab
text = ASTextNew();
ASTextSetEncoded (text, kCmdGenericTitle,
(ASHostEncoding)PDGetHostEncoding());
ASCabPutText (params, kAVCommandKeyGenericTitle, text);
}
if (doItAll || ASCabKnown (params, kAVCommandKeyTitle))
{
// Create another text object and insert it into the ASCab
text = ASTextNew();
ASTextSetEncoded (text, kCmdTitle,
(ASHostEncoding)PDGetHostEncoding());
ASCabPutText (params, kAVCommandKeyTitle, text);
```

Note: The `params` object was declared in "Running an AVCommand" on page 135.

File format conversion handlers

A plugin can add file conversion handlers to Acrobat (but not Adobe Reader) for performing the following file conversion operations:

- To import a PDF document from another file format.
- To export a PDF document to another file format.

To add a new file conversion handler, you provide a set of callback functions, specify them in the `AVConversionToPDFHandler` or `AVConversionFromPDFHandler` structures, and invoke the `AVAppRegisterToPDFHandler` or `AVAppRegisterFromPDFHandler` methods to register them.

Specify the file types that the plugin can convert and whether it can perform synchronous conversion (required for the handler to be accessible from the batch framework). Upon registration, the conversion handlers are automatically added to the respective Open and Save As dialog boxes.

By using a file format conversion handler, you can perform the following tasks:

- Provide default settings for the conversion by using the `AVConversionDefaultSettingsProc` callback method.
- Provide conversion parameter information by using the `AVConversionParamDescProc` callback method.
- Display a settings dialog box by using the `AVConversionSettingsDialogProc` callback method.
- Convert a non-PDF file to or from a PDF file by invoking either the `AVConversionConvertToPDFProc` or `AVConversionConvertFromPDFProc` callback methods.

File specification handlers

A file specification handler converts between a `PDFileSpec` object and an `ASPathName` object. Each file specification handler works with a single file system, which the handler specifies.

To create a new file specification handler, a plugin or application must provide callbacks that:

- Convert an `ASPathName` to a `PDFileSpec`. It is called by `PDFileSpecNewFromASPath`.
- Convert a `PDFileSpec` to an `ASPathName`.

Selection servers

A selection server enables the selection of specific data types such as annotations, text, or graphics. You can also create selection servers to enable the selection of data types not already supported. To add a new selection server, you must provide a set of callbacks, specify them in the `AVDocSelectionServer` data structure, and register them using an `AVDocRegisterSelectionServer` object.

By using a selection server, you can perform the following tasks:

- Return the selection type serviced by the handler by using the `AVDocSelectionGetTypeProc` callback method.
- Highlight or unhighlight a selection by using the `AVDocSelectionHighlightSelectionProc` callback method.
- Handle key presses by using the `AVDocSelectionKeyDownProc` callback method.
- Delete the selection by invoking the `AVDocSelectionDeleteProc` method.
- Cut the selection to the clipboard by using the `AVDocSelectionCutProc` callback method.
- Copy the selection to the clipboard by using the `AVDocSelectionCopyProc` callback method.
- Paste the selection from the clipboard by using the `AVDocSelectionPasteProc` callback method.

- Enumerate the items in the current selection by using the `AVDocSelectionEnumSelectionProc` callback method.
- Scroll the view so that the current selection is available by using the `AVDocSelectionShowSelectionProc` callback method.
- Determine whether or not the Properties menu item is enabled by using the `AVDocSelectionCanPropertiesProc` callback method.
- If the selection type has a properties dialog box, display the dialog box by using the `AVDocSelectionPropertiesProc` callback method.

For a complete list of the callbacks in a selection server, see the description of `AVDocSelectionServer` in the Acrobat and PDF Library API Reference.

Note: The SelectionServer sample plugin that is located in the Acrobat SDK shows an example of a selection server.

Tool callbacks

To add a new tool, you must provide a set of callbacks, specify them in the `AVTool` data structure, and register them using `AVAppRegisterTool`. By using tool callbacks, you can perform the following tasks:

- Activate the tool when the tool is selected by using the `ActivateProcType` callback method.
- Deactivate the tool when another tool is selected by using the `DeactivateProcType` callback method.
- Handle mouse clicks by using the `DoClickProcType` callback method.
- Handle key presses by using the `DoKeyDownProcType` callback method.
- Control the cursor shape by using the `AdjustCursorProcType` callback method.
- Return the tool's name by using the `GetTypeProcType` callback method.
- Indicate whether the tool stays active after it is used by using the `IsPersistentProcType` callback method.
- Determine whether the tool is enabled by using the `AVComputeEnabledProc` callback method. For example, if a tool is meant to be used within documents, but there are no documents open, it does not make sense to activate the tool.

Note: For a complete list of callbacks, see the description of `AVTool` in the Acrobat and PDF Library API Reference.

Window handlers

When a plugin creates a window, it can register the window, so that it behaves like other windows in Acrobat; for example, when Adobe Reader or Acrobat is minimized or hidden. For each window that a plugin provides, a window handler must be provided. Window handlers are used only in the Mac OS version of Adobe Reader or Acrobat. Windows versions of Acrobat instead use the platform's native window handling mechanisms. (See "Opening a PDF document in an external window" on page 65.)

To define a window handler, you must provide a set of callbacks, specify them in an `AVWindowHandler` structure, and pass the structure to `AVWindowNew` or `AVWindowNewFromPlatformThing`. The window handler's callbacks are automatically called by Acrobat. Default behavior is used for any missing callbacks.

By using a window handler, you can perform the following tasks:

- Handle mouse clicks in the window by using the `AVWindowMouseDownProc` callback method.
- Handle keystrokes in the window by using the `AVWindowKeyDownProc` callback method.
- Draw the window's contents by using the `AVWindowDrawProc` callback method.
- Permit or prevent closing of the window by using the `AVWindowWillCloseProc` callback method.
- Clean up after the window has been closed by using the `AVWindowDidCloseProc` callback method.
- Do anything that must be done when the window is activated or deactivated by using the following callback methods: `AVWindowDidActivateProc` or `AVWindowWillDeactivateProc`.
- Permit or constrain window size changes by using the `AVWindowWillBeResizedProc` callback method.
- Determine whether the Cut, Copy, Paste, Clear, SelectAll, and Undo menu items are enabled by using the `AVWindowCanPerformEditOpProc` callback method.
- Perform Cut, Copy, Paste, Clear, SelectAll, and Undo operations by using the `AVWindowPerformEditOpProc` callback method.
- Control the shape of the cursor when it is within the window by using the `AVWindowAdjustCursorProc` callback method.

For a complete list of callbacks in a window handler, see the description of `AVWindowHandler` in the Acrobat and PDF Library API Reference.

File systems

Plugins can add new file systems to Acrobat or Adobe Reader, to access files on a device that cannot be accessed as a local hard disk, such as a socket or a modem line.

To add a new file system, you must provide a set of callbacks and specify them in the `ASFileSysRec` structure. This structure is passed as a parameter to calls that require a file system. A file system handler does not require explicit registration.

By using a file system handler, you can perform the following tasks:

- Open a file by using the `ASFileSysOpenProc` callback method.
- Close a file by using the `ASFileSysCloseProc` callback method.
- Flush a file's buffered data to disk by using the `ASFileSysFlushProc` callback method.
- Get or set the current position in a file by using one of the following callback methods: `ASFileSysSetPosProc` or `ASFileSysGetPosProc`.
- Get or set a file's logical size by using one of the following callback methods: `ASFileSysGetEofProc` or `ASFileSysSetEofProc`.
- Read data from a file by using the `ASFileSysReadProc` callback method.
- Write data to a file by using the `ASFileSysWriteProc` callback method.
- Delete a file by using the `ASFileSysRemoveProc` callback method.
- Rename a file by using the `ASFileSysRenameProc` callback method.
- Get a file's name by using the `ASFileSysGetNameProc` callback method.
- Get a file system's name by using the `ASFileSysGetFileSysNameProc` callback method.

- Determine whether two files are the same by using the `ASFileSysIsSameFileProc` callback method.
- Get a path to a temporary file by using the `ASFileSysGetTempPathNameProc` callback method.
- Copy a path (not the underlying file) by using the `ASFileSysCopyPathNameProc` callback method.
- Convert between device-independent and device-dependent path by using the `ASFileSysDiPathFromPathProc` callback method.
- Dispose of a path (not the underlying file) by using the `ASFileSysDisposePathNameProc` callback method.
- Flush data on a volume by using the `ASFileSysFlushVolumeProc` callback method.
- Handle asynchronous I/O operations by using the following callback methods: `ASFileSysAsyncReadProc` or `ASFileSysAsyncWriteProc`.
- Handle multiple read requests by using the `ASFileSysMReadRequestProc` callback method.

For details about each of the callbacks in a file system, see the description of `ASFileSysRec` in the Acrobat and PDF Library API Reference.

Progress monitors

Progress monitors provide feedback to a user on the progress of a time-consuming operation. Some potentially time-consuming methods in the Acrobat core API require a progress monitor as a parameter. Acrobat has a default progress monitor, which generally is sufficient for plugins to use. The built-in progress monitor can be obtained by using the `AVAppGetDocProgressMonitor` method.

Plugins can use the default progress monitor or implement their own by providing a set of callbacks, specifying them in the `ASProgressMonitorRec` data structure, and passing a pointer to the structure to the methods that require a progress monitor (there is no explicit registration method).

Using a progress monitor, you can perform the following tasks:

- Initialize the progress monitor and display it with a current value of zero by invoking the `PMBeginOperationProc` method.
- Draw a full progress monitor, then remove the progress monitor from the display by invoking the `PMEndOperationProc` method.
- Set the value that corresponds to a full progress monitor display by invoking the `PMSetDurationProc` method.
- Set the current value of the progress monitor and update the display by invoking the `PMSetCurrValueProc` method.
- Get the progress monitor's maximum value by invoking the `PMGetDurationProc` method.
- Get the progress monitor's current value by invoking the `PMGetCurrValueProc` method.

For details, see the description of `ASProgressMonitorRec` in the Acrobat and PDF Library API Reference.

Transition handlers

Transitions allow effects such as dissolves or wipe-downs when displaying a new page. New transition types can be added by defining and registering a transition handler.

To add a new transition, you must provide a set of callbacks, specify them in the `AVTransHandler` data structure, and register them by invoking the `AVAppRegisterTransHandler` method.

Using a transition handler, you can perform the following tasks:

- Get the transition type by invoking the `AVTransHandlerGetTypeProc` method.
- Perform the transition (change to the next page with this transition style) by invoking the `AVTransHandlerExecuteProc` method.
- Fill in the transition dictionary in the PDF file by using either the `AVTransHandlerInitTransDictProc` or `AVTransHandlerCompleteTransDictProc` methods.
- Provide information for the user interface that sets the attributes of the transition by invoking the `AVTransHandlerGetUINameProc` method.

Adding message handling

Plugins can add their own DDE messages and Apple events to those supported by Acrobat and Adobe Reader. On Windows, plugins can register to receive DDE messages directly. On Mac OS, plugins must hook into Acrobat or Adobe Reader's Apple event handling loop to handle Apple events. To do this, replace the API's AVAppHandleAppleEvent method. For information about replacing methods, see "Replacing HFT methods" on page 165.

If a plugin receives an Apple event it does not want to handle, it should invoke the implementation of the method it replaced, allowing other plugins or Acrobat or Adobe Reader the opportunity to handle the Apple event.

Note: Plugins should use the DDEML library to handle DDE messages. Problems may arise if they do not.

13 Registering for Event Notifications

This chapter explains how to register for notification of a specific event. The Acrobat core API provides a notification mechanism so that plugins can synchronize their actions with Acrobat or Adobe Reader. Notifications enable plugins to indicate that they are interested in a specified event (such as the initialization of Adobe Reader or Acrobat) and to provide a callback function that is invoked by Adobe Reader or Acrobat each time an event occurs.

The order in which notifications occur may vary depending on the platform. For example, after opening a PDF document on the Windows platform, notifications occur in this order:

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

In contrast, after opening a PDF document in Mac OS, notifications occur in this order:

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

Registering event notifications

Register for an event notification when you want your plugin to be notified when a specific event occurs. For example, you can register for a notification when Acrobat or Adobe Reader is finished initializing. To register for an event notification, you provide a callback function that Acrobat or Adobe Reader invokes when the event occurs. To view a list of notification methods used to register an event notification, see the Acrobat and PDF Library API Reference.

You can register for an event notification by performing the following tasks:

1. Create a user-defined function that is invoked when the event occurs.
2. Invoke the `AVAppRegisterNotification` method and pass the following arguments:
   - The name of the Acrobat core API method that corresponds to the event notification. For example, to register for the event that occurs when Adobe Reader or Acrobat is finished initializing, pass `AVAppDidInitialize`. Append the value `NSEL` to the end of the method name.
   - An `ASExtension` object that represents the identity of the caller. For plugins, you can use `gExtensionID` (this is defined in the PIMain.c file).

- The callback function that is invoked when the event occurs. You can invoke the `ASCallbackCreateNotification` macro and pass the following arguments:
  - The name of the Acrobat core API method that corresponds to the event notification. Do not append a value to the method name.
  - The address of the user-defined function that is invoked when the event occurs.
- A pointer to user-supplied data. Pass `NULL` if you do not want to supply user-supplied data.

The following code example registers for the event that occurs when Adobe Reader or Acrobat is finished initializing. The name of the callback function is `myNotificationCallback`. This function simply displays an alert box. Note that `AVAppRegisterNotification` is invoked within the `PluginInit` method. For information about this method, see "Handshaking" on page 26.

Example: *Registering for an event notification*

```
ACCB1 ASBool ACCB2 PluginInit(void)
{

//Register for an event notification
AVAppRegisterNotification(AVAppDidInitializeNSEL,
gExtensionID,ASCallbackCreateNotification(AVAppDidInitialize,
&myNotificationCallback), NULL);

}

//Create a user-defined function that is invoked when Adobe Reader or Acrobat
//has finished initializing
ACCB1 void ACCB2 myNotificationCallback(void *clientData)
{
AVAlertNote("Acrobat has finished initializing");
}
```

Unregistering event notifications

You can unregister an event notification that you previously registered for by using the Acrobat core API. To unregister an event notification, invoke the `AVAppUnregisterNotification` method and pass the following arguments:

- The name of the Acrobat core API method that corresponds to the event notification. For example, to register for the event that occurs when Adobe Reader or Acrobat has initialized, pass `AVAppDidInitialize`. Append the value `NSEL` to the end of the method name.
- An `ASExtension` object that represents the identity of the caller. For plugins, you can use `gExtensionID` (this is defined in the PIMain.c file).
- The callback function that is invoked when the event occurs. You can invoke the `ASCallbackCreateNotification` macro and pass the following arguments:
  - The name of the Acrobat core API method that corresponds to the event notification. Do not append a value to the method name.
  - The address of the user-defined function that is invoked when the event occurs.
- A pointer to user-supplied data. Pass `NULL` if you do not want to supply user-supplied data.

The following example unregisters the event notification that occurs when Adobe Reader or Acrobat has initialized.

Example: *Unregistering an event notification*

```
AVAppUnregisterNotification(AVAppDidInitializeNSEL,
gExtensionID,ASCallbackCreateNotification(AVAppDidInitialize,
&myNotificationCallback), NULL);
```

Note: Pass the same arguments that you specified when you registered for the event notification. (See "Registering event notifications" on page 143.)

14 Working with Document Security

About document security

Encryption is controlled by an encryption dictionary in the PDF file. The Acrobat core API uses RC4 (a proprietary algorithm provided by RSA Data Security, Inc.) to encrypt document data, and a standard (proprietary) method to encrypt, decrypt, and verify user passwords to determine whether or not a user is authorized to open a document.

Each stream or string object in a PDF file is individually encrypted. This level of encryption improves performance because objects can be individually decrypted as needed rather than decrypting an entire file. All objects, except for the encryption dictionary (which contains the security handler's private data), are encrypted using the RC4 algorithm Adobe licenses from RSA Data Security, Inc. A plugin may not substitute another encryption scheme for RC4.

A plugin that implements a security handler is responsible for encrypting the values it places into the encryption dictionary, and it may use any encryption scheme. If the security handler does not encrypt the values it places into the encryption dictionary, the values are in plain text.

The core API provides two Cos layer methods to encrypt and decrypt data using the RC4 algorithm. These methods are `CosEncryptData` and `CosDecryptData`. (See the Acrobat and PDF Library API Reference.)

Security handlers may use these methods to encrypt data they want to put into the PDF file's encryption dictionary and decrypt data when it is read from the dictionary. Security handlers may instead choose to ignore these methods and use their own encryption algorithms.

About security handlers

Application logic that performs user authorization and sets permissions is known as a security handler. Acrobat has these built-in security handlers: password, Adobe and public key security handler. (See the Acrobat and PDF Library API Reference.)

A security handler supports two passwords:

- A user password that enables a user to open and read a protected document with whatever permissions the owner chose
- An owner password that allows a document's owner to also change the permissions granted to users

You can use the Acrobat core API's built-in security handler or write your own security handlers to perform user authorization (for example, by the presence of a specific hardware key or file, or by reading a magnetic card reader).

Security handlers are responsible for performing the following tasks:

- Setting permissions on a file
- Authorizing access to a file
- Setting up a file's encryption and decryption keys
- Maintaining the encryption dictionary of the PDF file containing the document

Security handlers are used in the following situations:

- A document is opened. The security handler determines whether a user is authorized to open the file and sets up the decryption key that is used to decrypt the PDF file.
- A document is saved. The security handler sets up the encryption key and writes extra security-related information into the PDF file's encryption dictionary.
- A user attempts to change a document's security settings. The security handler determines whether the user is permitted to perform this task.

A document may have zero, one, or two security handlers associated with it. A document has zero security handlers if no security is used on the file. When security is applied to a file, or the user selects a different security handler for a secured file, the newly-chosen security handler is not put in place immediately. Instead this new security handler is simply associated with the document; it is a pending security handler until the document is saved.

The new security handler is not put in place immediately because it is responsible for decrypting the contents of the document's encryption dictionary, and that dictionary is re-encrypted in the correct format for the new security handler only when the document is saved. As a result, a document may have both a current and a new security handler associated with it.

A security handler has two names: one that is placed in each PDF file that is saved by the handler (for example, ADBE_Crypt), and another name that Acrobat can use in any user interface items in which the security handler appears (for example, Acrobat Developer Technologies default encryption). This is similar to the two-name scheme used for menu items: a language-independent name that the application logic can refer to regardless of the user interface language, and another name that appears in the user interface. (See "Adding menu commands to menus" on page 83.)

Adding a security handler

You can add a security handler by performing the following tasks:

- Writing a set of callback routines to perform security-related functions.
- Specifying the callbacks in a `PDCryptHandlerRec` structure.
- Registering the handler by passing the structure to `PDRegisterCryptHandlerEx`.

Security handlers data

The following list describes three types of data used by security handlers:

- Authorization data is the data the security handler needs to determine the user's authorization level for a particular file (for example, not authorized to open the file, authorized to access the file with user permissions, authorized to access the file with owner permissions). Passwords are a common type of authorization data.
- Security data is whatever internal data the security handler uses. It includes security information, internal flag values, seed values, and so on.
- Security information is a subset of the security data. Specifically, it is a collection of flags that contains the information that Acrobat uses to display the current permissions to the user. This information includes permissions and the user's authorization level (user or owner).

Security handler callbacks

A security handler must provide callbacks that performs the following tasks:

- Determines whether a user is authorized to open a particular file and what permissions the user has once the file is open (`PDCryptAuthorizeExProc`).
- Creates and fills an authorization data structure, using whatever user interface is needed to obtain the data. For example, displaying a dialog box into which the user can type a password (`PDCryptGetAuthDataExProc`).
- Creates, fills, and verifies a security data structure (`PDCryptNewSecurityDataProc`).
- Extracts security information from the security data structure (`PDCryptGetSecurityInfoProc`).
- Allows the user to request different security settings, usually by displaying a dialog box. (`PDCryptDisplaySecurityDataProc`)
- Sets up the encryption key used to encrypt the file (`PDCryptNewCryptDataProc`).
- Fills or reads the PDF filefs encryption dictionary (`PDCryptFillEncryptDictProc`).
- Displays the current document's permissions (required with `PDCryptAuthorizeExProc` and `PDCryptGetAuthDataExProc` callbacks).

With Acrobat 5.0 and later, a finer granularity of permissions has been predefined for objects supported by a PDF document. Plugins can invoke the `PDDocPermRequest` method to request whether a particular operation is authorized to be performed on a specified object in a document.

To support the `PDDocPermRequest` method, there are two new callback methods: `PDCryptAuthorizeExProc` and `PDCryptGetAuthDataExProc`. Acrobat 5.0 and later also includes optional security handling for batch operations (actions on one or more files). There are a number of callbacks (indicated by PDCryptBatch...) that a security handler must provide to support batch processing. These callbacks are part of a `PDCryptBatchHandler` structure. The `PDCryptHandlerRec` structure contains a new member `CryptBatchHandler`, which points to this structure.

To support batch processing, a security handler should provide a non-NULL value for `CryptBatchHandler` and implement the batch callbacks. Prior to Acrobat 5.0, the maximum length of the encryption key that Acrobat accepted was 40 bits. Acrobat version 5.0 or later accommodates an encryption key length of 128 bits. These length limitations are imposed to comply with export restrictions.

Acrobat's authorization procedure

Acrobat's built-in authorization procedure works as follows:

1. Acrobat invokes the security handler's authorize callback (which is either `PDCryptAuthorizeExProc`, introduced with Acrobat 5.0, or the older `PDCryptAuthorizeProc`) to determine whether the user is allowed to open the file. It passes NULL authorization data, to handle the case where no authorization data is needed. Acrobat also passes the following values:
   - `PDPermReqObjDoc` and `PDPermReqOprOpen` when invoking `PDCryptAuthorizeExProc`.
   - `pdPermOpen` when calling `PDCryptAuthorizeProc`.
2. If the authorize callback returns `true`, the file is opened. Otherwise, the authorization procedure executes the following steps up to three times, to give the user three chances to enter a password, or whatever authorization the security handler uses.
   - It calls the security handler's get authorization data callback (`PDCryptGetAuthDataExProc` or the older `PDCryptGetAuthDataProc`). This callback should obtain the authorization data using

whatever user interface (for example, a dialog box used to obtain a password) or other means necessary, and then creates and fills the authorization data structure.

- It calls the security handler's authorize callback, passing the authorization data returned by the get authorization data callback. If the authorization succeeds, the authorize callback returns the permissions granted to the user, and the authorization procedure returns.

The authorize callback can access the encrypted PDF document, allowing it to encrypt the authorization data using a mechanism that depends on the document's contents. By doing this, someone who knows a document's password cannot easily find out which other documents use the same password. The authorize callback can return permissions that depend on the password as well as the permissions specified when encryption was set up. This allows, for example, more rights to be granted to someone who knows a document's owner password than to someone who knows the document's user password.

Opening a secured file

The Acrobat core API has several methods for opening files. The `PDDocOpen` (or `PDDocOpenEx`) method is used to open PDF files, even when a plugin calls AV layer methods such as `AVDocOpenFromASFileWithParams`. As a result, the sequence of operations is largely the same regardless of whether the document is being opened from the PD layer or from the AV layer. The difference is that if you call `PDDocOpen` directly, you must pass your own authorization procedure (`PDAuthProc`), while AV layer methods always use Acrobat's built-in authorization procedure.

The authorization procedure must implement the authorization strategy, such as giving the user three chances to enter a password. The `PDAuthProc` is not part of a security handler, but it must call the security handler's methods to authorize the user (for example, to get the password from the user and to check whether or not the password is valid).

Acrobat performs the following steps to open a secured PDF file:

1. Searches for an Encrypt key in the PDF document's trailer, to determine whether or not the document is encrypted. If there is no Encrypt key, Acrobat opens the document immediately.

2. If there is an Encrypt key, its value is an encryption dictionary. Acrobat gets the value of the Filter key in the dictionary to determine which security handler was used when the file was saved. It looks in the list of registered security handlers (which contains Acrobat's built-in handler and any handlers that plugins or applications have registered) for one whose name matches the name found in the PDF file. For information about a dictionary, see "Working with Cos dictionaries" on page 177.

3. If Acrobat finds no match, indicating that the necessary handler could not be found, it does not open the document. If it finds a matching security handler, it invokes that handler's `PDCryptNewSecurityDataProc` callback to extract and decrypt information from the PDF file's encryption dictionary.

4. Acrobat invokes the security handler's authorize callback (`PDCryptAuthorizeExProc`) with NULL authorization data, and with the requested permissions set to `PDPermReqOprOpen` or `pdPermOpen` (requesting that the user be allowed to open the file). This allows support for authorization schemes that do not need authorization data.

5. If authorization succeeds, the handler's authorization callback must return the `PDPermReqStatus` (when the callback is `PDCryptAuthorizeExProc`) or `pdPermOpen` (when the callback is `PDCryptAuthorizeProc`) indicating that the user is permitted to open the file.

6. If authorization fails, the authorization procedure passed in the call to open the `PDDoc` is called.

7. If authorization still fails, the file is not opened.

8. If authorization succeeds, Acrobat calls the security handler's `PDCryptNewCryptDataProc` callback to create the decryption key that is used to decrypt the file. The `PDCryptNewCryptDataProc` callback can construct the decryption key in any way it chooses, but generally performs some calculation based on the contents of the security data structure filled previously by the handler's `PDCryptNewSecurityDataProc` callback.

Saving a secured file

When saving a file, it is important to remember the following information:

- When a user selects document encryption for the first time or has selected a different security handler for an already encrypted file, the newly-selected handler does not take effect until the document is saved.
- To be allowed to save a file, the user must have `PDPermReqOprModify` or either `pdPermEdit` or `pdPermEditNotes` permission.
- In Acrobat 5.0, a save operation forces a complete encrypted copy of the file to be written.

The following information is applicable to when a secured file is saved:

- If the file is being saved in an encrypted form for the first time or if a different security handler is selected, Acrobat calls the new security handler's `PDCryptNewSecurityDataProc` callback. This action creates a new copy of the new security handler's security data structure.
- If the file is saved in an encrypted form for the first time or if a different security handler is selected, Acrobat calls the new security handlerfs `PDCryptUpdateSecurityDataProc` callback. This presents whatever user interface the security handler has for enabling the user to set permissions.
- Acrobat invokes the new security handler's `PDCryptFillEncryptDictProc` callback to encrypt and write into the PDF file's encryption dictionary whatever data the security handler wants to save in the PDF file.
- Acrobat writes out the encrypted file.
- Acrobat sets the new security handler as the document's current security handler.

Setting security for a document

Acrobat calls the new security handler's `PDCryptUpdateSecurityDataProc` callback to present whatever user interface the security handler has for allowing the user to set security, passwords, and so forth.

When security is set, the security handler obtains the permissions and authorization data (such as passwords) to be used for the file. The settings do not take effect until the file is saved, as described in the previous section.

Saving a file with an encryption dictionary

To save a file with a new encryption dictionary, use the following callbacks in the `PDCryptHandlerRec`:

1. `PDCryptNewSecurityDataProc` creates and initializes a security data structure. It is called with `encryptDict` (a Cos object) set either to NULL or to a valid encryption dictionary, in which case the

fields of the encryption dictionary are read and placed into the security data structure. For information about a Cos object, see "Working with Cos Objects" on page 169.

2. `PDCryptUpdateSecurityDataProc` gets the current security data structure by invoking the `PDDocGetNewSecurityData` method. It then makes a copy of the structure with which to work. This new copy is freed if an error or cancel condition is encountered. The user is requested to log in to their PKI infrastructure to access the user's keys and certificates.

   If the security data structure was seeded with information from `encryptDict`, an internal authorize procedure is called. This procedure decrypts and examines the data fields in the security data structure copy that are set to indicate the user's permissions and, possibly, information relating to the document symmetric key.

   A user interface is provided to enable your plugin to specify a list of recipients for the document. If all goes well, the `secDataP` argument to `PDCryptUpdateSecurityDataProc` is sent to the copy of the security data structure, and Acrobat frees the original security data structure.

3. `PDCryptFillEncryptDictProc` writes data from the security data structure into the encryption dictionary. When Acrobat is done with the security data structure, it invokes the `PDCryptFreeSecurityDataProc` method.

Opening an encrypted file

The following callbacks are used when opening an encrypted file:

1. `PDCryptNewSecurityDataProc` is invoked as described in the previous section.

2. `PDCryptAuthorizeExProc` is invoked and returns NULL since the authorization permissions have not been determined. This callback should not present a user interface.

3. The plugin does not use the authorization data structure, but instead only the security data structure. It calls an internal authorization procedure that determines the authorization level of the logged-in user. This authorization procedure is the same procedure as is called by `PDCryptUpdateSecurityDataProc` in the previous section.

4. `PDCryptAuthorizeEx` or `PDCryptAuthorize`. The authorization permissions have now been established (by the call to get the authorization data) and are returned. Acrobat opens the file.

15 Working with Unicode Paths

This chapter explains how to work with Acrobat or Adobe Reader's support of Unicode paths. Using this feature, you can programmatically open and save Unicode-named files and select Unicode-named folders. You can, for example, enable a user to open a Unicode-named file and view the corresponding PDF document in Acrobat or Adobe Reader.

About Unicode paths

The Unicode file path feature takes effect when an end user selects a Unicode-named PDF file to open or save. This feature is also used when a Unicode-named file path is passed as an argument to an Acrobat core API method. However, this feature is in effect only when it is required. That is, when a non-Unicode file system is used, the Unicode path feature is not in effect. As a result, the Unicode file system is separate from the default file system, which is non-Unicode.

You can programmatically use this feature by obtaining a pointer to the Unicode file system `fileSys` argument and then invoking a method that accepts the `fileSys` argument. The Windows Unicode file system can be obtained by either invoking the `ASGetDefaultUnicodeFileSys` method or by invoking the `ASFileGetFileSysByName` method and passing either `ASAtomFromString("Win")`.

Creating Unicode file path application logic

When creating application logic that requires a file system argument (either a Unicode file system or the default file system), do not pass `Null` and avoid invoking the `ASGetDefaultFileSys` method. A file system argument must be provided along with the path name argument.

Never assume that the `ASPathName` argument is a character pointer. Do not typecast any character value to an `ASPathName`, and do not typecast a returned `ASPathName` value to a character pointer. If you are passing an `ASPathName` argument without a file system argument, then ensure that you add the file system argument.

Never assume that path and file names can be stored and passed as character pointers (`char *` values). If you have limited code that passes file names, then change them to an `ASText` value or to something that is capable of storing a full Unicode path. If you have a lot of code that passes character pointer values as file names, then consider changing the internal representation of those character pointer values to UTF-8 encoded file names.

The following table lists Acrobat core API methods that should be replaced by newer methods in order to work with Unicode paths.

Old method	New method
`ASGetDefaultFileSys`	`ASGetDefaultFileSysForPath`
`ASPathFromPlatformPath`	`ASFileSysCreatePathName`
`ASPathFromPlatformPathEx`	`ASFileSysCreatePathName`

Old method	New method
`ASFileSysCreatePathName("Cstring")`	`ASFileSysCreatePathName("ASTextPath")`
`ASFileSysCreatePathName("Folder PathName")`	`ASFileSysCreatePathName("FolderPathName WithASText")`
`ASFileSysCreatePathName("DIPath")`	`ASFileSysCreatePathName("DIPathWithAS Text")`
`ASFileSysGetNameFromPath`	`ASFileSysGetNameFromPathAsASText`
`ASFileSysDisplayStringFromPath`	`ASFileSysDisplayASTextFromPath`
`ASFileSysDIPathFromPath`	`ASFileSysDIPathFromPathEx`
`ASFileSysPathFromDIPath`	`ASFileSysPathFromDIPathEx`

If you have Windows-specific application logic that uses `ASPlatformPathGetCstringPtr` to get the native path name, invoke the `ASFileSysAcquirePlatformPath` method and pass `WinUnicodePath` as the `platformPathType` argument. The `ASPlatformPathGetCstringPtr` method will return an `ASUTF16` path.

If you use any of the following methods `AVAppOpenDialog`, `AVAppSaveDialog`, `AVAppChooseFolderDialog`, `CUIOpenDialog`, `CUISaveDialog`, or `CUIFolderDialog` then ensure that the flag argument passed to these includes the `kAVOpenSaveAllowForeignFileSystems` flag so the Unicode file system can be used. (See "Opening a PDF document in an external window" on page 65.)

Retrieving Unicode path values

You can use the Acrobat core API to retrieve a Unicode path value. The Unicode file system is essentially the same as the classic Windows file system except that its `ASPathName` object supports a few additional calls (through the file system call table) and the implementation uses the wide-char (Unicode) version of the Window's APIs to access the native file system.

You can create an `ASPathName` object by using one of the following methods:

- `ASFileSysCreatePathName`
- `ASFileSysPathFromDIPathEx`

When you invoke either one of these methods, you must create an `ASFileSys` object to use as an argument.

Creating an ASFileSys object

Regardless whether you are working with Unicode paths or non-Unicode paths, you must create an `ASFileSys` object when performing tasks that manipulate files, such as opening a PDF file. An `ASFileSys` object represents the file system in which the file that you are manipulating is located.

To create an `ASFileSys` object, invoke the `ASGetDefaultFileSysForPath` method and specify the following arguments:

- An `ASAtom` object that defines the format of the `pathSpec` argument (second argument). To create an `ASAtom` object, invoke the `ASAtomFromString` method and pass one of the following values:
  - `DIPathWithASText` if the `pathSpec` is a `DIPath` being passed to `ASFileSysPathFromDIPathEx`.
  - `ASTextPath` for Windows
  - `FSRef`, `CFURLRef`, `POSIXPath`, `FSSpec` or `Cstring` for Mac OS
- A void pointer that specifies the location of the file.

On Windows, the `ASGetDefaultFileSysForPath` method checks the specified path values and decides if the classic default file system is used works or if the Unicode file system is used. On Mac OS, the default file system is always returned (because neither has a separate Unicode file system; Mac OS already supports Unicode-named paths).

The following code example creates an `ASFileSys` object as part of the process of opening a PDF file. (See "Opening PDF documents" on page 64.)

Example: *Creating an ASFileSys object*

```
#if NOT_USING_UNICODE
   //Specify the PDF file to open (host encoded names only)
   const char* myPath = "C\:\PurchaseOrder.pdf";
   ASAtom pathType = ASAtomFromString("ASTextPath");
#else
   //Specify the PDF file to open (Unicode)
   const ASUns16* myPath = L"C:\\PurchaseOrder(assumeUnicodeCharacters).pdf";
   ASAtom pathType = ASAtomFromString("ASTextPath");
#endif

//Create an ASText object
ASText titleText = ASTextNew();
ASTextSetPDText(titleText, "This PDF was opened by using the Acrobat SDK");

//Create an ASPathName object
ASFileSys fileSys = ASGetDefaultFileSysForPath(pathType, myPath);
ASPathName pathName = ASFileSysCreatePathName(fileSys, pathType, myPath,
NULL);

//Open the PDF file
AVDoc myDoc = AVDocOpenFromFile(pathName, fileSys, titleText);

//Do some clean up
ASFileSysReleasePath(fileSys, pathName);
ASTextDestroy(titleText);
```

Creating an ASFileSys object that supports Unicode paths

You can invoke the `ASGetDefaultUnicodeFileSys` method to create an `ASFileSys` object that represents a file system that supports Unicode paths. On Windows, this method returns an `ASFileSys` object that uses Unicode paths. On Mac OS, this method returns the value that the `ASGetDefaultFileSys` method returns because the Mac OS default file system already supports Unicode paths.

A Unicode file system can be retrieved by using the `ASFileGetFileSysByName` method if you pass `Win` (or `ASAtomFromString("Win")`) for the `ASAtom` name argument.

As of Acrobat 8, a new `platformPathType` type named `WinUnicodePath` is supported. This is the Unicode version of the `Cstring platformPathType` type. It is used to get the Unicode platform path on Windows.

Note: The classic Windows file system supports both `Cstring` and `WinUnicodePath` in its implementation of the `ASFileSysAcquirePlatformPath` and `ASPlatformPathGetCstringPtr` methods.

The SnippetRunner samples include a shared snippet named OpenUnicodeNamedDocSnip that demonstrates how to open a file with a Unicode (UTF-8) file name. The SnippetRunner samples are available at Acrobat Developer Center.

The following code example retrieves the host encoded platform path on Windows.

Example: *Retrieving a host encoded platform path*

```
char* path = NULL;
ASPlatformPath platformPath = NULL;
ASInt32 result = ASFileSysAcquirePlatformPath(
fileSys, pathName, ASAtomFromString("Cstring"), &platformPath);
if ((result == 0) && (platformPath != NULL)
   path = ASPlatformPathGetCstringPtr(platformPath);
ASFileSysReleasePlatformPath(fileSys, platformPath);
```

In contrast, the following code example retrieves a Unicode platform path on Windows.

Example: *Retrieving a Unicode platform path*

```
ASUTF16* path = NULL;
ASPlatformPath platformPath = NULL;
ASInt32 result = ASFileSysAcquirePlatformPath(
fileSys, pathName, ASAtomFromString("WinUnicodePath"), &platformPath);
if ((result == 0) && (platformPath != NULL)
   path = (ASUTF16*)ASPlatformPathGetCstringPtr(platformPath);
ASFileSysReleasePlatformPath(fileSys, platformPath);
```

Note that the `ASPlatformPathGetCstringPtr` method is still called to get the path string, but that a wide-char string is returned since `WinUnicodePath` was passed to the `ASFileSysAcquirePlatformPath` method.

16 Working with Host Function Tables

A host function table (HFT) is the mechanism through which plugins and PDF Library applications invoke methods in Adobe Reader and Acrobat, as well as other plugins. Acrobat and Adobe Reader have HFTs containing pointers to all Acrobat core API methods. In addition, a plugin may create its own HFT to export its methods to other plugins. This chapter illustrates how to export and import HFTs.

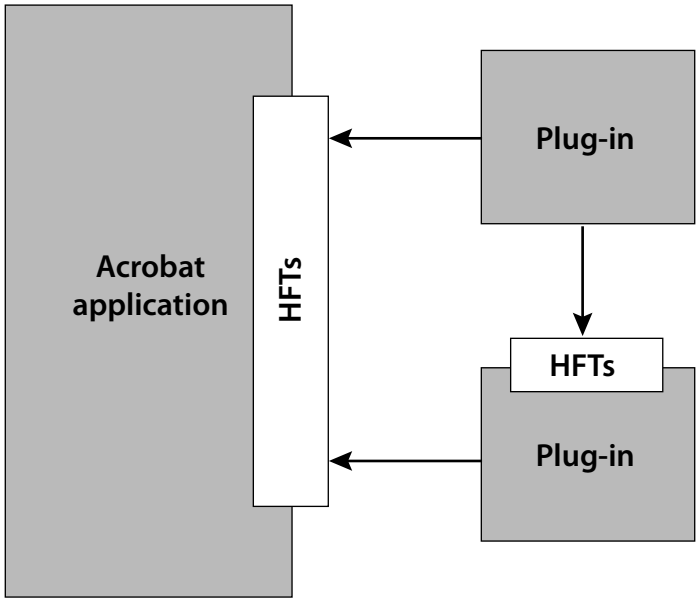
About host function tables

An HFT is a table of function pointers where each HFT contains the following information:

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

Each entry represents a single method that a plugin can invoke, and is defined as a linked list of function pointers. Adobe Reader or Acrobat uses linked lists because some HFT entries may be marked so that they can be replaced by a plugin. Also, it is useful to keep a list of each implementation of a method that was replaced to allow methods to call the implementations they replaced.

The following diagram shows the relationship between Adobe Reader or Acrobat, other plugins, and HFTs.



Plugins must use the `ASExtensionMgrGetHFT` method to import each HFT they intend to use. A plugin requests an HFT by its name and version number. An HFT is imported during plugin initialization. (See "Importing HFTs and registering for notifications" on page 27.)

When a plugin invokes a method in Adobe Reader, Acrobat, or another plugin, the function pointer at the appropriate location in the appropriate HFT is dereferenced and executed. Macros in the Acrobat SDK header files hide this functionality so that plugins contain only what appear to be normal function calls.

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, an HFT server can choose to support multiple versions of the HFT. These versions generally correspond to versions of Acrobat, Adobe Reader or the plugin that exposes the HFT.

The ability to provide more than one version of an HFT improves backward-compatibility by allowing existing plugins to continue to work when new versions of Acrobat or Adobe Reader (or other plugins whose HFTs they use) become available. It is expected that HFT versions typically will differ only in the number, not the order, of methods they contain.

Exporting host function tables

You can use the Acrobat core API to export HFTs that result in a plugin's methods being available to other plugins. To export an HFT, perform the following tasks:

1. Create the HFT methods that you want to make available to other plugins.
2. Create HFT method definitions.
3. Create HFT callback functions.
4. Create new HFTs.

Note: The remaining parts of this section examine each task in detail.

Creating HFT methods

The first step in exporting HFTs is to create the methods that will be exported and made available to other plugins. For the purpose of this discussion, assume that the following three methods exist.

```
ACCB1 void ACCB2 BeepOnceImplementation()
{
   AVSysBeep (0);
   AVAlertNote ("In the BeepOnce method.");
}
ACCB1 void ACCB2 BeepTwiceImplementation()
{
   AVSysBeep (0);
   AVSysBeep (0);
   AVAlertNote ("In the BeepTwice method.");
}
ACCB1 void ACCB2 BeepNTimesImplementation(ASInt32 numtimes)
{
   ASInt32 i;
   for (i=0; i < numtimes; i++)
      AVSysBeep (0);
      AVAlertNote ("In the BeepNTimes method.");
}
```

Creating HFT method definitions

When you invoke a method in an HFT, the methods are accessed through a function pointer. Part of the process of defining a function pointer through which HFT methods are accessed is to create an

enumeration that specifies the index of each method that you want to include within an HFT. The following enumeration enables indexing into the HFT. Note that the first element is not used.

```
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. You can specify the number of indexes in the HFT by defining the following statement:

```
#define NUMSELECTORS (NUMSELECTORSPlusOne - 1);
```

Also declare a global `HFT` object that is used in various tasks:

```
extern HFT gMyHFT
```

For example, to define an HFT method name, you must specify an HFT object. (See "Defining an HFT method name" on page 158.)

Defining function prototypes

After you define an enumeration and an HFT object, you can define a function pointer for each method by using the following syntax:

```
typdef ACCBPROTO1 return_type (ACCBPROTO2
*function_nameSELPROTO)(parameters);
```

The following table describes this syntax.

`return_type`	The return type of the HFT method
`function_name`	The name of the HFT method
`parameters`	The HFT method's parameters with their types

For example, to define a function pointer to the `BeepNTimes` method, specify the following syntax:

```
typedef ACCBPROTO1 void (ACCBPROTO2 *BeepNTimesSELPROTO)(ASInt32 numtimes);
```

`ACCBPROTO1` and `ACCBPROTO2` are macros whose definitions are platform-specific (for example, in Mac OS, `ACCBPROTO1` is defined as `pascal`). `BeepNTimesSELPROTO` specifies a pointer to the `BeepNTimes` method. Without using these macros, you would have to use the following syntax:

```
typedef void (*func)(ASInt32 numtimes);
```

Defining an HFT method name

You must specify a name for each method that is used to invoke the HFT method from other plugins. You can define an HFT method name by using the following syntax:

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

The following table describes this syntax.

`method_name`	The name of the HFT method that is used to invoke the method from external plugins
`HFTname`	The name of the HFT object

For example, to define a method name for the `BeepNTimesImplementation` method, specify the following:

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

This macro defines the symbol `BeepNTimes`, which is the HFT method name. `gMyHFT[BeepNTimesSEL]` is the function pointer obtained by indexing the HFT and `BeepNTimesSELPROTO` casts the pointer to the right type. The end result is that the method can be invoked by specifying the HFT method name:

```
BeepNTimes(3);
```

HFT method names and the implementation method names must be different to avoid conflict between the `#define` statement and the corresponding method name.

Creating HFT callback functions

You must create an HFT callback function in order to successfully export an HFT. It is recommended that you place the application logic to create an HFT callback in the `PluginExportHFTs` method. This is a handshaking method that enables your plugin to export an HFT. For information about handshaking, see "Handshaking" on page 26.

To create an HFT callback function, declare an HFT `ServerProvideHFTProc` object that represents the callback:

```
HFTServerProvideHFTProc provideMyHFTCallback
```

`HFTServerProvideHFTProc` is a callback for an HFT server. After you create an `HFTServerProvideHFTProc` object, you can invoke the `ASCallbackCreateProto` macro to convert a user-defined function to an HFT callback function. For example, you can invoke `ASCallbackCreateProto` to convert a user-defined function named `ProvideMyHFT` to a callback function.

The `ASCallbackCreateProto` macro requires the following arguments:

- The callback type. In this situation, specify `HFTServerProvideHFTProc`.
- The address of the user-defined function that you want to convert to a callback function.

The `ASCallbackCreateProto` macro returns a callback of the specified type that invokes the user-defined function whose address was passed as the second argument. The following lines of code show the `ASCallbackCreateProto` macro converting the `ProvideMyHFT` user-defined function to a `PDWordProc` callback.

```
HFTServerProvideHFTProc provideMyHFTCallback =
ASCallbackCreateProto(HFTServerProvideHFTProc, &ProvideMyHFT);
```

The callback function is invoked when another plugin attempts to use the HFT. After you create an HFT callback function, you can invoke the `HFTServerNew` method to obtain an HFT `Server` object, which is responsible for handling requests to obtain or destroy its HFT. An `HFTServer` object is required in order to create a new `HFT` object.

The `HFTServerNew` method requires the following arguments:

- A character pointer that specifies the name of the HFT server. An HFT server name is used to import the HFT. (See "Importing an existing HFT" on page 164.)
- An `HFTServerProvideHFTProc` object that specifies the HFT callback function.
- An `HFTServerDestroyProc` object that specifies the HFT callback function that releases memory from the HFT. This argument is optional and you can specify `NULL`.
- A pointer to user-supplied data to pass to the HFT server. This argument is optional and you can specify `NULL`.

The following code example creates an HFT callback function within the `PluginExportHFTs` method. After the `HFTServerProvideHFTProc` object is created, the `HFTServerNew` method is invoked which creates an `HFTServer` object.

Example: *Creating an HFT callback function*

```
ACCB1 ASBool ACCB2 PluginExportHFTs(void)
{
gMyHFT = NULL;
gMyHFTServer = NULL;
DURING
   //Create an HFT callback function
   provideMyHFTCallback = ASCallbackCreateProto(HFTServerProvideHFTProc,
   &ProvideMyHFT);

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

Note: In the previous code example, the `gMyHFT`, `gMyHFTServer`, and `gSomethingWentWrong` variables are declared as global variables. To view the complete code example, including the location of where these global variables are declared, see "Examining HFT header and source files" on page 162.

Creating new Host Function Tables

You can create a new HFT by performing the following tasks within the HFT callback function that you define:

1. Create an `HFT` object by invoking the `HFTNew` method. This method requires an `HFTServer` object and the number of entries in the new HFT as arguments. The number of entries determines how many methods that the HFT contains. Each method occupies one entry.

2. Invoke the `HFTReplaceEntry` method to populate the entries in the `HFT` object with pointers to the HFT methods. This method requires the following arguments:

   - An HFT object that you want to populate.

- The entry in the `HFT` object to replace. You can specify an index value that is specified in the enumeration that you created. For example, you can specify `BeepTwiceSEL`. (See "Creating HFT method definitions" on page 157.)
- An `HFTEntry` object that represents a method that will become available through the HFT. You can, for example, reference the `BeepTwiceImplementation` method by passing the `ASCallbackCreateReplacement` method, as shown in the following example:

  ```
  ASCallbackCreateReplacement(BeepTwiceSEL,&BeepTwiceImplementation)
  ```

- The new entry's properties. Currently, only `HFTEntryReplaceable` is defined.

You must invoke the `HFTReplaceEntry` method for each method that you expose through the HFT. For example, if you expose three methods through the HFT, then you invoke the `HFTReplaceEntry` method three times.

The following code example shows the syntax of the `ProvideMyHFT` method, which is the HFT callback function defined in the previous section. Within this method, a new HFT is created. For information about HFT callback methods, see "Creating HFT callback functions" on page 159.

Example: *Creating new Host Table Functions*

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

   DURING

      //Create a new HFT
      gMyHFT = HFTNew(gMyHFTServer, NUMSELECTORS);

      /*
      ** Replace the entries in the HFT
      ** with the methods that you want to make available.
      */
      HFTReplaceEntry (gMyHFT,
BeepOnceSEL,ASCallbackCreateReplacement(BeepOnceSEL,&BeepOnce
Implementation), 0);
      HFTReplaceEntry (gMyHFT,
BeepTwiceSEL,ASCallbackCreateReplacement(BeepTwiceSEL,&BeepTwice
Implementation), 0);
      HFTReplaceEntry (gMyHFT,
BeepNTimesSEL,ASCallbackCreateReplacement(BeepNTimesSEL,&BeepNTimes
Implementation), 0);

   HANDLER
        return NULL;

   END_HANDLER
return gMyHFT;
}
```

Examining HFT header and source files

To make it clear how to create HFTs, this section shows a typical header and source file that is used to create an HFT. All concepts that are discussed up to this point are shown.

Examining an HFT header file

The following code example shows the syntax of a header file named myhft.h that is used to define HFT constructs.

Example: *Examining an HFT header file*

```
#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 */
```

Examining an HFT source file

The following code example shows the syntax of a source file used to create an HFT. Notice that the methods (`BeepOnceImplementation`, `BeepTwiceImplementation`, and `BeepNTimesImplementation`) that the HFT will make available to other plugins are defined. Also notice that the `PluginExportHFTs` method is defined. For information about this method, see "Importing HFTs and registering for notifications" on page 27.

Example: *Examining an HFT source file*

```
#include "corcalls.h"
```

```
#include "avcalls.h"
#include "coscalls.h"
#include "pdcalls.h"
#include "ascalls.h"
#include "myhft.h"

//Declare global variables
HFTServer gMyHFTServer = NULL;
HFT gMyHFT = NULL;
/*
** 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 methods into the table via HFTReplaceEntry
*/
ACCB1 HFT ACCB2 ProvideMyHFT(HFTServer server, ASUns32 version,void *rock)
{
ACCB1 HFT ACCB2 ProvideMyHFT(HFTServer server, ASUns32 version,void *rock)
{
   //Ensure version is 1
   if (version != 1)
      return NULL;

   DURING

      //Create a new HFT
      gMyHFT = HFTNew(gMyHFTServer, NUMSELECTORS);
```

```
     /*
     ** Replace the entries in the HFT
     ** with the methods that you want to make available.
     */
     HFTReplaceEntry (gMyHFT,
BeepOnceSEL,ASCallbackCreateReplacement(BeepOnceSEL,&BeepOnce
Implementation), 0);
     HFTReplaceEntry (gMyHFT,
BeepTwiceSEL,ASCallbackCreateReplacement(BeepTwiceSEL,&BeepTwice
Implementation), 0);
     HFTReplaceEntry (gMyHFT,
BeepNTimesSEL,ASCallbackCreateReplacement(BeepNTimesSEL,&BeepNTimes
Implementation), 0);

   HANDLER
        return NULL;
   END_HANDLER
   return gMyHFT;
}
/*
** 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)
{
gMyHFTServer = NULL;
DURING

   //Create an HFT callback function
   HFTServerProvideHFTProc provideMyHFTCallback =
   ASCallbackCreateProto(HFTServerProvideHFTProc, &ProvideMyHFT);

   //Create an HFT server
   gMyHFTServer = HFTServerNew("MyHFT", provideMyHFTCallback,NULL, NULL);

HANDLER
   return false;
END_HANDLER
return true;
}
```

Importing an existing HFT

You must import an existing HFT to invoke methods that are exposed through the HFT. To import an existing HFT, you must invoke the `ASExtensionMgrGetHFT` method within the `PluginImportReplaceAndRegister` handshaking method. The `ASExtensionMgrGetHFT` method requires the following arguments:

- An `ASAtom` object that specifies the HFT server that corresponds to the HFT to import
- An `ASVersion` object that specifies the version of the HFT

The `ASExtensionMgrGetHFT` method returns an `HFT` object. The following code example shows the `PluginImportReplaceAndRegister` handshaking method that contains application logic that imports the `MyHFT` HFT.

Example: *Importing an existing HFT*

```
ACCB1 ASBool ACCB2 PluginImportReplaceAndRegister(void)
{
   gMyHFT = ASExtensionMgrGetHFT(ASAtomFromString("MyHFT"), 1);
   return (gMyHFT != NULL);
}
```

Note: Both the exporting and importing plugins must be located in Acrobat or Adobe Reader's plugins directory. If the exporting plugin is not located in this directory, the importing plugin cannot successfully import an HFT.

Invoking HFT methods

After you import an HFT, you can invoke a method that it has made available. For example, after you import the `MyHFT` HFT, you can invoke the following methods:

- `BeepOnce`
- `BeepTwice`
- `BeepNTimes`

However, you must include the header file that defines the HFT method name in the source file in which an HFT method is invoked. Because the above methods are declared in a header file named myhft.h, you must specify the following statement to successfully invoke these methods:

```
#include "myhft.h"
```

If you do not include the appropriate header file, you will receive a compile error.

Replacing HFT methods

You can use the Acrobat core API to replace methods that are located in existing HFTs. For example, a plugin could use this mechanism to change the appearance of all alert boxes displayed by Acrobat or Adobe Reader, or to override file opening behavior.

The following table lists all the replaceable Acrobat and Adobe Reader methods.

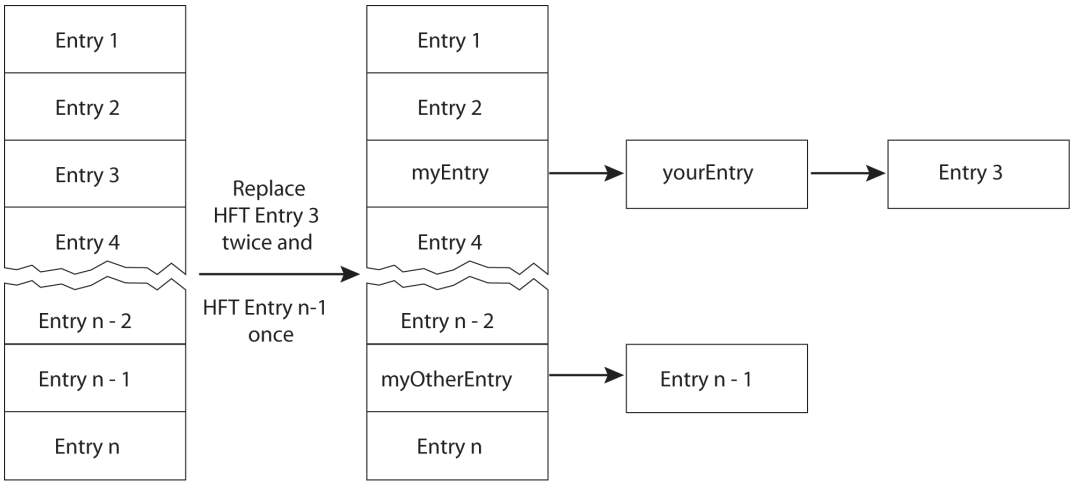
`AVAlert`	`AVAppCanQuit`
`AVAppChooseFolderDialog`	`AVAppHandleAppleEvent`
`AVAppOpenDialog`	`AVAppSaveDialog`
`AVDocClose`	`AVDocDoPrint`
`AVDocDoSave`	`AVDocDoSaveAs` (not replaceable in Adobe Reader)

`AVDocDoSaveAsWithParams` (not replaceable in Adobe Reader)	`AVDocOpenFromASFileWithParams`
`AVDocPrintPages`	`AVDocPrintPagesWithParams`
`AVPageViewGetNextView`	`PDDocSave` (not replaceable in Adobe Reader)
`PDDocSaveWithParams` (not replaceable in Adobe Reader)	

To replace one of these methods, a plugin invokes the `HFTReplaceEntry` method. In some cases, when the replacement method is finished executing, it should invoke 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 invoked automatically; it is up to the replacement implementation to invoke them.

When you replace an Acrobat HFT method, the replaced method is available from other plugins. For example, assume you replace the `AVAlert` method. When other plugins invoke the `AVAlert` method, the replacement version of `AVAlert` is invoked.

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 shown in the following diagram.



To replace an HFT method, perform the following tasks:

- Invoke the `ASCallbackCreateReplacement` macro to create the callback pointer.
- Invoke the `REPLACE` macro to replace the desired method and pass the following arguments:
  - The `HFT` object in which the method is replaced.
  - The entry in the HFT to replace. Append SEL to the method's name. For example, to replace the `AVAppCanQuit` method, specify `AVAppCanQuitSEL`.
  - The address of the replacement method.

The following example shows how to replace the `AVAppCanQuit` method with a custom method named `MyAvAppCanQuit`. The `MyAVAppCanQuit` method's arguments and return value are identical to those of the `AVAppCanQuit` method. Replaceable methods must be replaced with methods that have the same arguments and return type.

The first statement in the following code example initializes a global pointer named `gMyAVAppCanQuitPtr` to your replacement method. You can use this pointer to invoke the original method. For example, you can invoke your replacement method to exhibit custom functionality and then invoke the original method. To invoke the original method, use the `CALL_REPLACED_PROC` macro and pass the pointer to your replacement method. For more information about this macro, see the Acrobat and PDF Library API Reference.

Example: *Replacing an HFT method*

```
void* gMyAVAppCanQuitPtr = NULL;
/*
** A function that informs the application 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);

//Invoke the Replace macro
REPLACE (gMyHFT, AVAppCanQuitSEL, gMyAVAppCanQuitPtr);

HANDLER
   AVAlertNote("Trying to replace AVAppCanQuit");
END_HANDLER
}
```

Note: In the previous code example, an `HFT` object named `gMyHTF` is passed to the `REPLACE` macro. To execute this example, you must create this object. (See "Exporting host function tables" on page 157.)

Migrating non-HFT PDF Library applications to HFT applications

In previous versions of Acrobat, a PDF Library application did not support use of HFTs. However, as of Acrobat 8, PDF Library applications are able to link to the PDF Library DLL file using HFTs. As a result, the PDF Library API is more closely aligned to the Acrobat core API.

You can migrate existing non-HFT PDF Library applications to HFT PDF Library applications by performing the following tasks:

1. Change your project settings from `PRODUCT = \"Library.h\"` to `PRODUCT = \"HFTLibrary.h\"` (the header files include the necessary code to translate from direct calls into calls though HFTs).

2. Add the following files to your PDF Library application: PDFLInitHFT.c and PDFLInitCommon.c.

3. Compile and link your project with the new source files (PDFLInitHFT.c and PDFLInitCommon.c).

4. Invoke the `PDFLInitHFT` method instead of the `PDFLInit` to initialize the HFT mechanism and the PDF Library. The PDFLInitHFT method is defined in PDFLInitHFT.c file and the prototype for this function is defined in PDFInit.h along with the prototype for the `PDFLInit` method. The `PDFLInitHFT` method can be called more than once and a count of the initializations will be maintained by PDF Library.

5. Invoke the `PDFLTermHFT` method instead of the `PDFLTerm` method to shutdown the HFT mechanism and PDF Library. The `PDFLTermHFT` method is defined in the PDFLInitHFT.c file. The prototype for this function is defined in the PDFInit.h file along with the prototype of the `PDFLTerm` method. In case of multiple initializations, the library shuts down after the number of terminations matches the number of initializations.

The following table lists PDF Library API methods that should be changed to newer methods when working with HFTs.

Old method	**New method**
`ASSecs`	`ASGetDefaultFileSysForPath`
`ASPushExceptionFrame`	`ACPushExceptionFrame`
`ASPopExceptionFrame`	`ACPopExceptionFrame`
`ASGetExceptionErrorCode`	`ACGetExceptionErrorCode`

Note: Other PDF Library API methods will work as is without any code change.

17 Working with Cos Objects

A PDF file is structured as a tree of low-level objects, called Cos objects. Cos objects form all PDF document components, such as bookmarks, pages, fonts, images, and annotations. The Acrobat core API contains methods (the Cos layer) that enable you to operate directly on these low-level objects. You may encounter a situation where you want to perform a task that is not supported by using AV and PD layer methods. In such a situation, it is necessary to use Cos methods.

For example, the Creating Annotations chapter explains how to set text annotations properties by using `PDTextAnnot` methods. Some newer types of annotations, such as 3D annotations, have properties that cannot be accessed directly by PD layer methods. As a result, you must use Cos layer methods to access the PDF dictionary that represents the annotation. (See "Creating 3D Annotations" on page 194.)

Caution: Care is required when working with Cos objects. Unlike using AV and PD objects, Cos objects can produce invalid PDF files. Before working with Cos objects, it is strongly recommended that you be familiar with concepts such as resource dictionaries as discussed in the *PDF Reference*.

About Cos objects

PDF files contain various Cos object types. In addition to basic data types such as integer, fixed, and Boolean values, Cos objects also contain the following object types:

- Array
- Dictionary
- Name
- String
- Stream

About direct and indirect objects

You can create Cos objects as either direct or indirect objects; the choice is specified as a parameter to the method that creates the object. 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 labeled so that it can be referenced by other objects multiple times. The following is the syntax of an indirect object.

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

`<object number>` `<generation number>` is known as an indirect object identifier. 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 object 7, you would get the Cos object that represents this string value:

```
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
```

About Cos object types

Two API objects exist in the Cos layer:

- `CosDoc`, which represents an entire PDF file.
- `CosObj`, which represents all the individual object types, such as a Cos string, 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.

Cos strings

A string object consists of a series of bytes—unsigned integer values in the range 0 to 255. The string elements are not integer objects, but are stored in a more compact format. String objects can be written in the following ways:

- As a sequence of literal characters enclosed in parentheses
- As hexadecimal data enclosed in angle brackets

Literal strings

A literal string is written as an arbitrary number of characters enclosed in parentheses. Any characters may appear in a string except unbalanced parentheses and the backslashes, which must be treated specially. Balanced pairs of parentheses within a string require no special treatment.

The following are examples of literal strings:

```
(This is a string)
(Strings may contain newlines
and such)
(Strings may contain balanced parentheses ( ) and
special characters (*!&}^% and so on).)
(The following is an empty string.)
()
(It has zero (0) length.)
```

Within a literal string, the backslash (\) is used as an escape character for various purposes, such as including newline characters, nonprinting ASCII characters, unbalanced parentheses, or the backslash

character itself in the string. The character immediately following the backslash determines its precise interpretation. If the character following the backslash is not one of those shown in the following table, the backslash is ignored. The following table shows valid literal string escape sequences.

Escape sequence	Description
\n	Line feed (LF)
\r	Carriage return (CR)
\t	Horizontal tab (HT)
\b	Backspace (BS)
\f	Form feed (FF)
\(	Left parenthesis
\)	Right parenthesis
\\	Backslash
\ddd	Character code ddd (octal)

If a string is too long to be conveniently placed on a single line, it may be split across multiple lines by using the backslash character at the end of a line to indicate that the string continues on the following line. The backslash and the end-of-line marker following it are not considered part of the string. For example, the following strings examples are equivalent:

```
(These \
two strings \
are the same.)
(These two strings are the same.)
```

Hexadecimal strings

Strings may also be written in hexadecimal form, which is useful for including arbitrary binary data in a PDF file. A hexadecimal string is written as a sequence of hexadecimal digits (0–9 and either A–F or a–f) enclosed within angle brackets (< and >). Consider the following example:

```
<4E6F762073686D6F7A206B6120706F702E >
```

Each pair of hexadecimal digits defines one byte of the string. White-space characters (such as space, tab, carriage return, line feed, and form feed) are ignored. If the final digit of a hexadecimal string is missing, that is, if there is an odd number of digits, the final digit is assumed to be 0. Consider the following example:

```
<901FA3>
```

This is a 3-byte string consisting of the characters whose hexadecimal codes are 90, 1F, and A3, but `<901FA>` is a 3-byte string containing the characters whose hexadecimal codes are 90, 1F, and A0.

Cos arrays

Arrays are one-dimensional collections of objects accessed by a numeric index. Array indexes are zero-based and may be any combination of the Cos data types. The following array has seven elements:

three integers, a string, a Boolean value, 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 ]
```

Cos names

A name object is an atomic symbol uniquely defined by a sequence of characters. Uniquely defined means that any two name objects made up of the same sequence of characters are identically the same object. Atomic means that a name has no internal structure; although it is defined by a sequence of characters, those characters are not considered elements of the name.

```
/AName
```

A slash character (/) introduces a name. The slash is not part of the name but is a prefix indicating that the following sequence of characters constitutes a name. There can be no white-space characters between the slash and the first character in the name. The name may include any regular characters, but not delimiter or white-space characters. Uppercase and lowercase letters are considered distinct: /A and /a are different names. The following examples are valid literal names:

```
/Name1
/ASomewhatLongerName
/A;Name_With-Various***Characters?
/1.2
/$$
/@pattern
/.notdef
```

Beginning with PDF 1.2, any character except null (character code 0) may be included in a name by writing its 2-digit hexadecimal code, preceded by the number sign character (#). This syntax is required to represent any of the delimiter or white-space characters or the number sign character itself; it is recommended but not required for characters whose codes are outside the range 33 (!) to 126 (~). The examples shown in the following table are valid literal names in PDF 1.2 and later.

Literal name	Result
/Adobe#20Green	Adobe Green
/PANTONE#205757#20CV	PANTONE 5757 CV
/paired#28#29parentheses	paired()parentheses
/The_Key_of_F#23_Minor	The_Key_of_F#_Minor
/A#42	AB

The length of a name is subject to an implementation limit. The limit applies to the number of characters in the name's internal representation. For example, the name /A#20B has four characters (/, A, space, B), not six.

Name objects are treated as atomic symbols within a PDF file. Ordinarily, the bytes making up the name are never treated as text to be presented to a user. However, occasionally the need arises to treat a name object as text, such as one that represents a font name.

In such situations, it is recommended that the sequence of bytes (after expansion of # sequences, if any) be interpreted according to UTF-8, a variable-length byte-encoded representation of Unicode in which the

printable ASCII characters have the same representations as in ASCII. This enables a name object to represent text in any natural language, subject to the implementation limit on the length of a name.

Cos dictionaries

A dictionary object is an associative table containing pairs of objects, known as the dictionary's entries. The first element of each entry is the key and the second element is the value. The key must be a name. The value can be any kind of object, including other dictionaries and streams. A dictionary entry whose value is null is equivalent to an absent entry.

A dictionary is a table data structure whose elements are object pairs:

- The first element is the key, which is always a name object, a sequence of characters beginning with the forward slash (/) character. No two entries in the same dictionary should have the same key. If a key does appear more than once, its value is undefined.
- 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.

The following is an example of a dictionary:

```
<< /Name John /Age 27 /AnArray [1 2 3]>>
```

The value associated with the `Name` key is the value `John`. The value for the `Age` key is `27`. And the value for the `AnArray` key is an array with the values `1`, `2`, and `3`. For information about creating a Cos dictionary, see "Creating Cos dictionaries" on page 177.

Cos streams

A stream is a sequence of bytes that can be read a portion at a time. For this reason, objects with potentially large amounts of data, such as images and page descriptions, are represented as streams. A stream consists of a dictionary followed by zero or more bytes bracketed between the keywords `stream` and `endstream`. The following example shows the basic syntax of a stream:

```
dictionary
stream
   ...Zero or more bytes...
endstream
```

The `stream` keyword should be followed by an end-of-line marker consisting of either a carriage return and a line feed or just a line feed, and not by a carriage return alone. The sequence of bytes that make up a stream is located between the `stream` and `endstream` keywords. Streams must be indirect objects and the stream dictionary must be a direct object. (See "About direct and indirect objects" on page 169.)

Note: For more information about streams, see the *PDF Reference*.

Working with Cos strings

This section discusses ways in which you can work with Cos strings. (See "Cos strings" on page 170.)

Creating Cos strings

You can use the Acrobat core API to create a `CosObj` object that is based on a Cos string.

To create a Cos string:

1. Create a `CosDoc` object that represents a PDF file by invoking the `PDDocGetCosDoc` method and passing a `PDDoc` object.
2. Create a `CosObj` object that is based on a Cos string by invoking the `CosNewString` method and passing the following arguments:
   - A `CosDoc` object.
   - An `ASBool` object that specifies whether the `CosObj` object is an indirect or direct object. If `true`, the string is an indirect object. If `false`, the string is a direct object. (See "About direct and indirect objects" on page 169.)
   - A character pointer that specifies the string. Cos strings can contain NULL characters.
   - The length of the character pointer.

The following code example creates a `CosObj` that is based on a Cos string. A `PDDoc` object named `myPDDoc` is passed to the `PDDocGetCosDoc` method. (See "Creating a PDDoc object" on page 76.)

Example: *Creating a Cos string*

```
//Create a new Cos string
char* mystr = "New String";
CosDoc cd = PDDocGetCosDoc(myPDDoc);
CosObj strObj = CosNewString(cd, false, mystr, strlen(mystr));
```

Retrieving the string value

You can retrieve the string value from a `CosObj` that is based on a Cos string. To retrieve the string value, invoke the `CosStringValue` method and pass the following arguments:

- A `CosObj` that is based on a Cos string.
- The address of an `ASTCount` object that is used to store the string length.

An exception is thrown if the `CosObj` object that is passed to the `CosStringValue` method is not based on a Cos string. The following code example expands the previous code example by retrieving the string value by invoking the `PDDocGetCosDoc` method.

Example: *Retrieving the string value from a CosDoc object*

```
//Create a new Cos string
char* mystr = "New String";
CosDoc cd = PDDocGetCosDoc (myPDDoc);
CosObj strObj = CosNewString(cd,false,mystr,strlen(mystr));

//Retrieve the string value
char* strValue;
ASTCount length;
strValue = CosStringValue(strObj, &length);

//Display the string value
AVAlertNote(strValue);
```

Working with Cos arrays

This section discusses ways in which you can work with Cos arrays.

Creating Cos arrays

You can use the Acrobat core API to create a CosObj object that is based on a Cos array. You specify the number of elements that the Cos array stores when you create it. However, you can add elements dynamically as needed. For example, assume that you create a Cos array that stores three elements. If required, you can add a fourth element. An exception is thrown if the CosObj object that is added to the Cos array is a direct object that is already located in another Cos collection object.

To create a Cos array:

1. Create a CosDoc object that represents a PDF file by invoking the PDDocGetCosDoc method and passing a PDDoc object.
2. Create a CosObj object that is based on a Cos array by invoking the CosNewArray method and passing the following arguments:
   - A CosDoc object.
   - An ASBool object that specifies whether the CosObj object is an indirect or direct object. If true, the string is an indirect object. If false, the string is a direct object. (See "About direct and indirect objects" on page 169.)
   - An ASTArraySize object that specifies the number of elements.
3. Create a CosObj object that stores a value to add to the Cos array. For example, to create a CosObj object that is based on an integer value, invoke the CosNewInteger method and pass the following arguments:
   - A CosDoc object.
   - An ASBool object that specifies whether the CosObj object is an indirect or direct object. If true, the string is an indirect object. If false, the string is a direct object. (See "About direct and indirect objects" on page 169.)
   - An ASInt32 value that specifies the integer value.
4. Add the value to the Cos array by invoking the CosArrayPut method and passing the following arguments:
   - A CosObj object that represents a Cos array.
   - An ASTArraySize object that specifies a 0-based index value.
   - A CosObj object that stores the value to add to the array.

The following code example creates a Cos array and adds the values 1, 2, 3, 4, and 5 to it. A PDDoc object named myPDDoc is passed to the PDDocGetCosDoc method. (See "Creating a PDDoc object" on page 76.)

Example: *Creating a Cos array*

```
//Create a new Cos array
CosObj ArrayObj, IntObj;
CosDoc cd = PDDocGetCosDoc(myPDDoc);
ArrayObj = CosNewArray (cd, false, 5);
```

```
for (int i=1; i<=5; 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);
}
```

Retrieving Cos array values

You can use the Acrobat core API to retrieve values from a `CosObj` object that is based on a `Cos` array.

To retrieve values from a Cos array:

1. Determine the number of elements by invoking the `CosArrayLength` method. Pass the `CosObj` object that represents the `Cos` array as an argument.
2. Get the `CosObj` object that represents an array element by invoking the `CosArrayGet` method and passing the following arguments:
   - The `CosObj` object that represents the `Cos` array
   - An `ASTArraySize` object that represents the index of the array element to retrieve

   The `CosArrayGet` method returns a `CosObj` object that represents the element
3. Get the element value. However, you must invoke the method that corresponds to the `CosObj` object's data type. If, for example, the `Cos` array stores integer values, invoke the `CosIntegerValue` method to obtain the element's integer value. Pass the `CosObj` object that represents the element. This method returns the corresponding value. If the `CosIntegerValue` method is invoked, then an `ASInt32` value is returned.

The following code illustrates a user-defined function named `GetArrayValues` that retrieves the value of each element and displays it in an alert box. Notice that a `CosObj` that represents a `Cos` array is passed to the `GetArrayValues` as its only parameter.

Example: *Retrieving Cos array values*

```
void GetArrayValues(CosObj array)
{
   CosObj IntObj;
   ASInt32 value, i, NumElements;
   char buf[256];

   //Determine the number of elements in the array
   NumElements = CosArrayLength(array);

   //Iterate through the array
   for (i=0; i < NumElements; i++)
   {
      //Retrieve a specific element
   IntObj = CosArrayGet(array, i);

      //Convert the CosObj to its ASInt32 value
```

```
   value = CosIntegerValue (IntObj);

      //Display the value
   sprintf(buf, "The element value is %d",value);
   AVAlertNote(buf);
   }
}
```

Working with Cos dictionaries

This section discusses ways in which you can work with Cos dictionaries. (See "Cos dictionaries" on page 173.)

Creating Cos dictionaries

You can create a `CosObj` object that is based on a Cos dictionary. Both the key and its value are `CosObj` objects that you create and add to the Cos dictionary, which is also a `CosObj` object.

To create a Cos dictionary:

1. Create a `CosDoc` object that represents a PDF file by invoking the `PDDocGetCosDoc` method and passing a `PDDoc` object.
2. Create a `CosObj` object that represents the dictionary by invoking the `CosNewDict` method and passing the following arguments:
   - A `CosDoc` object.
   - An `ASBool` object that specifies whether the `CosObj` object is an indirect or direct object. If `true`, the string is an indirect object. If `false`, the string is a direct object. (See "About Cos objects" on page 169.)
   - An `ASTArraySize` object that specifies the number of dictionary entries (the number of key and value pairs).

   The `CosNewDict` method returns a `CosObj` object that represents the new Cos dictionary.
3. Create a `CosObj` object that represents a dictionary value. You must invoke a method that corresponds to the value's data type. For example, to add an integer value, invoke the `CosNewInteger` method and pass the following arguments:
   - A `CosDoc` object.
   - An `ASBool` object that specifies whether the `CosObj` object is an indirect or direct object. If `true`, the string is an indirect object. If `false`, the string is a direct object. (See "About direct and indirect objects" on page 169.)
   - An `ASInt32` value that specifies the integer value.
4. Place the `CosObj` object that represents a dictionary value into the dictionary by invoking the `CosDictPut` method and passing the following arguments:
   - A `CosObj` that represents the dictionary
   - An `ASAtom` object that specifies the key name
   - A `CosObj` object that specifies the dictionary value
5. Repeat steps 3 and 4 for each dictionary entry that you want to add.

The following code example creates a Cos dictionary with the following entries: `/Key1 1 /Key2`. A `PDDoc` object named `myPDDoc` is passed to the `PDDocGetCosDoc` method. (See "Creating a PDDoc object" on page 76.)

Example: *Creating a Cos dictionary*

```
//Create a Cos dictionary
CosObj Dict, IntObj;
CosDoc cd;

//Get the CosDoc
cd = PDDocGetCosDoc(myPDDoc);

//Make a new dictionary with two entries
Dict = CosNewDict (cd, false, 2);
IntObj = CosNewInteger (cd, false, 1);

//Place the key value pair of /Key1 1 into the dictionary
CosDictPut (Dict, ASAtomFromString ("Key1"), IntObj);
IntObj = CosNewInteger (cd, false, 2);

//Place the key value pair of /Key2 2 into the dictionary
CosDictPut (Dict, ASAtomFromString ("Key2"), IntObj);
```

Retrieving values from a Cos dictionary

You can retrieve a dictionary element value by performing the following steps:

1. Get a dictionary key value by invoking the `CosDictGet` method and passing the following arguments:
   - A `CosObj` object that represents the dictionary.
   - An `ASAtom` object that represents the key name.

   The `CosDictGet` method returns a `CosObj` object that represents the dictionary value.
2. Get the element value. However, you must invoke the method that corresponds to the `CosObj` object's data type. If, for example, the `Cos` array stores integer values, invoke the `CosIntegerValue` method to obtain the dictionary entry value. Pass the `CosObj` object that represents the dictionary entry. This method returns the corresponding value. If the `CosIntegerValue` method is invoked, then an `ASInt32` value is returned.

The following code example retrieves the value of a dictionary element whose key is named Key1. The element value is displayed within an alert box.

Example: *Retrieving a value from a Cos dictionary*

```
//Retrieve the value from the dictionary entry whose key is named Key1
CosObj dictEntry;
ASInt32 dicValue;
char buf[256] ;

//Get the element whose key is named Key1
dictEntry = CosDictGet(Dict, ASAtomFromString("Key1"));
dicValue = CosIntegerValue(dictEntry);

//Display the value of the dictionary element
```

```
sprintf(buf,"The value of the dictionary element is %d",dicValue);
AVAlertNote(buf);
```

Note: The `Dict` object is a `CosObj` that represents the dictionary. (See "Creating Cos dictionaries" on page 177.)

Querying a Cos dictionary for a key

You can use the Acrobat core API to determine whether a specific key-value pair exists. To perform this task, invoke the `CosDictKnown` method and pass the following arguments:

- A `CosObj` object that represents the dictionary.
- An `ASAtom` object that represents the key name.

This method returns an `ASBool` value that specifies whether the key-value pair exists. If this method returns `true`, then the key-value pair exists. The following code example queries a dictionary to determine whether a key named `Key1` exists.

Example: *Querying a Cos dictionary for a key*

```
//Determine whether a key named Key1 exists
ASBool keyExist = CosDictKnown(Dict, ASAtomFromString("Key1"));
if (keyExist == true)
   AVAlertNote("The dictionary contains a key named Key1");
else
   AVAlertNote("The dictionary does not contain a key named Key1");
```

Working with Cos names

This section discusses ways in which you can work with Cos names. (See "Cos names" on page 172.)

Creating Cos names

You can use the Acrobat core API to create a `CosObj` object that is based on a Cos name.

To create a Cos name:

1. Create a `CosDoc` object that represents a PDF file by invoking the `PDDocGetCosDoc` method and passing a `PDDoc` object.
2. Create a `CosObj` object that represents the name by invoking the `CosNewName` method and passing the following arguments:
   - A `CosDoc` object.
   - An `ASBool` object that specifies whether the `CosObj` object is an indirect or direct object. If `true`, the string is an indirect object. If `false`, the string is a direct object. (See "About direct and indirect objects" on page 169.)
   - An `ASAtom` object that represent the name to create.

   The `CosNewName` method returns a `CosObj` object that represents the new Cos name.

The following code example creates a Cos name with the value `Name1`. A `PDDoc` object named `myPDDoc` is passed to the `PDDocGetCosDoc` method. (See "Creating a PDDoc object" on page 76.)

Example: *Creating a Cos name*

```
//Create a Cos name
CosObj nameObj;
CosDoc cd = PDDocGetCosDoc(myPDDoc);
nameObj = CosNewName(cd, false, ASAtomFromString("Name1"));
```

Retrieving the value of a name object

You can retrieve the value of a name object by using the Acrobat core API. For example, assume that you retrieve the value from the Cos name object created in the previous code example. In this situation, the value that is retrieved is `Name1`.

To retrieve the value from a Cos name object:

1. Invoke the `CosNameValue` method and pass the `CosObj` that represents the Cos name. This method returns an `ASAtom` object that represents the name value.
2. Invoke the `ASAtomGetString` method to get a constant character pointer that specifies the Cos name value. Pass the `ASAtom` object that is returned from the `CosNameValue` method.

The following code example retrieves the value of a name object.

Example: *Retrieving the value of a name object*

```
//Create a Cos name
CosObj nameObj;
CosDoc cd = PDDocGetCosDoc(myPDDoc);
nameObj = CosNewName (cd, false, ASAtomFromString ("Name1"));

//Get and display the value of a Cos name object
ASAtom nameVal = CosNameValue(nameObj);
const char * str = ASAtomGetString(nameVal);
AVAlertNote(str);
```

Note: The return value of the `ASAtomGetString` method is a constant character pointer, not a character pointer. You will generate a compile error if you omit the `const` keyword.

Working with Cos streams

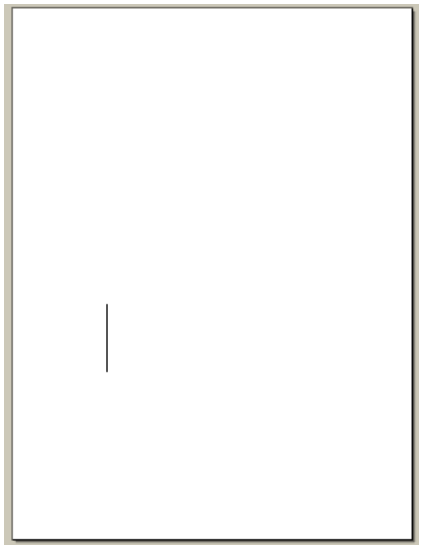
This section discusses ways in which you can work with Cos streams. (See "Cos streams" on page 173.)

A stream is represented by an `ASStm` object definition. A data stream can be a buffer in memory, a file, or an arbitrary user-written procedure. When writing or extracting data streams, an `ASStm` object must be converted to a Cos stream.

Note: Before reading this section, it is strongly recommended that you are familiar with concepts discussed earlier in this chapter, such as `Cos` arrays and Cos dictionaries.

Creating Cos streams

You can create a data stream in memory and then insert the stream into a PDF document page. The following diagram shows the result of a data stream that creates a thin black line segment being inserted into a PDF document.



The following example shows the syntax of a stream that creates a thin line segment:

```
150 250 m 150 350 l S
```

In contrast, the following example shows the syntax of a stream that inserts the text `Hello There` into a PDF document:

```
BT /F0 1 Tf 24 0 0 24 36 756 Tm 0 Tr 0 g 0 Tc 0 Tw \(Hello There) Tj ET
```

Note: For information about stream syntax, see the *PDF Reference*.

Creating a stream dictionary

Each Cos stream has a stream dictionary that contains a `Length` entry that indicates how many bytes are used for the stream's data (if the stream has a filter, the `Length` entry is the number of bytes of encoded data). A limit of 4096 bytes exists for the `Length` entry. A stream dictionary also has optional entries that are not discussed in this section. For more information about stream dictionaries, see the *PDF Reference*.

In addition, most filters are defined so that the data is self-limiting; that is, they use an encoding scheme in which an explicit end-of-data (EOD) marker delimits the extent of the data. Finally, streams are used to represent many objects from whose length attributes can be inferred. All of these constraints must be consistent.

For example, an image with 10 rows and 20 columns, using a single color component and 8 bits per component, requires exactly 200 bytes of image data. If the stream uses a filter, there must be enough bytes of encoded data in the PDF file to produce those 200 bytes. An error occurs if the `Length` entry is too small, if an explicit EOD marker occurs too soon, or if the decoded data does not contain 200 bytes. It is also an error if the stream contains too much data, with the exception that there may be an extra end-of-line marker in the PDF file before the `endstream` keyword. All streams created in this section have a stream dictionary defined.

For each stream that you want to insert into a PDF document, create a stream dictionary that contains at least the `Length` entry, as shown in the following example:

```
<</Length 100>>
```

To create a stream dictionary with the Length entry defined:

1. Create a `CosDoc` object that represents a PDF file by invoking the `PDDocGetCosDoc` method.
2. Create an `ASUns32` object that represents the stream length.
3. Create a `CosObj` object that represents the length of the stream by invoking the `CosNewInteger` method and passing the following arguments:
   - A `CosDoc` object that you created in step 1.
   - An `ASBool` object that specifies whether the `CosObj` object is an indirect or direct object. If `true`, the string is an indirect object. If `false`, the string is a direct object. (See "About direct and indirect objects" on page 169.)
   - The `ASUns32` object created in step 2 that represents the stream length.

   The `CosNewInteger` method returns a `CosObj` object that represents the stream length.
4. Create a `CosObj` object that represents the Cos dictionary. (See "Creating Cos dictionaries" on page 177.)
5. Set the stream dictionary key and value by invoking the `CosDictPutKeyString` method and passing the following arguments:
   - The `CosObj` object that you created in step 4 that represents the dictionary.
   - A character pointer that specifies the name of the key, which in this situation is `Length`.
   - The `CosObj` object created in step 3 that specifies the length of the stream.

The following code example creates a stream dictionary. The first part of this code example creates a `PDPage` object by using an `AVDoc` object. For information about this application logic, see "Creating a PDEContent object" on page 119.

Example: *Creating a stream dictionary*

```
//Create a PDPage object using the current page
AVDoc avDoc = AVAppGetActiveDoc();
AVPageView pageView = AVDocGetPageView(avDoc);
PDPageNumber pageNum = AVPageViewGetPageNum(pageView);
PDDoc pdDoc = AVDocGetPDDoc(avDoc);
PDPage pdPage = PDDocAcquirePage(pdDoc, pageNum);

//Use the PDPage object to get the CosDoc object
CosDoc cd = PDDocGetCosDoc(PDPageGetDoc(pdPage));

//Define a stream to draw a line
char buf [500];
sprintf(buf,"150 250 m 150 350 l S");

//Get the stream length
ASUns32 streamLength = (ASUns32)strlen(buf);

//Create a CosObj object that represents the stream length
```

```
CosObj LengthEntry = CosNewInteger(cd, false,streamLength);

//Create a CosObj that represents a stream dictionary
//and sets its key-value pairs
CosObj AttrDict = CosNewDict(cd, false, 5);
char *Length_KStr = "Length";
CosDictPutKeyString(AttrDict, Length_KStr, LengthEntry);

//Determine if the stream dictionary is valid
if (CosObjEqual (AttrDict, CosNewNull ()) == true)
{
   AVAlertNote ("The attributes dictionary could not be created.");
   return;
}
```

Note: This code example creates a `CosObj` object named `AttrDict` that represents a stream dictionary.

Inserting a Cos stream into a PDF document

When inserting a stream into a PDF document, ensure that the stream is valid; otherwise, a run-time error occurs. This section explains how to create a stream that draws a thin black line segment and then inserts the stream into the current PDF page.

To insert a stream into the current PDF document page:

1. Create a `PDPage` object that represents the current PDF page. (See "Creating a PDEContent object" on page 119.)
2. Create a `CosDoc` object that represents a PDF file by invoking the `PDDocGetCosDoc` method.
3. Define the stream that draws a thin black line segment. You can populate a character array with a stream by invoking the `sprintf` method.
4. Create an `ASUns32` object that represents the stream length.
5. Create a `CosObj` object that represents the stream dictionary. (See "Creating a stream dictionary" on page 181.)
6. Read the stream into memory by invoking the `ASMemStmRdOpen` method and passing the following arguments:
   - A character pointer that contains the data stream
   - An `ASUns32` object that specifies the stream length

   This method returns an `ASStm` object that represents an in-memory data stream.
7. Create a new Cos stream that is based on data located in the `ASStm` object by invoking the `CosNewStream` method and passing the following arguments:
   - A `CosDoc` object that specifies the PDF document in which the Cos stream is inserted (pass the `CosDoc` object created in step 2).
   - An `ASBool` object that specifies whether the Cos stream is an indirect object. Because all streams are indirect objects, this argument must be set to `true`.
   - An `ASStm` object that contains the stream data (pass the `ASStm` object created in step 6).

- A `CosStreamStartAndCode` object that specifies the byte offset from which data reading starts. You can pass `0` to ensure that data reading starts at the beginning of the stream.
- An `ASBool` object that specifies whether the data is encoded using filters specified in the stream dictionary before it is written to the Cos stream.
- A `CosObj` object that represents the stream dictionary (pass the `CosObj` object created in step 5).
- A `CosObj` object that represents the parameters that are used by the encoding filter if the source data is encoded before it is written to the file. If encoding parameters are not required, this value is ignored. For information about encoding filters, see the *PDF Reference*.
- A `CosByteMax` object that specifies the amount of data read from the source. If this value is `-1`, data is read from the source until it reaches the end of the stream.

The `CosNewStream` method returns a `CosObj` object that represents the Cos stream.

8. Replace the contents of the specified page with the Cos stream by invoking the `PDPageAddCosContents` method and passing the following arguments:
   - A `PDPage` object that represents the current page of the PDF document (pass the `PDPage` object created in step 1).
   - A `CosObj` object that contains the Cos stream.

The following code example creates a Cos stream and inserts it into the current page of a PDF document.

Example: *Inserting a Cos stream into a PDF document page*

```
//Declare local variables used in this code example
CosDoc cd;
CosObj PageStrm, LengthEntry, AttrDict;
CosObj EncodeParms = CosNewNull();
ASStm OpenedStream;
char buf [500];

//Create a PDPage object using the current page
AVDoc avDoc = AVAppGetActiveDoc();
AVPageView pageView = AVDocGetPageView(avDoc);
PDPageNumber pageNum = AVPageViewGetPageNum(pageView);
PDDoc pdDoc = AVDocGetPDDoc(avDoc);
PDPage pdPage = PDDocAcquirePage(pdDoc, pageNum); // acquire current page

//Use the PDPage object to create a CosDoc object
cd = PDDocGetCosDoc(PDPageGetDoc(pdPage));

//Define a stream that creates a thin line segment
sprintf(buf,"150 250 m 150 350 l S");

//Get the stream length
ASUns32 streamLength = (ASUns32)strlen(buf);

//Create a CosObj object that represents the stream length
LengthEntry = CosNewInteger(cd, false,streamLength);

//Create a CosObj that represents a stream dictionary
//and set it key-value pairs
AttrDict = CosNewDict(cd, false,5);
char *Length_KStr = "Length";
```

```
CosDictPutKeyString(AttrDict, Length_KStr, LengthEntry);

//Determine if the stream dictionary is valid
if (CosObjEqual (AttrDict, CosNewNull ()) == true)
{
   AVAlertNote ("The stream dictionary could not be created");
   return;
}
//Read the stream into memory by invoking the ASMemStmRdOpen method
OpenedStream = ASMemStmRdOpen(buf,streamLength);

DURING

   //Create a new Cos stream using data from the ASStm object
   PageStrm = CosNewStream(cd, true, OpenedStream, 0,
    false, // StmDataIsNotDecoded
    AttrDict, //The stream dictionary
    EncodeParms, -1);

//Close the stream
ASStmClose(OpenedStream);

HANDLER
AVAlertNote ("Trying to create new CosStream");
CosObjDestroy (AttrDict);
ASStmClose (OpenedStream);
return;
END_HANDLER

//Completely replace the contents of the specified page with newContents
PDPageAddCosContents (pdPage, PageStrm);
```

Caution: If you execute this code example without having a PDF document open, you will cause an Adobe Reader or Acrobat run-time error. The run-time error occurs because this code example creates a `PDPage` object that is based on the current PDF document page.

Populating a PDF document with a content stream

This section explains how to use the Acrobat core API to create a new PDF document, insert a page into the document, and populate the page with a Cos content stream that inserts the text `Hello There`. When inserting a content stream into a PDF document, in addition to creating a stream dictionary, you must also create a resource dictionary and a page dictionary. A resource dictionary defines attributes such as the font that a content stream uses and a page dictionary defines attributes such as the page's height and width. For information about these dictionaries, see the *PDF Reference*.

The following example shows the resource dictionary that is created in this section.

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

The following example shows the font descriptor that is created in this section.

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

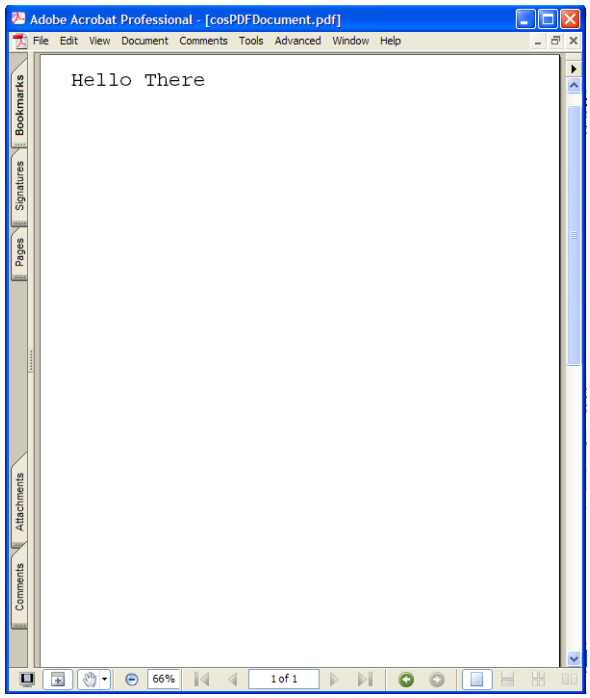
The following example shows the Procset resource created in this section.

```
This is a procset resource.
6 0 obj
[
/PDF /Text
]
endobj
```

The following example shows the page dictionary that is created in this section.

```
This is the page dictionary.
7 0 obj
<<
/Type /Page
/MediaBox [ 0 0 612 792 ]
/Parent 2 0 R
/Resources 4 0 R
/Contents 8 0 R
>>
endobj
```

The following diagram shows the PDF document that is created in this section.



To create a PDF document and populate it with a Cos content stream:

1. Define the media box rectangle used in the PDF document's page.

   ```
   ASFixedRect MedBox;
   MedBox.left = ASInt32ToFixed (0);
   MedBox.top = ASInt32ToFixed (792);
   MedBox.right = ASInt32ToFixed (612);
   MedBox.bottom = ASInt32ToFixed (0);
   ```

2. Define the stream that is written to the PDF document page, as shown in the following example.

   ```
   char* StreamBuf = (char*)"BT /F0 1 Tf 24 0 0 24 36 756 Tm 0 Tr 0 g 0 Tc 0 Tw \
   (Hello There) Tj ET";
   ```

3. Create a `PDDoc` object that represents the new document by invoking the `PDDocCreate` method. After the document is created, at least one page must be added before Acrobat or Adobe Reader can display the document.

   ```
   PDDoc NewDoc = PDDocCreate ();
   ```

4. Create a `PDPage` object that represents the page by invoking the `PDDocCreatePage` method and passing the following arguments:
   - The `PDDoc` object that you created.
   - The `PDBeforeFirstPage` enum value that specifies where to place the page.
   - The `ASFixedRect` object that defines the media box rectangle.

   This method returns a `PDPage` object that represents the new page.

5. Create a `CosObj` object that represents a resource dictionary. In the following code example, a resource dictionary is created in a user-defined function named `SetResourceForPage`.

6. Set the page's resource key. In the following code example, the page's resource's key is set in a user-defined function named `CreateResourceDicts`.

7. Add a Cos stream to the page. In the following code example, a Cos stream is added to the page in a user-defined function named `AddStreamToPage`.

8. Open the PDF document in Adobe Reader or Acrobat. In the following code example, this task occurs in the user-defined function named `MakeTheFile`.

9. Save the PDF document. In the following code example, this task occurs in the user-defined function named `MakeTheFile`.

The following code example represents an entire C source file that creates a PDF document and populates it with a Cos content stream. This source file is made up by various user-defined functions. To make it easier to view these functions, all function signatures are bolded. The entry point to this source file is the `MakeTheFile` function. You can invoke the `MakeTheFile` function from a menu item or toolbar button to execute this code example.

Example: *Creating a PDF document and populating it with a Cos content stream*

```
#include "ascalls.h"
#include "avcalls.h"
#include "avcalls.h"
#include "coscalls.h"
#include "pdcalls.h"
```

```
#include "ascalls.h"
#include "corcalls.h"
#include "dos.h"
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


//Declare global variables
CosObj PageStrm; /* To hold the newly created Cos stream */
CosObj AttrDict; /* As returned by CreateAttribsDict */
CosObj EncodeDict;
/*
** Used to specify what filters are used to encode the stream if
** used for output.
*/
CosObj ResDict; /* Resource dictionary for the page */
CosObj FontDictObj;
CosObj FontDict;
CosObj procArray;

/*
** Set the page's resource key. Return true if everything is valid,
** else false.
*/
ASBool SetResourceForPage (PDPage page)
{
CosObj PageCosObj;

//Make sure that the page is valid
if (!page)
   return false;

DURING
   //Get a CosDoc object by using the PDPage page passed to this object
   PageCosObj = PDPageGetCosObj(page);

   if (CosDictKnown (PageCosObj, ASAtomFromString ("Resources") == true))
      CosObjDestroy (CosDictGet (PageCosObj, ASAtomFromString ("Resources")));

   //Place the ResDict object into the page's Resource Dictionary
   CosDictPut (PageCosObj, ASAtomFromString ("Resources"), ResDict);
HANDLER
   return false;
END_HANDLER
return true;
} /* end of SetResourceForPage*/

/*
** Create the font's resources. Return true if all is valid, else false.
** Creates the required font and proc set dictionaries; then creates the
** global resource dictionary for the PDF page
*/
ASBool CreateResourceDicts (CosDoc cd)
```

```
{
DURING
   ResDict = CosNewDict (cd, true, 10);

HANDLER
   AVAlertNote("Trying to create the resource dictionary.");
   return false;
   END_HANDLER

DURING
   FontDictObj = CosNewDict (cd, true, 5);

HANDLER
   AVAlertNote ("Trying to create the font's dictionary.");
   CosObjDestroy (ResDict);
   return false;
END_HANDLER

/* Create this font descriptor dictionary.
** <<
** /Type /Font
** /Subtype /Type1
** /Name /F0
** /BaseFont /Courier
** /Encoding /WinAnsiEncoding
** >>
*/
DURING
   CosDictPut (FontDictObj, ASAtomFromString ("Type"),
   CosNewName (cd, false, ASAtomFromString ("Font")));
   CosDictPut (FontDictObj, ASAtomFromString ("Subtype"),
   CosNewName (cd, false, ASAtomFromString ("Type1")));
   CosDictPut (FontDictObj, ASAtomFromString ("Name"),
   CosNewName (cd, false, ASAtomFromString ("F0")));
   CosDictPut (FontDictObj, ASAtomFromString ("BaseFont"),
   CosNewName (cd,false, ASAtomFromString ("Courier")));
   CosDictPut (FontDictObj, ASAtomFromString ("Encoding"),
   CosNewName (cd,false, ASAtomFromString ("WinAnsiEncoding")));
HANDLER
   AVAlertNote("Trying to add key-value pairs to the Font descriptor
dictionary.");
   CosObjDestroy (FontDictObj);
   return false;
END_HANDLER
DURING
   FontDict = CosNewDict (cd, false, 2);
HANDLER
   AVAlertNote ("Trying to create page's resource dictionary.");
   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, ASAtomFromString ("F0"), FontDictObj);
   CosDictPut (ResDict, ASAtomFromString ("Font"), FontDict);
HANDLER
   AVAlertNote  ("Trying to add key-value pairs to the page's resource dict.");
   CosObjDestroy (FontDictObj);
   CosObjDestroy (FontDict);
   CosObjDestroy (ResDict);
   return false;
END_HANDLER
/* Create the following proc set resource array.
** [
** /PDF /Text
** ]
*/
DURING
   procArray = CosNewArray (cd, true, 5);
HANDLER
   AVAlertNote  ("Trying to create proc set array.");
   CosObjDestroy (FontDictObj);
   CosObjDestroy (FontDict);
   CosObjDestroy (ResDict);
   return false;
END_HANDLER

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

// Create and return the stream's dictionary that defines the Length attribute
CosObj CreateAttribsDict(CosDoc Doc, ASInt32 Len)
{
//Declare local variables
CosObj Dict; /* Holds newly created dictionary */
ASAtom Key; /* Key used to retrieve CosObj in dictionary */
CosObj Value; /* Assigned, then added to dictionary */
CosObj DecodeArray;

//Create the stream dictionary
```

```
Dict = CosNewDict(Doc, false, 10);
Key = ASAtomFromString("Length");
Value = CosNewInteger(Doc, false, Len);
CosDictPut(Dict, Key, Value);
return Dict;
}

//Add stream to page. Return false if there are problems
ASBool AddStreamToPage (PDPage page, char* StreamBuf, ASInt32 StreamBufLen)
{
//Declare local variables
CosDoc cd;
ASStm Stm=NULL;
CosObj PageStrm;
CosObj EncodeParms = CosNewNull();

DURING
   //Create the CosDoc object
   cd = PDDocGetCosDoc (PDPageGetDoc (page));

HANDLER
   AVAlertNote("Unable to get CosDoc");
   return false;
END_HANDLER

//Retrieve the Attributes dictionary
AttrDict = CreateAttribsDict (cd, StreamBufLen);

if (CosObjEqual (AttrDict, CosNewNull ()) == true){
   AVAlertNote("Not making stream. Attribs dict not created.");
   return false;
}
//Read the stream into memory by invoking the ASMemStmRdOpen method
Stm = ASMemStmRdOpen (StreamBuf, StreamBufLen);
if (!Stm){
   AVAlertNote ("Unable to open data stream to create content stream.");
   return false;
}
DURING
   //Creates a new Cos stream using data from the ASStm object
   PageStrm = CosNewStream(cd, true, Stm, -1,
   false,
   AttrDict, /* attributesDict */
   EncodeParms,
   -1);

ASStmClose (Stm);

HANDLER
   AVAlertNote ("Trying to create new CosStream.");
   CosObjDestroy (AttrDict);
   ASStmClose (Stm);
   return false;
END_HANDLER
```

```
//Add the content stream to the page
PDPageAddCosContents (page, PageStrm);
return true;
} /* end of AddStreamToPage */

//Create the new PDF document
void MakeTheFile (void)
{
//Declare local variables
volatile PDDoc NewDoc = NULL;
volatile PDPage NewPage;
ASFixedRect MedBox;
ASInt32 PageCount = 0;
char* StreamBuf = NULL;
int StreamBufLen = 0;
ASBool result ;

//Set up the page's media box.
MedBox.left = ASInt32ToFixed (0);
MedBox.top = ASInt32ToFixed (792);
MedBox.right = ASInt32ToFixed (612);
MedBox.bottom = ASInt32ToFixed (0);

//Define a stream to set the text matrix and write out the text
StreamBuf = (char*)"BT /F0 1 Tf 24 0 0 24 36 756 Tm 0 Tr 0 g 0 Tc 0 Tw \(Hello
There) Tj ET";

//Get the length of StreamBufLen - this is where is it determined
StreamBufLen = strlen (StreamBuf);

DURING
   //Create a PDDoc object
   NewDoc = PDDocCreate();
   if (NewDoc){
      //Invoke the PDDocCreatePage method
      NewPage = PDDocCreatePage(NewDoc, PDBeforeFirstPage, MedBox);
      if (!NewPage)
         ASRaise (0);

      //Invoke CreateResourceDicts
      if (CreateResourceDicts(PDDocGetCosDoc(NewDoc)) == false)
         ASRaise (0);

   //Invoke SetResourceForPage method
   result = SetResourceForPage(NewPage);

   // Invoke AddStreamToPage
   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

   //Open the new doc
   AVDocOpenFromPDDoc(NewDoc, NULL);

   //Save the PDF document to the root of C and name it cosPDFDocument.pdf
   PDDocSave(NewDoc, PDSaveFull | PDSaveLinearized ,ASPathFromPlatformPath
("C:\\cosPDFDocument.pdf"), NULL, NULL, NULL);

HANDLER
   AVAlertNote ("Cannot open new document.");
END_HANDLER
}
```

18 Creating 3D Annotations

Three-dimensional annotations can be viewed in a PDF document by means of *3D annotations*. This chapter explains how to create 3D annotations and modify existing ones by using Cos objects. Before you read this chapter, it is strongly recommended that you become familiar with Cos objects. (See "Working with Cos Objects" on page 169.)

The underlying 3D object data in a 3D annotation must conform to the Adobe PRC Format Specification the Universal 3D (U3D) format developed by the 3D Industry Forum (http://www.3dif.org). Acrobat supports a subset of U3D, as described in the document *U3D Elements*, which can be found at the Acrobat Developer Center.

Creating annotations

Before you can add three-dimensional data to an annotation, you must create it. Annotations are represented by an annotation dictionary. A *dictionary* is a data structure with one or more entries, which are *key-value pairs:*

- The key is a *name object*.
- The value is some type of PDF object. The *PDF Reference* describes all the PDF object types. If the value is a dictionary, that dictionary has its own key-value pairs. Therefore, dictionaries can be nested within other dictionaries (as you will see with the 3D structures). For details, see the *PDF Reference*.

General annotation dictionary entries are described in Table 8.15 in the *PDF Reference*. Those specific to 3D annotations are described in Table 9.33.

The following code example creates a 3D annotation with corners (1, 9.5) and (7,4) using the `PDPageAddNewAnnot` method:

```
ASFixedRect annotRect;
annotRect.left   = Int16ToFixed(1*72);
annotRect.top    = Int16ToFixed(9.5*72);
annotRect.right  = Int16ToFixed(7.0*72);
annotRect.bottom = Int16ToFixed(4*72);

PDAnnot newAnnot = PDPageAddNewAnnot(pdPage, -2, ASAtomFromString("3D"),
&annotRect);
```

This method contains the following parameters:

- The `PDPage` on which to place the annotation.
- An `ASInt32` indicating where to add the annotation in the page's annotation array. The value -2 is normally used and means the annotation is added to the end of the page's annotation array. For information about this value, see the Acrobat and PDF Library API Reference.
- The type of the annotation, which in this case is `3D`. This sets the value of the annotation dictionary's `Subtype` entry. It is important to note that in PDF this is a *name object*, not a string. In the API, the `ASAtom` type is frequently used to represent names; the `ASAtomFromString` method converts a string to a name.

- The rectangle in which the annotation appears on the page. This sets the value of the annotation dictionary's `Rect` entry.

After the annotation has been created, you must complete the other entries in the annotation dictionary. The `F` (flags) entry is set here:

```
PDAnnotSetFlags(newAnnot, pdAnnotPrint | pdAnnotReadOnly);
```

The annotation's initial appearance (the `AP` entry) can be set. (See "Setting the annotation appearance" on page 201.)

Other entries are set using the `EmbedDataIn3Dannot` function (See Adding 3D data to an annotation.)

If you add PRC data to a 3D annotation, you must also ensure that the document's `Catalog` dictionary contains an Adobe `Extension` dictionary that specifies the PDF base version (`1.7`) and extension level (`1` or greater) that added PRC data to 3D annotations. (See PDF version extensions.)

Adding 3D data to an annotation

The Acrobat and PDF Library API does not contain methods for working specifically with 3D annotations. To add entries to a 3D annotation dictionary, you must use Cos-level API methods. These methods enable you to create PDF objects of any type (see the *PDF Reference*) and to set their values.

The Cos API uses two primary object types:

- `CosDoc` represents an entire PDF document.
- `CosObj` represents all other PDF objects: simple types like numbers and string values as well as complex types such as dictionaries, arrays, and streams.

The API provides methods for creating and working with different object types. The creation methods are of the form `CosNew`*`Object`*, where *`Object`* is the object being created: for example, `CosNewDict`, which creates a dictionary object, `CosNewBoolean`, and `CosNewStream`.

The first two parameters for the `CosNew`*`Object`* methods (with the exception of `CosNewNull`, which takes no parameters) are these:

- The `CosDoc` object that represents the current document.
- A Boolean parameter that specifies whether the new object should be indirect. Indirect objects (see the *PDF Reference*) are given object numbers and can be referred to from more than one place in a PDF file. Direct objects are specified only where they are used and cannot be referred to from anywhere else.

The remaining parameters vary depending on the object type:

- For simple types, the third parameter specifies its value. For arrays and dictionaries, the third parameter specifies the number of elements.
- For streams, there are several additional parameters. See the Acrobat and PDF Library API Reference for details on each method.

The return value of these methods is the newly created object. In the API, this object is always a `CosObj`. That is, even though you call `CosNewDict` to create a dictionary object, and you use methods like `CosDictPut` to work with dictionary objects, there is not a formal object type called `CosDict`.

If necessary, you can determine the type of a `CosObj` by calling the `CosObjGetType` method, which returns a constant (`CosNull`, `CosInteger`, `CosFixed`, `CosReal`, `CosBoolean`, `CosName`, `CosString`, `CosDict`, `CosArray`, or `CosStream`).

Most of the code described here involves setting the entries of dictionaries (`CosObjs` of type `CosDict`). Dictionaries contain a number of key-value pairs, where the key is a *name object* and the value is any type of `CosObj`.

There are several common methods that can be used, which differ only in how the keys are specified. The `CosDictPutKeyString` method (available in Acrobat 7.0 or later) allows the key to be specified as a string and is the recommended method, as in the following example:

```
CosDictPutKeyString(theDict,    // The dictionary
   "TheKey",                   // The key: a string
   theCosValue);               // The value: a CosObj
```

`CosDictPut` requires the key to be specified as an `ASAtom`. `CosDictPutKey` requires the key to be a name object (a `CosObj` of type `CosName`).

Creating the 3D annotation dictionary entries

To work with annotations at the Cos level, you must get the Cos object corresponding to the `PDAnnot` object. To perform this task, invoke the `PDAnnotGetCotObj` method and pass the `PDAnnot` object, as shown in the following example:

```
CosObj cosAnnot = PDAnnotGetCosObj(theAnnot);
```

Next, get the `CosDoc` object corresponding to the document by invoking the `CosObjGetDoc` method and passing the `CosObj` object, as shown in the following example:

```
CosDoc cosDoc = CosObjGetDoc(cosAnnot);
```

Two additional dictionary entries (which are not specific to 3D annotations)—the `P` (page) and `Contents` entries—can be set as follows:

```
CosDictPutKeyString(cosAnnot, "P", PDPageGetCosObj(pdPage));
CosDictPutKeyString(cosAnnot, "Contents",
CosNewString(cosDoc, false, "3D Model", strlen("3D Model")));
```

The following sections show how to set the dictionary entries that are specific to 3D annotations: (See Table 9.33 in the *PDF Reference*.)

`3DD`: A 3D stream specifying the 3D content (See "Specifying the 3D stream" on page 196.)

`3DV`: The initial view of the 3D content (See "Setting the default view" on page 200.)

`3DA`: The activation dictionary (See "Setting the activation dictionary" on page 204.)

Specifying the 3D stream

The `3DD` entry of the annotation dictionary specifies a *stream* containing the PRC or U3D data. Streams are PDF objects that can be thought of as having two parts, the stream data and an associated dictionary:

- Stream data is the PRC or U3D data that represents the 3D content.
- The associated dictionary (sometimes called the *attributes dictionary*) contains entries that specify information about the stream. Some entries are common to all stream dictionaries (see Table 3.4 in the *PDF Reference*). They include:
  - **`Length`** (required): The length of the stream data

- **`Filter`** (optional): A compression filter that is applied to the data to reduce its size (there are also filters that do not compress data)

Other entries are unique to 3D streams (see Table 9.35 in the *PDF Reference*). They include:

- **`Type`** (optional): Must be `3D` if present.
- **`Subtype`** (required): For PRC data, set this value to `PRC`; for U3D data, set this value to `U3D`.
- **`OnInstantiate`** (optional): A JavaScript script to be executed when the 3D stream is read. (See "Specifying JavaScript code" on page 199.)

Creating the stream object

You must create a Cos stream that is based on the file containing the PRC or U3D data (this file was created with 3D authoring software) in order to create a 3D annotation. A data stream can be a buffer in memory, a file, or an arbitrary user-written procedure. A stream is represented by an `ASStm` object, which must be converted into a Cos stream.

To create a Cos stream that is based on the file that contains the PRC or U3D data, perform the following steps:

1. Create an `ASPathName` object that represents the file that contains the PRC or U3D data by invoking the `ASPathFromPlatformPath` method and passing a character pointer that specifies the location of the file that contains the PRC or U3D data. If you are working on the Mac OS platform, invoke the `GetMacPath` method and pass a character pointer that specifies the file location.
2. Declare an `ASFile` object.
3. Populate the `ASFile` object with the PRC or U3D data by invoking the `ASFileSysOpenFile` method and passing the following arguments:
   - An `ASFileSys` object that represents the file system in which the PDF file is located. Invoke the `ASGetDefaultFileSys` method to get the default file system.
   - An `ASPathName` object that represents the path in which the file that contains the PRC or U3D data is located (pass the `ASPathName` object created in step 1).
   - An `ASFileMode` object that represents the mode in which to open the file. For example, specify `ASFILE_READ` to open the file in read mode.
   - The address of an `ASFile` object. The `ASFileSysOpenFile` method populates this argument using the file that was opened (file information is obtained from the `ASPathName` object).
4. Create an `ASStm` object by invoking the `ASFileStmRdOpen` method and passing the following arguments:
   - The `ASFile` object with the file that contains the PRC or U3D data.
   - Length of data buffer, in bytes. If you specify `0`, then the default buffer size (currently 4kB) is used.
5. Invoke the `CosNewStream` method to create a Cos stream containing the data that is located within the `ASStm` object. This Cos stream will become the value of the 3DD entry of the 3D annotation. Pass the following arguments to this method:
   - A `CosDoc` object that specifies the PDF document in which the Cos stream is inserted.
   - An `ASBool` object that specifies whether the Cos stream is an indirect object. Because all streams are indirect objects, this argument must be set to `true`.
   - An `ASStm` object that contains the stream data (pass the `ASStm` object created in step 4).

- A `CosStreamStartAndCode` object that specifies the byte offset from which data reading starts. You can pass `0` to ensure that data reading starts at the beginning of the stream.
- An `ASBool` object that specifies whether the data is encoded using filters specified in the stream dictionary before it is written to the Cos stream.
- A `CosObj` object that represents the stream dictionary. You can invoke the `CosNewNull` method.
- A `CosObj` object that represents the parameters that are used by the encoding filter if the source data is encoded before it is written to the file. If encoding parameters are not required, this value is ignored. For information about encoding filters, see the *PDF Reference*.
- A `CosByteMax` object that specifies the amount of data read from the source. If this value is `-1`, data is read from the source until it reaches the end of the stream.

The `CosNewStream` method returns a `CosObj` object that represents the Cos stream. The following code example creates a Cos stream.

```
//Create an ASPathName that specifies the location of the.U3D file
//u3dFileName is a character pointer that specifies the path to this file
ASPathName u3DPathName =  ASPathFromPlatformPath((void*) u3dFileName);

//Create an ASFile object and populate it
ASFile asFile = NULL;
ASInt32 err = ASFileSysOpenFile(ASGetDefaultFileSys(), u3DPathName,
ASFILE_READ, &asFile);
ASFileSysReleasePath (ASGetDefaultFileSys(), u3DPathName);

if (asFile == NULL)
  AVAlertNote("Error opening 3D data file.");

//Read data stream from the file
ASStm fileStm = ASFileStmRdOpen(asFile, 0);
if (fileStm == NULL)
  AVAlertNote("Empty 3D data stream.");

//Create a new Cos stream and set it under 3DD key in the annot dictionary
CosObj stm3D = CosNewStream(cosDoc, true, fileStm, 0, false, CosNewNull(),
CosNewNull(), -1);
```

Adding the Cos stream to the annotation dictionary

After you create a a `CosObj` object that represents the Cos stream, invoke the `CosDictPutKeyString` method to add the Cos stream as the value of the `3DD` entry of the annotation dictionary:

```
CosDictPutKeyString(cosAnnot,    // The annotation dictionary
  "3DD",                         // The key
  stm3D);                        // The CosObj object used as the value
```

Creating the attributes dictionary

Complete the entries in the 3D stream dictionary. The `CosStreamDict` method obtains the Cos dictionary associated with the stream:

```
CosObj attrObj = CosStreamDict(stm3D);
```

Next, entries can be added to the dictionary. The `Type` and `Subtype` entries both take name objects as values. Therefore, strings specified in the code must be explicitly converted to names:

```
//Set the stream's dictionary
CosDictPutKeyString(attrObj, "Subtype", CosNewName(cosDoc, false,
ASAtomFromString("U3D")));
CosDictPutKeyString(attrObj, "Type", CosNewNameFromString(cosDoc, false,
"3D"));
```

Specifying JavaScript code

You can create JavaScript that manipulates the 3D annotation. JavaScript is optional and if you do not specify it, then the 3D annotation is still inserted into the PDF document; however, it remains a static graphic. For information about creating JavaScript that manipulates 3D annotations, see the *JavaScript for Acrobat 3D Annotations API Reference*.

The following code example creates an `ASFile` object and populates it with the JavaScript file. The `JsFileName` variable is a character pointer that specifies the location of the JavaScript file.

```
//Create a char pointer that specifies the location of the JavaScript file
char*JsFileName = "C:\\3DJavaScript.js"

//Declare an ASFile object that will reference the JavaScript file
ASFile jsFile = NULL;

//Create an ASPathName object based on the JavaScript file
ASPathName JsPathName =  ASPathFromPlatformPath((void*) JsFileName);

//Populate the ASFile object
if(JsPathName)
   ASInt32 err1 = ASFileSysOpenFile(ASGetDefaultFileSys(), JsPathName,
   ASFILE_READ, &jsFile);
```

Next, the data from the file is read into a stream:

```
ASStm JsFileStm = ASFileStmRdOpen(jsFile, 0);
```

In the following code, an entry is added to the stream dictionary in the process of creating the stream, rather than afterwards as in the previous code. First, the `CosNewDict` method is used to create a new dictionary:

```
CosObj dictJsStm = CosNewDict(cosDoc, false, 1);
```

This method requires three parameters:

- The document in which the dictionary is used.
- A Boolean value that specifies whether the dictionary should be an indirect object. All stream dictionaries must be direct; hence the value of this parameter is `false`.
- A hint for the number of entries in the dictionary (however, dictionaries grow dynamically as needed).

Next, the value of the `Filter` entry is set to `FlateDecode` using the `CosDictPutKeyString` method. This means that the stream will be compressed using Flate (ZIP) compression.

```
CosDictPutKeyString(dictJsStm, "Filter",
   CosNewNameFromString(cosDoc, false, "FlateDecode"));
```

Next, the Cos stream is created, using the stream data and attributes dictionary already created:

```
stm3Djscode = CosNewStream(cosDoc, true,
   JsFileStm,    //The stream
```

```
    0, true,
    dictJsStm,   // The stream dictionary
    CosNewNull(), -1);
```

Set it as the value of the `OnInstantiate` entry of the 3D stream dictionary. The following code example specifies a JavaScript script as the value of the `OnInstantiate` entry of the 3D stream dictionary.

```
CosDictPutKeyString(attrObj, "OnInstantiate", stm3Djscode);
```

Then some cleanup is done:

```
ASFileSysReleasePath (ASGetDefaultFileSys(), JsPathName);
ASStmClose(JsFileStm);
```

Setting the default view

A 3D *view* specifies parameters such as position, orientation, and projection style, which are applied to the *virtual camera* associated with the 3D annotation (see section 9.5.3 in the *PDF Reference*). The *default view* is the view that is chosen when the annotation is activated.

3D data typically contains a default initial view. This view is used by default if not otherwise specified. In addition, views can be specified by the entries in a view dictionary.

The `VA` entry in the 3D stream dictionary is an array of view dictionaries. One of the views can be chosen as the default by means of the `3DV` entry in the 3D annotation dictionary or the `DV` entry in a 3D stream dictionary.

The following code creates a view dictionary and specifies its entries. The code assumes the Cos objects `cosAnnot` for the annotation and `cosDoc` for the document have already been obtained. First, a view dictionary is created by invoking the `CosNewDict` method:

```
CosObj cosView = CosNewDict (cosDoc, true, 8);
```

Next, the code sets the following entries: (See Table 9.39 in the *PDF Reference* for more detailed information.)

`Type` (optional): If present, must be the name `3DView`.

`XN` (required): The name of the view, a string that can be displayed in the user interface.

`IN` (optional): The internal name of the view, a string that can be used to refer to the view from other objects, such as in JavaScript code.

`C2W` (optional): A transformation matrix specifying the camera position. To use this, it is also necessary to set the value of the `MS` entry to `M`.

`CO` (optional): A number indicating the distance to the center of orbit for this view.

The following code creates an array of type `double` and specifies values for views:

```
char* externalViewName = "Default View";
char* internalViewName = "Sample3dView";

double gMatrixVals[12] =
   {1.0, 0.0, 0.0, 0.0, 0.000000000000000000612303, -1.0,
   0.0, 1.0, 0.000000000000000000612303, 82.9517, -883.324, 115.166};
float gCOvalue = (float) 725.305;
```

Now the values of the dictionary entries are set:

```
CosDictPutKeyString(cosView, "Type",
   CosNewNameFromString(cosDoc, false,"3DView"));

CosDictPutKeyString(cosView, "XN", CosNewString(cosDoc, false,
   externalViewName, strlen(externalViewName)));

CosDictPutKeyString(cosView, "IN", CosNewString(cosDoc, false,
   internalViewName, strlen(internalViewName)));

CosDictPutKeyString(cosView, "MS",
   CosNewNameFromString(cosDoc, false, "M"));

CosDictPutKeyString(cosView, "CO", CosNewFixed(cosDoc, false,
   FloatToASFixed(gCOvalue)));
```

Here the `C2W` matrix is populated with the appropriate values:

```
CosObj matrixArray = CosNewArray(cosDoc, false, 12);
   for(int i=0; i<12; i++)
      CosArrayPut(matrixArray, i,
         CosNewFloat(cosDoc, false, (float) gMatrixVals[i]));
CosDictPutKeyString(cosView, "C2W", matrixArray);
```

Last, the dictionary is set as the value of the `3DV` key in the annotation dictionary:

```
CosDictPutKeyString(cosAnnot, "3DV", cosView);
```

Setting the annotation appearance

You may optionally provide a *poster* as the initial appearance of the annotation. The poster may be an image or other graphic content that is in a file or in memory. It must be converted to a PDF *form XObject* to be used as the annotation appearance (see section 4.9 of the *PDF Reference*).

The `AP` entry of the annotation dictionary specifies an *appearance dictionary*. This dictionary contains one or more *appearance streams* (see section 8.4.4 of the *PDF Reference*) that are PDF content streams (form XObjects) rendered inside the annotation rectangle.

For 3D annotations, the appearance stream is used in the following situations:

- To provide an annotation appearance for PDF viewers that do not support 3D.
- To provide an initial appearance for the annotation prior to activation. The settings in the activation dictionary determine whether this appearance is ever displayed.

There are several ways to get the poster. The function described below, `GetFormXObjectFromFile`, illustrates one method. The appearance is generated from a separate PDF file containing an image or other content. You call this function as follows:

```
CosObj formXObject = GetFormXObjectFromFile
   (gsPosterFilePath, //The external file
   pdDoc);
```

The function returns a Cos object, `formXObject`, which is the form XObject to be used as the appearance.

```
CosObj cosAnnot = PDAnnotGetCosObj(newAnnot);
CosDoc cosDoc = CosObjGetDoc(cosAnnot);
```

Then you create the appearance dictionary:

```
CosObj apprDict = CosNewDict(cosDoc, false, 1);
```

and set its `N` (normal) entry to the appearance stream obtained above.

```
CosDictPutKeyString(apprDict, "N", formXObject);
CosDictPutKeyString(cosAnnot, "AP", apprDict);
```

The following is the `GetFormXObjectFromFile` function:

```
CosObj GetFormXObjectFromFile
   (char* pdfImageFilePath, //Path of image PDF file
   PDDoc TargetPdDoc)// The current document
{
PDDoc posterPDFDoc = NULL;//Initialization code
PDPage pdPageImage = NULL;
ASPathName asPathName;
CosObj contentFormXObject = CosNewNull();
CosObj formXObject = CosNewNull();
```

First, the PDF file containing the image is opened:

```
if(strlen(pdfImageFilePath) > 0 ) {
   char sPathFlag[16] = "Cstring";
#ifdef MAC_PLATFORM
if (!strchr(pdfImageFilePath,(int)':'))
   strcpy (sPathFlag, "POSIXPath");
#endif

asPathName = ASFileSysCreatePathName (ASGetDefaultFileSys(),
   ASAtomFromString(sPathFlag), pdfImageFilePath, 0);
```

The content to be used is expected to be on the first page of the PDF file. The `PDDocAcquirePage` method returns a `PDPage` object for the first page.

```
pdPageImage = PDDocAcquirePage(posterPDFDoc, 0);
```

The code then uses PDE-layer (PDFEdit) methods that work with the content streams on the PDF page. (See the *Overview* guide for more information on how these methods work.)

The `PDPageAcquirePDEContent` method returns a `PDEContent` object representing the page's contents. The first parameter is the `PDPage` and the second identifies the caller: for PDF Library, it is zero; for plugins, it should be the `gExtensionID` extension:

```
PDEContent pdeContent = PDPageAcquirePDEContent (pdPageImage, 0);
```

The `PDEContentGetAttrs` method gets information about the content in a `PDEContentAttrs` structure:

```
PDEContentAttrs pdeContentAttrs;
PDEContentGetAttrs
   (pdeContent, &pdeContentAttrs, sizeof(pdeContentAttrs));

CosObj contentResources = CosNewNull();
CosDoc pdDocCos = PDDocGetCosDoc(posterPDFDoc);
```

The `PDEContentToCosObj` method converts the `PDEContent` to a form XObject Cos object.

```
PDEContentToCosObj (pdeContent,
   kPDEContentToForm,              // To Form XObject
   &pdeContentAttrs,               // PDEContentAttrsP
```

```
    sizeof(pdeContentAttrs),        // attrsSize,
    pdDocCos,                       // The CosDoc
    NULL,                           // PDEFilterArrayP
    &contentFormXObject,            // Resulting form Cos object
    &contentResources);             // Resulting resource Cos object
```

The following are parameters to this method:

- The `PDEContent` object.
- A flag indicating what type of Cos object should be created; in this case, a form XObject.
- The `PDEContentAttrs` structure containing information about the `PDEContent`.
- The size of the `PDEContentAttrs` structure.
- The Cos document.
- A pointer indicating which filters to use to encode the contents (in this case, null).
- The resulting Cos object (in this case, the form XObject which is the variable `contentFormXObject`).
- The resulting Cos object representing the resources needed by the Cos object. These resources can include fonts and other items (see section 3.7.2 of the *PDF Reference*).

```
  if (!CosObjEqual(contentFormXObject, CosNewNull()) &&
    !CosObjEqual(contentResources, CosNewNull())) {
```

The returned resources must be put into the form XObject's `Resources` dictionary:

```
CosDictPutKeyString(contentFormXObject, "Resources",
     contentResources);
```

The `BBox` entry of the form XObject is required and is set to the value of the page's media box:

```
    ASFixedRect boundingBox;
    PDPageGetMediaBox(pdPageImage, &boundingBox);
    CosObj BBoxArray = CosNewArray(pdDocCos, 4, false);
    CosArrayPut(BBoxArray,0, CosNewInteger(pdDocCos, false,
ASFixedRoundToInt16(boundingBox.left)));
    CosArrayPut(BBoxArray,1, CosNewInteger(pdDocCos, false,
ASFixedRoundToInt16(boundingBox.bottom)));
    CosArrayPut(BBoxArray,2, CosNewInteger(pdDocCos, false,
ASFixedRoundToInt16(boundingBox.right)));
    CosArrayPut(BBoxArray,3, CosNewInteger(pdDocCos, false,
ASFixedRoundToInt16(boundingBox.top)));
    CosDictPutKeyString(contentFormXObject, "BBox", BBoxArray);
    // Set matrix key in form object
```

The `Matrix` entry of the form XObject is set to the values obtained from the page by means of the `PDPageGetDefaultMatrix` method:

```
ASFixedMatrix defaultMatrix;
PDPageGetDefaultMatrix(pdPageImage, &defaultMatrix);
CosObj MatrixArray = CosNewArray(pdDocCos, 6, false);
CosArrayPut(MatrixArray,0, CosNewFixed(
  pdDocCos, false, defaultMatrix.a));
CosArrayPut(MatrixArray,1, CosNewFixed
  (pdDocCos, false, defaultMatrix.b));
CosArrayPut(MatrixArray,2, CosNewFixed
  (pdDocCos, false, defaultMatrix.c));
```

```
CosArrayPut(MatrixArray,3, CosNewFixed
   (pdDocCos, false, defaultMatrix.d));
CosArrayPut(MatrixArray,4, CosNewFixed
   (pdDocCos, false, defaultMatrix.h));
CosArrayPut(MatrixArray,5, CosNewFixed
   (pdDocCos, false, defaultMatrix.v));
CosDictPutKeyString(contentFormXObject, "Matrix", MatrixArray);
}
```

Finally, the `CosObjCopy` method is used to copy the Cos object `contentFormXObject` into the current PDF document. The following are parameters are available:

- The `CosObj` to copy.
- The `CosDoc` for the document in which to copy it.
- A Boolean value: `true` means that all indirectly referenced objects from the source should be copied to the destination.

```
formXObject = CosObjCopy (contentFormXObject,
   PDDocGetCosDoc(TargetPdDoc), true);
```

And finally, there is some cleanup code:

```
ASFileSysReleasePath (ASGetDefaultFileSys(), asPathName);
PDPageRelease(pdPageImage);
return formXObject;
}
```

Setting the activation dictionary

The optional `3DA` entry of the 3D annotation specifies an *activation dictionary* whose entries indicate when the annotation should be activated and deactivated and the state of the 3D content at these times.

When an annotation is inactive, it displays its normal appearance. (See "Setting the annotation appearance" on page 201.) When it is activated, one of its views (specified by the `3DV` entry) is displayed.

First the dictionary is created and set as the `3DA` entry of the 3D annotation:

```
CosObj activationDict = CosNewDict(CosObjGetDoc(cosAnnot), false, 1);
CosDictPutKeyString (cosAnnot, "3DA", activationDict);
```

It is not necessary to set any entries whose default values are acceptable. Here the non-default entries are set.

The `DIS` entry of the activation dictionary specifies the state of the 3D content when it is deactivated. In this case, it is set to `I`, meaning that it should be instantiated. (The default is `U` for uninstantiated.)

```
CosDictPutKeyString (activationDict, "DIS",
   CosNewNameFromString (cosDoc, false, "I"));
```

The code provides a variable to determine the value of the `A` entry. The default value is `XA`, meaning that the annotation needs to be explicitly activated. `PO` means that the annotation should be activated as soon as the page containing the annotation is opened:

```
// Optional activation choice
if(gbShowDefaultViewWhenOpenPage == true)
   CosDictPutKeyString(activationDict, "A",
      CosNewNameFromString (cosDoc, false, "PO"));
```

19 Parsing and Creating PRC Files

Note: Starting Acrobat X, the ability to convert various 3D formats for use with PDF is no longer supported. The associated APIs are, therefore, not available from Acrobat X SDK onwards.

The Adobe Acrobat 3D API lets you develop Acrobat plugins and PDF Library applications that create and access data in a PRC stream. Such a stream can be embedded within a PDF document as a 3D annotation. This chapter explains how to export the contents of a PRC stream, and how to create a PRC stream.

PRC data can appear in PDF documents as streams referenced by 3D annotations. PRC data is a highly-compressed 3D representation supported in Acrobat 8.1 and later. It can represent faceted and exact geometry data. The PRC format is specified in the PRC Format Specification.

Before you read this chapter, it is strongly recommended that you become familiar with 3D concepts and OpenGL, and with 3D annotations. (See "Creating 3D Annotations" on page 194.) OpenGL is an open-source API for setting and accessing 3D data.

For a description of the PRC file format, see the PRC Format Specification.

Working with the Acrobat 3D API

The Acrobat 3D API provides a programmatic interface to the Acrobat 3D Library. It is the only mechanism that lets you create, access, and change PRC data that appears in a PDF document. It does not enable reading or writing PDF documents. (U3D is another 3D format that can be referenced by 3D annotations.)

You can use the Acrobat 3D API to develop the following settings:

- Plugins that work with Acrobat Pro Extended
- Applications that use the PDF Library. (Acrobat Pro Extended must also be installed.) When the Acrobat 3D Library initializes itself, it verifies the presence of Acrobat Pro Extended on the same computer.

Versions

The following table shows the correlation between the Acrobat 3D Library versions and the Acrobat products that supports them. The table also shows the PRC format supported by the Acrobat 3D Library version.

Acrobat 3D Library version	Product	Supports PRC format version
2.0	Acrobat Pro Extended version 9.0 PDF Library version 9.0	7094 and earlier
2.1	Acrobat Pro Extended version 9.1 PDF Library version 9.1	7094 and earlier

Compatibility with different PRC format versions

In addition to supporting the PRC format specified in the previous table, Acrobat 3D Library provides forward and backward compatibility with other PRC format versions:

Forward compatibility: The Acrobat 3D Library is designed to avoid failing if it reads PRC documents that conform to future releases of the PRC format. It does so by ignoring any information it does not understand.

Backward compatibility: In a major release, Acrobat 3D Library support for PRC is backward compatible. Each new version of the software can read PRC files that conform to an earlier PRC format version.

Compatibility between the Acrobat 3D Library and the Acrobat 3D API

The Acrobat 3D API provides the public declarations for a particular version of the Acrobat 3D Library. Plugins developed with one version of the Acrobat 3D API are installed with the same version of Acrobat 3D Library, where that library is contained in Acrobat Pro Extended.

Acrobat 3D Library is designed to support backward compatibility within minor releases of the library.

Requirements

To develop an Acrobat 3D Extended plugin that uses the Acrobat 3D Library, you must download the samples, documentation, and header files from the Acrobat 3D Developer Center. You must also download the Acrobat SDK from the Acrobat Developer Center.

To develop a PDF Library application that uses the Acrobat 3D Library, you must download the samples, documentation, and header files from the Acrobat 3D Developer Center. You must also download the PDF Library, which is available through your Adobe representative.

A licensed copy of Acrobat Pro Extended must be installed on the computer on which the plugin or PDF Library application is running. The Acrobat 3D Library verifies the presence of Acrobat Pro Extended before initializing its relationship with the application or plugin.

The file that contains the Acrobat 3D Library is A3DLIB.dll, which is located in the following directory:

C:\Program Files\Adobe\Acrobat DC\Acrobat

Data types, naming conventions, and character encoding

The Acrobat 3D API adopts the Acrobat library definitions for basic types. Here are the basic types from the Acrobat library: ASInt8, ASInt16, ASInt32, ASUns8, ASUns16, ASUns32, and ASBool.

The Acrobat 3D API declares all non-basic types, which fall into these general categories:

PRC entities. Correspond to the non-terminal structures in the PRC format, such as product occurrences, tessellation base data, and texture transformations. Your software uses NULL pointers to reference PRC entities.

Data structures. Contain the data used to create a PRC entity. Data structures can also receive the data obtained from parsing a PRC entity. There is one data structure for each type of PRC entity.

Acrobat 3D API uses a naming convention that is self-documenting. The name for each function, structure, enumeration, and enumerator provides clues as to its type, its relationship to a PRC entity described in the PRC format, and its role relative to that PRC entity.

Character encoding is UTF-8, which uses the character encoding scheme described by the Internet Engineering Task Force (IETF) document *UTF-8, a transformation format of ISO 10646*. That document is available at http://tools.ietf.org/html/rfc3629.

Structured and recursive nature of PRC parsing

Plugins and applications that process PRC data should be structured to reflect the hierarchy of the PRC format. Ideally, they implement one function to process or create each of the PRC entities. Additionally, they implement other functions to perform repeated tasks, such as processing or creating attribute data that can apply to a broad category of PRC entities.

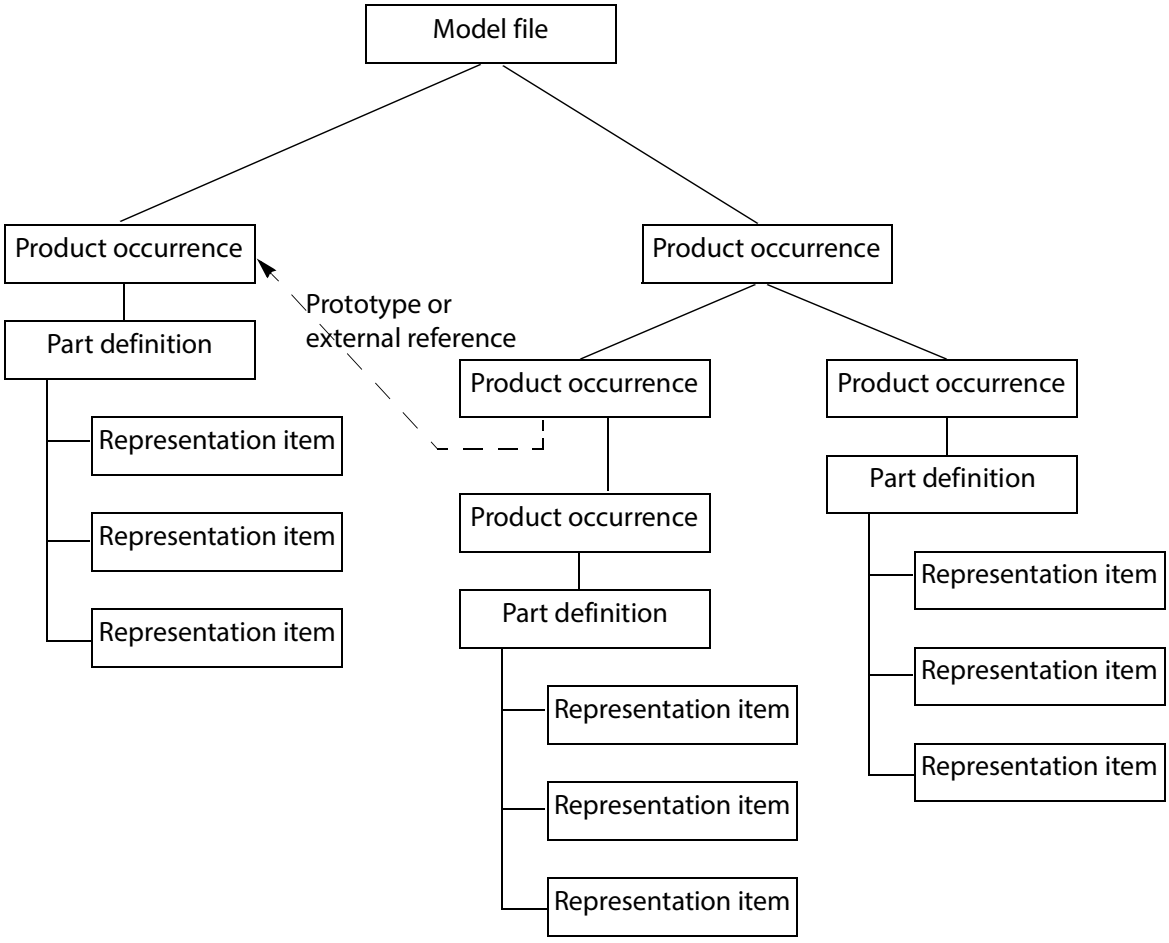
Additionally, plugins and applications that process PRC data should support recursive function calls.

The PRC format is a highly-compressed hierarchical structure. The structural entities in a PRC document enable the definition of subassemblies that represent some part of the overall image, such as a gear or a nut. Subassemblies are then grouped to create more complex assemblies, or they can be referenced from other subassemblies similar to a macro.

The following diagram illustrates the hierarchical relationships that can exist between the structural elements:

- Model files can have multiple child product occurrences.
- Product occurrences can have multiple child product occurrences and a part definition. They can also reference product occurrences that act as prototypes or external data.

- Part definitions have multiple child representation items.



Because some PRC entities can contain child entities of the same type, plugins that parse or create PRC data use recursion.

Implementing external linking in your plugin

The Acrobat 3D Library's import library (the .lib file) is not provided in the Acrobat SDK or through any other means. As a result, you must use explicit linking to resolve the external links in your plugin.

With explicit linking, the executable using the DLL must make function calls to explicitly load and unload the DLL, and to access the DLL's exported functions. Your plugin must call the exported functions through a function pointer. More specifically, your plugin must perform the following tasks:

- Call the `LoadLibrary` function (or a similar function) to load the DLL and obtain a module handle.
- Call the `GetProcAddress` function to obtain a function pointer to each exported function that your plugin calls. Because applications are calling the DLL's functions through a pointer, the compiler does not generate external references, so there is no need to link with an import library.
- Call the `FreeLibrary` function when done with the DLL.

Note: The Acrobat 3D API functions in this chapter are described as though implicit linking were possible. Wherever you see a direct call to one of these functions, you should instead use the pointer to the function. See Implementing external linking.

Implementing external linking

1. Call the `LoadLibrary` function (or a similar function) to load the DLL and obtain a module handle.
2. Call the `GetProcAddress` function to obtain a function pointer to each exported function that your plugin calls. Because applications are calling the DLL's functions through a pointer, the compiler does not generate external references, so there is no need to link with an import library.

Example: ***Main code segment that loads the Acrobat 3D library and defines function pointers***

```
HMODULE hModuleA3DPRCSDK = A3DPRCLoadLibrary();

if (!hModuleA3DPRCSDK) {
   AVAlertNote("Failed to load A3DLIB.dll!");
   _unlink(prcName);
   _unlink(jsName);
   return;
}

A3DPRCFunctionPointersInitialize(hModuleA3DPRCSDK);

/*-------------------------------------
/* Your plugin initializes its relationship with the Acrobat 3D Library
/* and then it parses or creates PRC content.
/*-------------------------------------

A3DPRCUnloadLibrary(hModuleA3DPRCSDK);
} else {
   char strMsg[128];
   sprintf(strMsg, "A3DDllInitialize returned %d\n", iRet);
   AVAlertNote(strMsg);
   _unlink(prcName);
   _unlink(jsName);
   A3DPRCUnloadLibrary(hModuleA3DPRCSDK);
}
```

Example: ***Loading the DLL and obtaining a module handle***

```
HMODULE A3DPRCLoadLibrary()
{
   HMODULE hModuleA3DPRCSDK;

   wchar_t acFilePath[MAX_PATH];
   GetModuleFileNameW(NULL, acFilePath, MAX_PATH);
   wchar_t* backslash = wcsrchr(acFilePath, L'\\');

   if (backslash)
      acFilePath[backslash - acFilePath] = 0;

   wcscat(acFilePath, L"\\A3DLIB.dll");
```

```
#ifdef UNICODE
   hModuleA3DPRCSDK = LoadLibraryExW(acFilePath, NULL,
LOAD_WITH_ALTERED_SEARCH_PATH);
#else
   hModuleA3DPRCSDK = LoadLibraryExA(acFilePath, NULL,
LOAD_WITH_ALTERED_SEARCH_PATH);
#endif
   if (hModuleA3DPRCSDK)
      return hModuleA3DPRCSDK;

   return NULL;
}
```

Example: ***Setting up the function pointers***

```
void A3DPRCFunctionPointersInitialize(HMODULE hModule)
{
#define A3D_API(returntype,name,params) st_PF##name =
(PF##name)GetProcAddress(hModule,#name);
#include <A3DSDK.h>
#include <A3DSDKTypes.h>
#include <A3DSDKBase.h>
#include <A3DSDKErrorCodes.h>
#include <A3DSDKGeometry.h>
#include <A3DSDKTessellation.h>
#include <A3DSDKGraphics.h>
#include <A3DSDKStructure.h>
#include <A3DSDKRootEntities.h>
#include <A3DSDKRepItems.h>
#include <A3DSDKTessellation.h>
#include <A3DSDKMarkup.h>
#include <A3DSDKGlobalData.h>
#include <A3DSDKTexture.h>
#include <A3DSDKMisc.h>
#include <A3DSDKGeometryCrv.h>
#include <A3DSDKGeometrySrf.h>
#include <A3DSDKTopology.h>
#undef A3D_API
}
```

Example: ***Declaring the macro that resolves to the function pointer***

```
#define A3DCALL(name,params) st_PF##name params
```

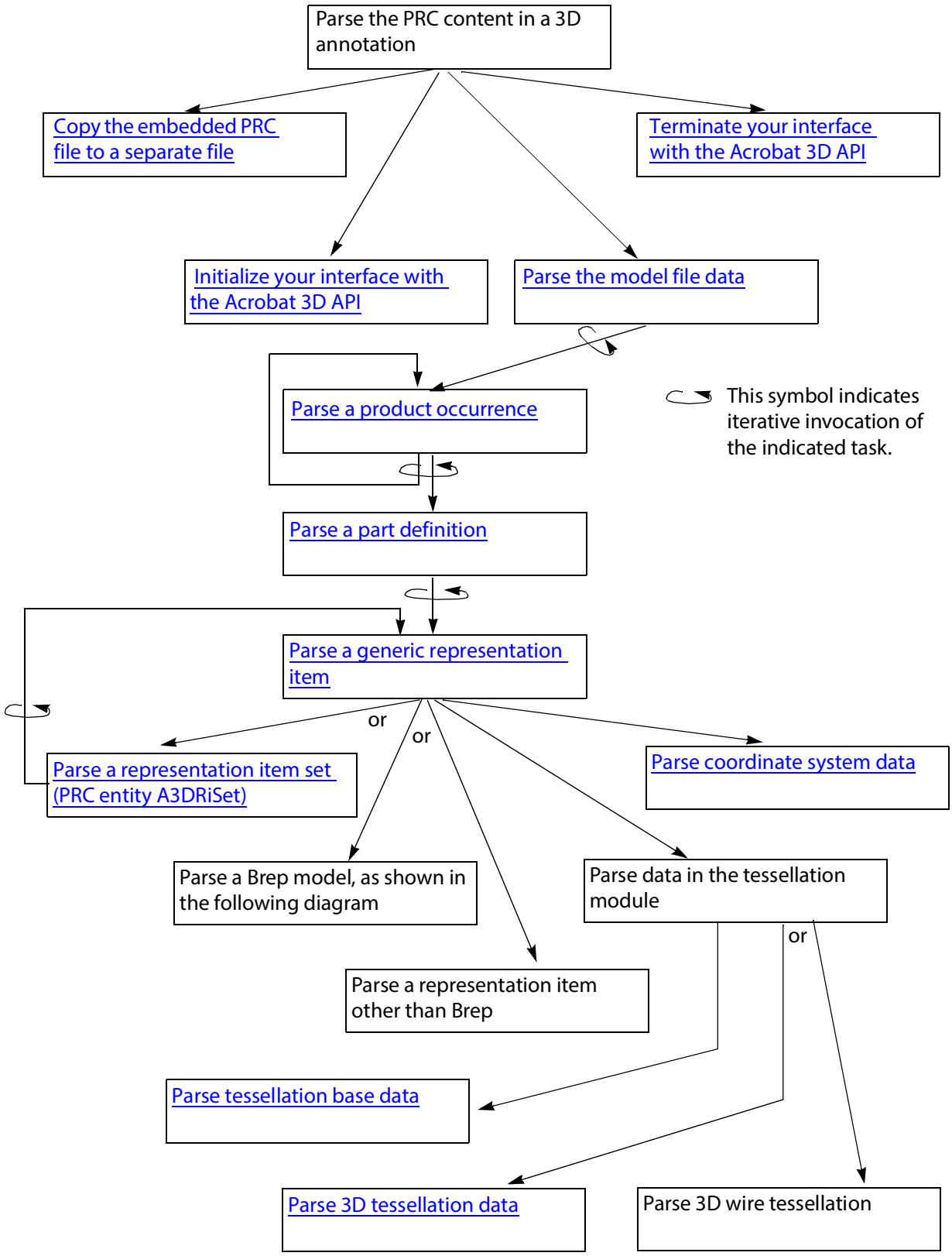
Example: ***Using the macro to invoke an Acrobat 3D API function***

```
if(iErr == A3D_SUCCESS) {
     ASInt32 iRet = A3DCALL(A3DDllInitialize,(iMajorVersion, iMinorVersion));
     if(iRet == A3D_SUCCESS) {
     …
   }
}
```

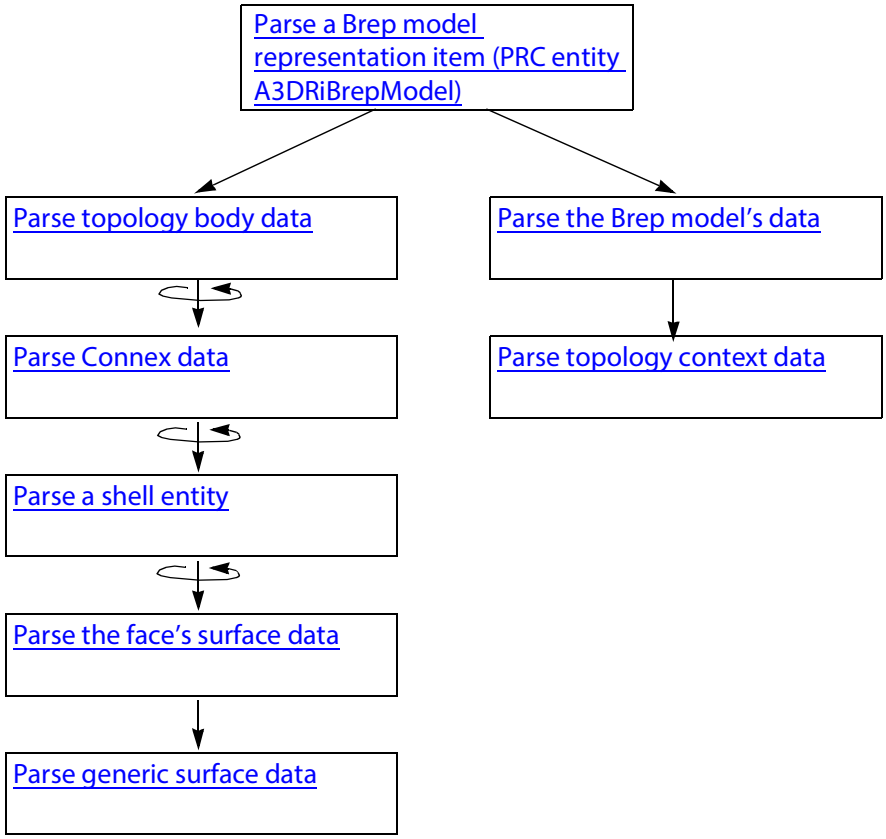
Parsing a PRC file

This section describes how to parse the data in a PRC file. You can then export the PRC data to an external file that uses a different CAD format, provided you understand the structure of that format and have an API that supports that format. It is beyond the scope of this chapter to explain how to convert PRC data into a specific CAD format. It is also beyond the scope of this chapter to provide an exhaustive explanation of parsing all aspects of a PRC file.

The following diagram shows the structure of a plugin that parses PRC data.



The following diagram shows the sequence of tasks required to parse a Brep model entity.



Handling errors

Most of the Acrobat 3D API functions return an integer that indicates success or failure. A return value of `A3D_SUCCESS` indicates success, and any negative return value indicates failure. The following example shows one approach for evaluating this returned result:

```
ASInt32 iRet = A3DAsmModelFileLoadFromFile(acFileName, NULL, &p);
if (iRet == A3D_SUCCESS) {
   …
} else
   fprintf("Error: %1d\n", iRet);
```

Although subsequent explanations in this section omit this description, you must still check for errors when an Acrobat 3D API function returns.

Copying the embedded PRC file to a separate file

This section explains how to export a PRC stream represented in a PDF document as a 3D annotation.

Copy the embedded PRC file to a separate file

3. Retrieve the 3D annotation from within a PDF document. (See "Retrieving existing annotations" on page 102.) Ensure that the annotation dictionary type is `3DD` and that the subtype is `PRC`.

4. Write the PRC stream to a separate file. This will simplify accessing the data in the stream.

Initializing the Acrobat 3D API

This section explains how to initialize the Acrobat 3D API.

Initialize your interface with the Acrobat 3D API

5. Verify that your plugin is compatible with the Acrobat 3D Library by comparing the version of the Acrobat 3D API against which you compiled and the version of the Acrobat 3D Library available to your plugin. Obtain the version identifiers for the currently installed Acrobat 3D Library by invoking the `A3DDllGetVersion` function. Evaluate the returned values as follows:
   - Ensure that the value of the `piMajorVersion` argument matches the `A3D_DLL_MAJORVERSION` enumeration.
   - Ensure that the value of the `piMinorVersion` argument is equal to or greater than the `A3D_DLL_MINORVERSION` enumeration. The Acrobat 3D Library provides backward compatibility for earlier releases of the API that have the same major version identifier.
6. Initialize the Acrobat 3D API by supplying the version identifiers for the Acrobat 3D API.

```
if(iErr == A3D_SUCCESS) {
     ASInt32 iRet = A3DCALL(
        A3DDllInitialize,(A3D_DLL_MAJORVERSION, A3D_DLL_MINORVERSION));
     if(iRet == A3D_SUCCESS) { … }
}
```

Note: For simplicity, this example uses a syntax that assumes the import library is available. That library is not publicly available, so your code must use a syntax that supports external linking (see Implementing external linking in your plugin). The following example shows the syntax to use:

Parsing structure PRC entities

The Acrobat 3D API Reference groups the PRC entities that provide structure to the PRC document into the structure module. The PRC entities in the structure module are described here:

Model file: Is a root PRC entity. There is only one model file in a PRC file.

Product occurrence: Can contain multiple child product occurrences and one part definition. It can also reference other product occurrences used as prototypes or external data.

Part definition: Contains representation items.

Filter: Specifies inclusion or exclusion of topology, geometry, tessellation, or graphic entities, based on the entity's layer or entity type.

Parse the model file data

1. Load the PRC file into memory by invoking the `A3DAsmModelFileLoadFromFile` function. In the following example, the `acFileName` argument points to an `A3DUTF8Char` array that contains the name of the PRC file, the second parameter must be `NULL`, and the `p` pointer references a null `A3DAsmModelFile` object in which the Acrobat 3D Library stores the PRC model file.

```
A3DAsmModelFile* p = NULL;
ASInt32 iRet = A3DAsmModelFileLoadFromFile(acFileName, NULL, &p);
```

2. Populate the model file with the model file data by invoking the `A3DAsmModelFileGet` function to place the data in that structure. The `A3D_INITIALIZE_DATA` macro clears the memory allocated to the structure and checks the size of the structure to avoid alignment problems. The

`A3DAsmModelFileGet` function creates a data structure from the model file pointer created in the previous step.

```
A3DAsmModelFileData sData;
A3D_INITIALIZE_DATA(sData);
ASInt32 iRet = A3DAsmModelFileGet(p, &sData);
```

3. Save the value of the `m_dUnit` member in the structure in which you are saving data obtained from parsing the PRC file (your export structure). This member specifies a multiple of millimeters. The value of this member must be non-zero. A value of 1 indicates the units are in millimeters; and a value of 10 indicates the units are in centimeters.

4. If the `m_bUnitFromCAD` is present, save its value in your export structure. If the `bUnitFromCAD` member is true, then the `m_bUnit` is from the native CAD file.

5. Parse the entity base. (See Parsing root-base entity data.)

6. Declare an `A3DMiscCascadedAttributes` structure. This will be the root entry in a stack on which you store child product occurrences and other subordinate structures. This stack is used to iterate through the PRC structure.

```
A3DMiscCascadedAttributes* pAttr;
A3DMiscCascadedAttributesCreate(&pAttr);
```

   As you parse subsequent PRC entities, you push the attributes onto this stack. For more information about using cascaded attribute stacks, see "Parsing PRC entities that specify graphics" on page 226 or click the Related Pages tab on the Acrobat 3D API Reference.

7. Parse each product occurrence pointer in the `sData.m_ppPOccurrences` array. The number of pointers is specified in `sData.m_uiPOccurrencesSize`.

```
for (ASUns32 ui = 0; ui < sData.m_uiPOccurrencesSize; ui++)
   parsePOccurrence(sData.m_ppPOccurrences[ui], pAttr);
```

   For information about parsing product occurrences, see Parse a product occurrence.

Parse a product occurrence

1. Parse the entity base, saving the entity name and other relevant data to your export structure. (See Parsing root-base entity data.)

2. Create and push a cascaded attributes structure for a product occurrence object, and push that structure onto the stack. (See Parsing graphic attributes using miscellaneous cascaded attributes.) Save relevant values to your export structure.

3. Declare and initialize an `A3DAsmProductOccurrenceData` structure.

```
A3DAsmProductOccurrenceData sData;
A3D_INITIALIZE_DATA(sData);
```

4. Get the product occurrence data by invoking the `A3DAsmProductOccurrenceGet` function.

```
iRet = A3DAsmProductOccurrenceGet(p, &sData);
```

5. Parse the product occurrence data to identify subordinate or imported product occurrences and recursively parse the contents of each, as described in "Parse a product occurrence" on page 215 (this section). The member names and their significance are described below:

- `m_pPrototype` : Pointer to a product occurrence prototype, which typically represents a subassembly or part
- `m_pExternalData` : Pointer to an external product occurrence
- `m_uiPOccurrencesSize` : Number of child product occurrences, where the `m_ppPOccurrences` array references those child product occurrences

The following example parses prototype or external product occurrences. The `sData` variable is a populated `A3DAsmProductOccurrenceData` structure.

```
if (sData.m_pPrototype) {
   parsePOccurrence(sData.m_pPrototype, pAttr);
}
if (sData.m_pExternalData) {
   parsePOccurrence(sData.m_pExternalData, pAttr);
}
```

The following example parses child product occurrences (`sData.ppPOccurrences[ui]`. As before, the `sdata` variable is a populated `A3DAsmProductOccurrenceData` structure.

```
for (ASUns32 ui = 0; ui < sData.m_uiPOccurrencesSize; ui++)
   parsePOccurrence(sData.m_ppPOccurrences[ui], pAttr);
```

6. If the product occurrence data references a part (`sData.m_pPart`), parse it. (See "Parse a product occurrence" on page 215.)
7. Parse any scene display parameters in the product occurrence.
8. Delete the `A3DAsmProductOccurrenceData` structure created in Step 3 by invoking the `A3DAsmProductOccurrenceGet` function with the first argument set to `NULL` and the second argument pointing to the product occurrence data.

```
iRet = A3DAsmProductOccurrenceGet(NULL, &sData);
```

Parse a part definition

1. Parse the entity base, saving the entity name and other relevant data to your export structure. (See Parsing root-base entity data.)
2. Create and push a cascaded attributes structure for a product occurrence object, and push that structure onto the stack. (See Parsing graphic attributes using miscellaneous cascaded attributes.) Record information from the data structure of the miscellaneous cascaded attributes structure that is meaningful to your representation.
3. Declare and initialize an `A3DAsmPartDefinitionData` structure, as shown in the following example:

```
A3DAsmPartDefinitionData sData;
A3D_INITIALIZE_DATA(sData);
iRet = A3DAsmPartDefinitionGet(p, &sData);
```

4. Parse each representation item referenced by the part definition (See Parsing representation items.) In the following example, `sData.m_uiRepItemsSize` is the number of representation item entities in the part definition, and entry in the `sData.m_ppRepItems[ui]` array references a separate representation item.

```
for (ASUns32 ui = 0; ui < sData.m_uiRepItemsSize; ui++)
   parseRI(sData.m_ppRepItems[ui], pAttr);
```

5. Delete the `A3DAsmPartDefinitionData` structure created in Step 3 by invoking the `A3DAsmPartDefinitionGet` function with the first argument set to `NULL` and the second argument pointing to the part definition data.

```
A3DAsmPartDefinitionGet(NULL, &sData);
```

Parsing representation items

Representation items describe objects present in the 3D image, such as a wheel or a bolt. The exception is the representation item set (type `kA3DTypeRiSet`), which is a container for other representation items.

Parse a generic representation item

1. Parse the entity base, saving the entity name and other relevant data to your export structure. (See Parsing root-base entity data.)

2. Create and push a miscellaneous cascaded attributes structure for the representation item, and then get the data for that structure (See "Parsing PRC entities that specify graphics" on page 226.) Record information from the miscellaneous cascaded attributes data structure that is meaningful to your representation.

3. Invoke the `A3DEntityGetType` function to determine the type of representation item, providing the following arguments:
   - The first argument (`p` in the following example) is a pointer to the representation item.
   - The second argument (`eType` in the following example) is a pointer to a variable in which the function stores the type enumerator.

   ```
   A3DEEntityType eType;
   iRet = A3DEntityGetType(p, &eType);
   ```

4. Parse the representation item depending on the type of representation item, as shown in the following example. (See Parse a representation item set (PRC entity A3DRiSet) or Parse a Brep model representation item (PRC entity A3DRiBrepModel).)

```
switch(eType) {
case kA3DTypeRiBrepModel:
   parseRiBrepModel(p, father);
   break;
case kA3DTypeRiSet:
   …
   break;
case kA3DTypeRiPointSet:
   …
   break;
…
default:
   …
}
```

5. Get the representation item attributes by invoking the `A3DRiRepresentationItemGet` function.

```
A3DRiRepresentationItemData sData;
A3D_INITIALIZE_DATA(sData);
iRet = A3DRiRepresentationItemGet(p, &sData);
```

6. Parse the tessellation base data referenced by the `m_pTessBase` member. (See Parsing tessellation PRC entities.)

```
parseTess(sData.m_pTessBase, p, pAttr);
```

7. Parse the coordinate system referenced by the `m_pCoordinateSystem` member. (See Parse coordinate system data.)

```
parseRiCSys(sData.m_pCoordinateSystem);
```

8. Delete the `A3DRiRepresentationItemData` created in Step 5 by invoking the `A3DRiRepresentationItemGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sData` in the following example).

```
A3DRiRepresentationItemGet(NULL, &sData);
```

Parse a representation item set (PRC entity A3DRiSet)

1. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

2. Create and push a cascaded attributes structure for a product occurrence object, and push that structure onto the stack. (See Parsing graphic attributes using miscellaneous cascaded attributes.) Save relevant values to your export structure.

3. Declare and initialize an `A3DRiSetData` structure, and then get the data from the representation PRC entity.

```
A3DRiSetData sData;
A3D_INITIALIZE_DATA(sData);
iRet = A3DRiSetGet(p, &sData);
```

4. Parse each representation item in the representation item set. (See Parse a generic representation item.) The `m_uiRepItemsSize` field specifies the number of child representation items, and the `m_ppRepItems[]` field is an array of pointers to the child representation items.

```
for(ASUns32 ui = 0; ui<sData.m_uiRepItemsSize; ui++)
   parseRI(sData.m_ppRepItems[ui], pAttr, son);
```

5. Delete the `A3DRiSetGet` function created in Step 3 by invoking the `A3DAsmPartDefinitionGet` function with the first argument set to `NULL` and the second argument pointing to the part definition data.

```
A3DRiSetGet(NULL, &sData);
```

Parse a Brep model representation item (PRC entity A3DRiBrepModel)

1. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

2. Get the Brep model data by invoking the `A3DRiBrepModelGet` function. The first argument (`p` in the following example) is a pointer to the PRC entity, and the second argument (`&sData` in the following example) is the location of the `A3DRiBrepModelData` structure.

```
A3DRiBrepModelData sData;
A3D_INITIALIZE_DATA(sData);
ASInt32 iRet = A3DRiBrepModelGet(p, &sData);
```

3. Parse the Brep model's topology body data, referenced by the `m_pBrepData` member. (See Parse topology body data.)

4. Parse the Brep model's data, referenced by the `m_pBrepData` member. (See Parse the Brep model's data.)

   Note: The same member (`m_pBrepData`) is used to parse topology body data and topology model data. The Brep model's topology body data is recast as the `A3DTopoBody` type, which is an abstract root type for any topological body. The `A3DTopoBodyGet` function takes an argument of type `A3DTopoBody`.

5. Delete the `A3DRiBrepModelData` created in Step 2 by invoking the `A3DRiBrepModelGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sData` in the following example).

```
A3DRiBrepModelGet(NULL, &sData);
```

Parse coordinate system data

1. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

2. Declare and initialize an `A3DRiCoordinateSystemData` structure.

```
A3DRiCoordinateSystemData sData;
A3D_INITIALIZE_DATA(sData);
```

3. Get the coordinate system data by invoking the `A3DRiCoordinateSystemGet` function. The first argument (`p` in the following example) is a pointer to the `A3DRiCoordinateSystem` entity, and the second argument (`&sData` in the following example) is the location of the `A3DRiCoordinateSystemData` structure.

```
ASInt32 iRet = A3DRiCoordinateSystemGet(p, &sData);
```

4. Parse the coordinate system transformation data referenced by the `m_pTransformation` member of the `A3DRiCoordinateSystemData` structure.

5. Delete the `A3DRiCoordinateSystemData` structure created in Step 2 by invoking the `A3DRiCoordinateSystemGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sData` in the following example).

```
A3DRiCoordinateSystemGet(NULL, &sData);
```

Parsing tessellation PRC entities

Tessellation entities represent polygon facets.

Parse tessellation base data

1. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

2. Populate an `A3DTessBaseData` structure with the tessellation base data by invoking the `A3DTessBaseGet` function. The first argument (`p` in the following example) is a pointer to the tessellation base data, and the second argument (`&sData` in the following example) is the location of the `A3DTessBaseData` structure.

```
A3DTessBaseData sData;
A3D_INITIALIZE_DATA(sData);
ASInt32 iRet = A3DTessBaseGet(p, &sData);
```

3. Export the coordinate size (`sData.m_uiCoordSize`), the coordinates array (`sData.m_pdCoords`), and the calculate attributes (`sData.m_bIsCalculated`) to your tessellation base data element.

4. Parse the tessellation data based on its type. The PRC format defines these types for tessellation data:
   - `kA3DTypeTess3D`, which is used for solids and surfaces (See Parse 3D tessellation data.)
   - `kA3DTypeTess3DWire`, which is used for 3D wireframes
   - `kA3DTypeTessMarkup`, which is used for markups

```
A3DEEntityType eType;
ASInt32 iErr = A3DEntityGetType(p, &eType);
if (iErr == A3D_SUCCESS) {
   switch(eType) {
case kA3DTypeTess3D:
   parse3DTess(p, pRepItem, pFatherAttr);
   break;
case kA3DTypeTess3DWire:
   parse3DWireTess(p);
   break;
case kA3DTypeTessMarkup:
   parse3DTessMarkup(p);
   break;
default:
   // error response.
}
```

5. Delete the `A3DTessBaseData` created in Step 2 by invoking the `A3DRiBrepModelGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sData` in the following example).

```
A3DTessBaseGet(NULL, &sData);
```

Parse 3D tessellation data

1. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

2. Populate an `A3DTess3DData` structure with the tessellation data by invoking the `A3DTess3DGet` function.

```
A3DTess3DData sData;
A3D_INITIALIZE_DATA(sData);
ASInt32 iRet = A3DTess3DGet(p, &sData);
```

3. Represent relevant data from the 3D tessellation data element in your export structure.

4. For each face in the tessellation data, create and push the cascaded attributes face data and then access its data members. (See Parsing graphic attributes using miscellaneous cascaded attributes.) In the following example, the `CreateAndPushCascadedAttributesFace` is a private function that pushes the miscellaneous cascaded attributes for a tessellation face onto the stack.

```
ASUns32 uiNumberOfFaces = sData.m_uiFaceTessSize;
```

```
for (ASUns32 ui = 0; ui < uiNumberOfFaces; ui++) {
   A3DTessFaceData& sTessFaceData = sData.m_psFaceTessData[ui];

   A3DMiscCascadedAttributes* pAttr;
   A3DMiscCascadedAttributesData sAttrData;
   // Read CascadedAttributes for one of the facec
   CreateAndPushCascadedAttributesFace(pRepItem, p, &sTessFaceData,
      ui, pFatherAttr, &pAttr, &sAttrData);

   A3DMiscCascadedAttributesDelete(pAttr);
   A3DMiscCascadedAttributesGet(NULL, &sAttrData);
}
```

5. Delete the cascaded attributes data structure.

```
A3DMiscCascadedAttributesDelete(pAttr);
```

6. Delete the cascaded attributes structure.

```
A3DMiscCascadedAttributesGet(NULL, &sAttrData);
```

7. Delete the `A3DTess3DData` structure created in Step 2.

```
A3DTess3DGet(NULL, &sData);
```

Parsing topology PRC entities

The Acrobat 3D API Reference groups the PRC entities that specify topology into the topology module. The PRC entities in this module specify the surfaces of 3D objects. This section describes how to parse the topology data in a Brep model.

Topology data in a Brep model contains data specific to the Brep and data that is generic to topology models.

Data specific to Brep topology models

The following entities contain the geometric data that represents the Brep data:

Topology Brep data: A topological boundary representation comprised of a bounding box and references to multiple Connex entities (See Create a topology Brep data entity.)

Connex: A collection of shell entities, such as a hollow sphere that contains another sphere that is represented by two connexes (See Parse Connex data.)

Shell: A collection of face entities (See Parse a shell entity.)

Face: A surface and a collection of loops (See Parse the face's surface data.)

Data general to all topology models

The following entities contain data that applies to any topology model:

Topology body data: A mask indicating the source of the bounding box (See Parse topology body data.)

Topology context: A mask indicating the behavior of the topology context (See Parse topology context data.)

Parse topology body data

1. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

2. Get the topology body data by invoking the `A3DTopoBodyGet` function. The first argument (`p` in the following example) is a pointer to the Brep model's topology body data, and the second argument (`&sData` in the following example) is the location of the `A3DTopoBodyData` structure.

```
A3DTopoBodyData sData;
A3D_INITIALIZE_DATA(sData);
ASInt32 iRet = A3DTopoBodyGet(p, &sData);
```

3. Parse the context of the topology body context data, referenced by the `m_pContext` member. (See Parse topology context data.)

```
parseContext(sData.m_pContext);
```

4. Delete the `A3DTopoBodyData` structure created in Step 2, by invoking the `A3DTopoBodyGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sData` in the following example).

```
A3DTopoBodyGet(NULL, &sData);
```

Parse topology context data

1. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

2. Get the data in the topology context data by invoking the `A3DTopoContextGet` function. The first argument (`p` in the following example) is a pointer to the topology context data, and the second argument (`&sData` in the following example) is the location of the `A3DTopoContextData` structure.

```
A3DTopoContextData sData;
A3D_INITIALIZE_DATA(sData);
ASInt32 iRet = A3DTopoContextGet(p, &sData);
```

3. If you plan to evaluate graphic data relative to the current surface, save the Body Scale for use in scaling the face's surface data to fit the Nurbs surface. (See Parse the face's surface data.) Body scale has no particular meaning for Nurb data conversion and can be called only on specific surface types.

```
stdContextScale = sData.m_dScale;
```

4. Delete the `A3DTopoContextData` structure, created in Step 2, by invoking the `A3DTopoContextGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sData` in the following example).

```
A3DTopoContextGet(NULL, &sData);
```

Parse the Brep model's data

1. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

2. Get the data in the Brep model by invoking the `A3DTopoBrepDataGet` function. The first argument (`p` in the following example) is a pointer to the Brep model's data, and the second argument (`&sData` in the following example) is the location of the `A3DBrepDataData` structure.

```
A3DTopoBrepDataData sData;
A3D_INITIALIZE_DATA(sData);
ASInt32 iRet = A3DTopoBrepDataGet(p, &sData);
```

3. The Brep data can reference Connex entities. The `m_uiConnexSize` member indicates the number of such references, and the `m_ppConnexes[]` member provides the references. Parse each Connex entry. (See Parse Connex data.)

```
for(ASUns32 ui = 0; ui < sData.m_uiConnexSize; ui++)
   parseConnex(sData.m_ppConnexes[ui]);
```

4. Save the bounding box referenced by `m_sBoundingBox` as an attribute of the Brep data element in your export structure.

5. Delete the `A3DTopoBrepDataData` structure, created in Step 2, by invoking the `A3DTopoBrepDataGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sData` in the following example).

```
A3DTopoBrepDataGet(NULL, &sData);
```

Parse Connex data

1. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

2. Get the data in the Brep model by invoking the `A3DTopoConnexGet` function. The first argument (`p` in the following example) is a pointer to the Connex entity, and the second argument (`&sData` in the following example) is the location of the `A3DTopoConnexData` structure.

```
A3DTopoConnexData sData;
A3D_INITIALIZE_DATA(sData);
ASInt32 iRet = A3DTopoConnexGet(p, &sData);
```

3. The Connex data can reference multiple Shell entities. The `m_uiShellSize` member indicates the number of such references, and the `m_ppShells[]` member provides the references. Parse each Shell entity. (See Parse a shell entity.)

```
for(ASUns32 ui = 0; ui < sData.m_uiShellSize; ui++)
   parseShell(sData.m_ppShells[ui]);
```

4. Delete the `A3DTopoConnexData` structure, created in Step 2, by invoking the `A3DTopoConnexGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sData` in the following example).

```
A3DTopoConnexGet(NULL, &sData);
```

Parse a shell entity

1. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

2. Get the data in the topology shell entity by invoking the `A3DTopoShellGet` function. The first argument (`p` in the following example) is a pointer to the topology shell entity, and the second argument (`&sData` in the following example) is the location of the `A3DTopoShellData` structure.

```
A3DTopoShellData sData;
A3D_INITIALIZE_DATA(sData);
ASInt32 iRet = A3DTopoShellGet(p, &sData);
```

3. The shell data can reference multiple face entities. The `m_uiFaceSize` member indicates the number of such references, and the `m_ppFaces[]` member provides the references. Parse each face entity, as described in Parse the face's surface data. [

```
for(ASUns32 ui = 0; ui < sData.m_uiFaceSize; ui++)
   parseFace(ui, sData.m_ppFaces[ui]);
```

4. Save the closed indicator (the `m_bClosed` member) and the orientation of the surface normal with respect to the shell normal (the `m_pucOrientationWithShell` member) as attributes in the shell element in your export structure.

5. Delete the `A3DTopoShellData` structure, created in Step 2, by invoking the `A3DTopoShellGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sData` in the following example).

```
A3DTopoShellGet(NULL, &sData);
```

Parse the face's surface data

6. Parse the entity base. (See Parsing root-base entity data.) Save the entity name and other relevant data to your export structure.

7. Get the data in the topology face entity by invoking the `A3DTopoFaceGet` function. The first argument (`p` in the following example) is a pointer to the topology face entity, and the second argument (`&sData` in the following example) is the location of the `A3DTopoFaceData` structure.

```
A3DTopoFaceData sData;
A3D_INITIALIZE_DATA(sData);
ASInt32 iRet = A3DTopoFaceGet(p, &sData);
```

8. Parse the surface data referenced by the `m_pSurface` member in the `A3DTopoFaceData` structure. (See Parse generic surface data.)

9. If you are converting the PRC data to a 3D model that does not support UV mapping, you must create a curve on surfaceto editors: this is a standard graphic term and convert it to Nurbs data. (See Convert surface data to Nurbs (for other than UV-mapped surfaces).)

10. Delete the `A3DTopoFaceData` structure created in Step 7 by invoking the `A3DTopoFaceGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sData` in the following example).

```
A3DTopoFaceGet(NULL, &sData);
```

Convert surface data to Nurbs (for other than UV-mapped surfaces)

1. Declare and initialize an `A3DCrvOnSurfData` structure.

2. Populate the `A3DCrvOnSurfData` structure with the surface base and surface domain information from the `A3DTopoFaceData` structure.

3. Create an `A3DCrvOnSurf` PRC entity by invoking the `A3DCrvOnSurfCreate` function. The first argument in this function call is the `A3DCrvOnSurfData` structure, and the second is a pointer to the resulting PRC entity.

4. Convert a curve on the surface in the `A3DCrvOnSurf` PRC entity by invoking the `A3DCrvBaseGetAsNurbs` function.

Parse generic surface data

1. Determine the type of surface data by invoking the `A3DEntityGetType` function. The first argument references the surface, and the second is a pointer to a variable in which the function stores the type enumerator.

2. Parse the surface data using the Acrobat 3D API functions most appropriate for the surface type. Use the `A3DEntityGetType` function to determine surface type. If the surface type is of type `kA3DTypeSurfNurbs` or if the surface type is unknown, parse the surface data as Nurbs surface data (see Parse Nurbs surface data).

```
A3DEEntityType eType;
iRet = A3DEntityGetType(p, &eType);
switch (eType) {
case kA3DTypeSurfNurbs:
   parseSurfaceNurbs(p);
   break;
case kA3DTypeSurfSphere:
   parseSurfSphere(p);
   break;
case kA3DTypeSurfBlend01:
   parseSurfaceBlend01(p);
   break;
…
}
```

Parse Nurbs surface data

1. Declare and initialize an `A3DSurfNurbsData` structure.

2. Populate the surface Nurbs data structure by invoking the `A3DSurfBaseGetAsNurbs` function, providing the following arguments:
   - The first argument (`sData.m_pSurface` in the following example) is a pointer to the surface base entity.
   - The second argument (`1e-3 / stdContextScale`) is a ratio that the function uses to scale the Nurbs data. The value `1e-3` specifies the desired tolerance (0.001 mm in the following example). The `stdContextScale` global variable is the Body Scale, which was obtained from the `A3DTopoContextData` structure. (See Parse topology body data.) The adjustment is calculated by dividing the targeted tolerance by the Body Scale.
   - The third argument (`bUseSameParameterization`) specifies that the conversion from the surface data to Nurbs data should use the parameterization data (if any) that is already specified in the face's surface definition.
   - The fourth argument (`&sNurbsData`) is a pointer to an empty structure in which the function stores the Nurbs data.

```
ASBool bUseSameParameterization = TRUE;
A3DSurfNurbsData sNurbsData;
A3D_INITIALIZE_DATA(sNurbsData);
iRet = A3DSurfBaseGetAsNurbs(sData.m_pSurface, 1e-3 / stdContextScale,
   bUseSameParametrization, &sNurbsData);
```

3. Determine the result of the `A3DSurfBaseGetAsNurbs` function. In addition to returning the standard success and failure values, this function can also return the warning

`A3D_SRF_NURBS_CANNOT_KEEP_PARAMETERIZATION`. This warning indicates that the conversion yielded a valid Nurbs surface but that associated space parametric trimming curves may be unreliable.

```
if (iRet == A3D_SUCCESS ||
   iRet == A3D_SRF_NURBS_CANNOT_KEEP_PARAMETERIZATION) {
   …
}
```

4. If Step 8 did not return with an error, delete the `A3DSurfNurbsData` structure by invoking the `A3DSurfNurbsGet` function with the first argument set to `NULL` and the second argument set to the location of the structure (`&sNurbsData` in the following example).

```
A3DSurfNurbsGet(NULL, &sNurbsData);
```

Parsing PRC entities that specify graphics

The Acrobat 3D API Reference groups the PRC entities that specify graphics into the graphic module. The graphics module includes the following categories of PRC entities:

- **Graphics entities:** Specify attributes that can apply to any graphic entity, such as color, line pattern, and coordinate system transformation.
- **Camera entities:** Specify the position and direction from which a scene is viewed.
- **Light entities:** Specify the quality of light applied to a 3D scene.
- **SceneDisplay:** Specify viewing characteristics for a 3D representation, such as camera, lights, plane clipping, and background style.

[DG says: "parsing graphic PRC entities". Explanation of Graphic Entities is very strange, and I cannot understand its meaning. These entities are very precise, and define precise things!!!!]

Some characteristics of PRC entities that specify graphics can be accessed only through the Acrobat 3D API *miscellaneous cascaded attributes*. These characteristics include the entity's layer, coordinate system, and style attributes. Style attributes include line pattern index, RGB color index, and transparency setting. (See Parsing attributes that appear in an entity base.)

Parsing attributes that appear in an entity base

This section explains how to obtain data from the root-base entity or from the miscellaneous cascaded attributes.

The Acrobat 3D Library uses a parallel structure for representing attributes that apply to PRC entities that have a particular PRC entity base. Data that is specific to the entity is defined in `A3D`*`EntityName`*`Data` structures (for example the `A3DAsmPartDefinitionData` structure). These structures are used to create or parse the entity. Other data is defined in entities specific to the global or base types. (See the Related Pages on the Acrobat 3D API Reference.)

Parsing root-base entity data

This section explains how to parse data stored in the root base for a PRC entity. The root base data applies to all PRC entities.

Parse the root-base entity data

5. Declare and initialize a `A3DRootBaseData` structure.

```
A3DRootBaseData sData;
A3D_INITIALIZE_DATA(sData);
```

6. Populate the `A3DRootBaseData` structure by invoking the `A3DRootBaseGet` function.

```
ASInt32 iRet = A3DRootBaseGet(p, &sData);
```

7. Save the value of the `m_pcName` to your export structure.

8. For each `A3DMiscAttribute` entity referenced by the array that is referenced by `m_ppAttributes`, process the contents. (See Parse the miscellaneous attribute entity.)

9. Delete the `A3DRootBaseData` structure.

```
A3DRootBaseGet(NULL, &sData);
```

Parse the miscellaneous attribute entity

10. Declare and initialize an `A3DMiscAttributeData` structure.

11. Populate the `A3DMiscAttributeData` structure by invoking the `A3DMiscAttributeGet` function. The first argument references the `A3DMiscAttribute` entity (from root base data `m_ppAttributes` member), and the second argument references the `A3DMiscAttributeData` being populated.

12. Determine whether the root data contains modeler data by checking the `m_pSingleAttributesData` member. If its value is non-null, save the modeler data to your export structure.

13. Save the value of the `m_pcName` attribute to your export structure.

14. Delete the `A3DMiscAttributeData` structure created in Step 10.

Parsing graphic attributes using miscellaneous cascaded attributes

Miscellaneous cascaded attributes manage the inheritable graphics data that can be applied to a PRC entity. Such inheritable data includes the show and remove settings, style attributes such as color and pattern, layer attributes, and coordinate system transformations.

The `A3DGraphicsCreate` function lets you set the graphics data for an entity. It also lets you set inheritance behavior for the attributes. These bits can specify that an attribute be inherited from a child entity or from a parent entity.

The `A3DGraphicsGet` function gets the graphics data for an entity, but it does not take into account inherited settings.

The following diagram illustrates the steps required to create and push miscellaneous cascaded attributes, depending on whether the entity represents a tessellation face.

To create and push a miscellaneous cascaded attributes structure

1. Declare the structure for creating and pushing a cascaded attributes structure.

2. Create and push a cascaded attributes structure. For entities in the tessellation module, include a pointer to the tessellation base entity (see Create and push a cascaded attributes structure for entities in the tessellated module. For all other entities in the graphics module, see Create and push a cascaded attributes structure for graphic entities.

3. Delete the structure created for the miscellaneous cascaded attributes.

Declare the structure for creating and pushing a cascaded attributes structure

1. Declare an empty cascaded data structure in which the cascaded attributes are to be stored. The macro `A3D_INITIALIZE_DATA` clears the memory allocated to the structure and checks the size of the structure to avoid alignment problems.

```
A3DMiscCascadedAttributesData sAttrData;
A3D_INITIALIZE_DATA(sAttrData);
```

2. Declare a null pointer to a miscellaneous cascaded attributes object.

```
A3DMiscCascadedAttributes* pAttr;
```

Create and push a cascaded attributes structure for graphic entities

1. Create an empty miscellaneous cascaded attributes object by invoking the `A3DMiscCascadedAttributesCreate` function, providing as an argument the address of the previously created null pointer.

```
ASInt32 iRet = A3DMiscCascadedAttributesCreate(&pAttr);
```

2. Invoke the `A3DMiscCascadedAttributesPush` function, which creates the `A3DMiscCascadedAttributes` object for the current PRC entity and pushes the parent `A3DMiscCascadedAttributes` object onto the stack. Provide the following arguments to `A3DMiscCascadedAttributesPush`.
   - The first argument (`pAttr` in the following example) is a pointer to an empty miscellaneous cascaded attribute object. The `A3DMiscCascadedAttributesPush` function populates this object.
   - The second argument (`pBase` in the following example) is a pointer to the PRC entity of interest, recast as a `const A3DRootBaseWithGraphics*`.
   - The third argument (`pFatherAttr` in the following example) is a pointer to the parent entity's `A3DMiscCascadedAttributes` object. Do not confuse this structure with the parent entity's `A3DMiscCascadedAttributesData` structure.

   Here is an example:

   ```
   iRet = A3DMiscCascadedAttributesPush (pAttr, pBase , pFatherAttr);
   ```

3. Get the data from the newly created cascaded attributes structure by invoking the `A3DMiscCascadedAttributesGet` function. Provide the following arguments:
   - The first argument (`pAttr` in the following example) references the cascaded attribute structure populated in Step 2.
   - The second argument (`sAttrData` in the following example) references the empty `A3DMiscCascadedAttributesData` structure created previously. (See Declare the structure for creating and pushing a cascaded attributes structure.) The `A3DMiscCascadedAttributesGet` function stores the entity's attributes in this structure.

   Here is an example:

```
iRet = A3DMiscCascadedAttributesGet(pAttr, &sAttrData);
```

Create and push a cascaded attributes structure for entities in the tessellated module

1. Invoke the `A3DMiscCascadedAttributesPushTessFace` function, providing the following arguments:

- The first argument (`pAttr` in the following example) is a pointer to an empty miscellaneous cascaded attribute object. The `A3DMiscCascadedAttributesPushTessFace` function populates this object.
- The second argument (`pRepItem` in the following example) is a pointer to the representation item that contains the tessellation face.
- The third argument (`pTessBase` in the following example) is a pointer to the tessellation entity that contains the face, recast as a `const A3DTessBase*` pointer. This argument can be cast from the `A3DTess3D` entity.
- The fourth argument (`psTessFaceData` in the following example) references an array of pointers to the `A3DTessFaceData` structures that contain the face data. Use the `A3DTess3DGet` function to obtain that structure.
- The fifth argument (`uiFaceIndex` in the following example) is the face index used with the fourth argument to access the face data from the `m_psFaceTessData` member of the `psTess3DData` structure.
- The sixth argument (`pFatherAttr` in the following example) is a pointer to the parent entity's `A3DMiscCascadedAttributes` object. Do not confuse this structure with the parent entity's `A3DMiscCascadedAttributesData` structure.

Here is an example:

```
ASInt32 iRet = A3DMiscCascadedAttributesCreate(ppAttr);
// This API is dedicated to tessellation
iRet = A3DMiscCascadedAttributesPushTessFace(pAttr, pRepItem, pTessBase,
   psTessFaceData, uiFaceIndex, pFatherAttr);
A3D_INITIALIZE_DATA((*psAttrData));
iRet = A3DMiscCascadedAttributesGet(*ppAttr, psAttrData);
```

2. Save relevant data to your export structure.

Delete the structure created for the miscellaneous cascaded attributes

1. Delete the `A3DMiscCascadedAttributes` structure by invoking the `A3DMiscCascadedAttributesDelete` function.

```
A3DMiscCascadedAttributesDelete(pAttr);
```

2. Delete the `A3DMiscCascadedAttributesData` structure by invoking the `A3DMiscCascadedAttributesGet` function, setting the first argument to `NULL` and the second to the address of the structure. Here is an example:

```
A3DMiscCascadedAttributesGet(NULL, &sAttrData);
A3DMiscCascadedAttributes* pAttr;
```

Terminating the interface with the Acrobat 3D API

When all parsing is complete, you must delete the in-memory model of the PRC, de-allocate any other memory allocations, and terminate the Acrobat 3D API.

Terminate your interface with the Acrobat 3D API

1. Delete the model file in memory. This model was created when the `A3DAsmModelFileLoadFromFile` function was called.

```
A3DAsmModelFileDelete(p)
```

2. Ensure that all memory is de-allocated.

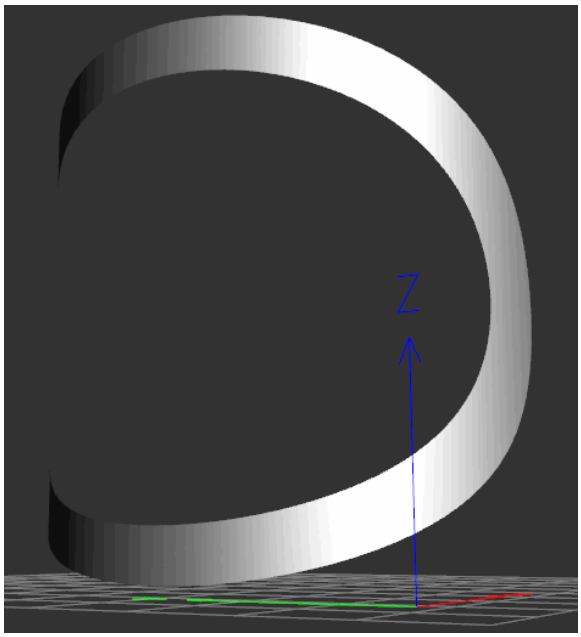
3. Terminate the Acrobat 3D API by invoking the `A3DDllTerminate` function, as follows:

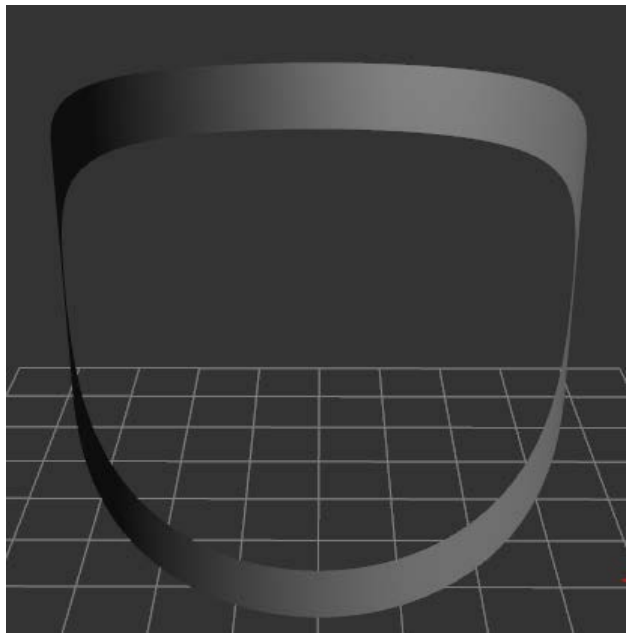
```
A3DDllTerminate();
```

Creating a PRC file that uses boundary representation

The Acrobat 3D SDK lets you create a representation item for a boundary representation that is optionally accompanied by tessellation data. Boundary representations with or without tessellation data can represent curves, or open or closed solids. (See Creating a tessellation entity for representing faceted objects.)

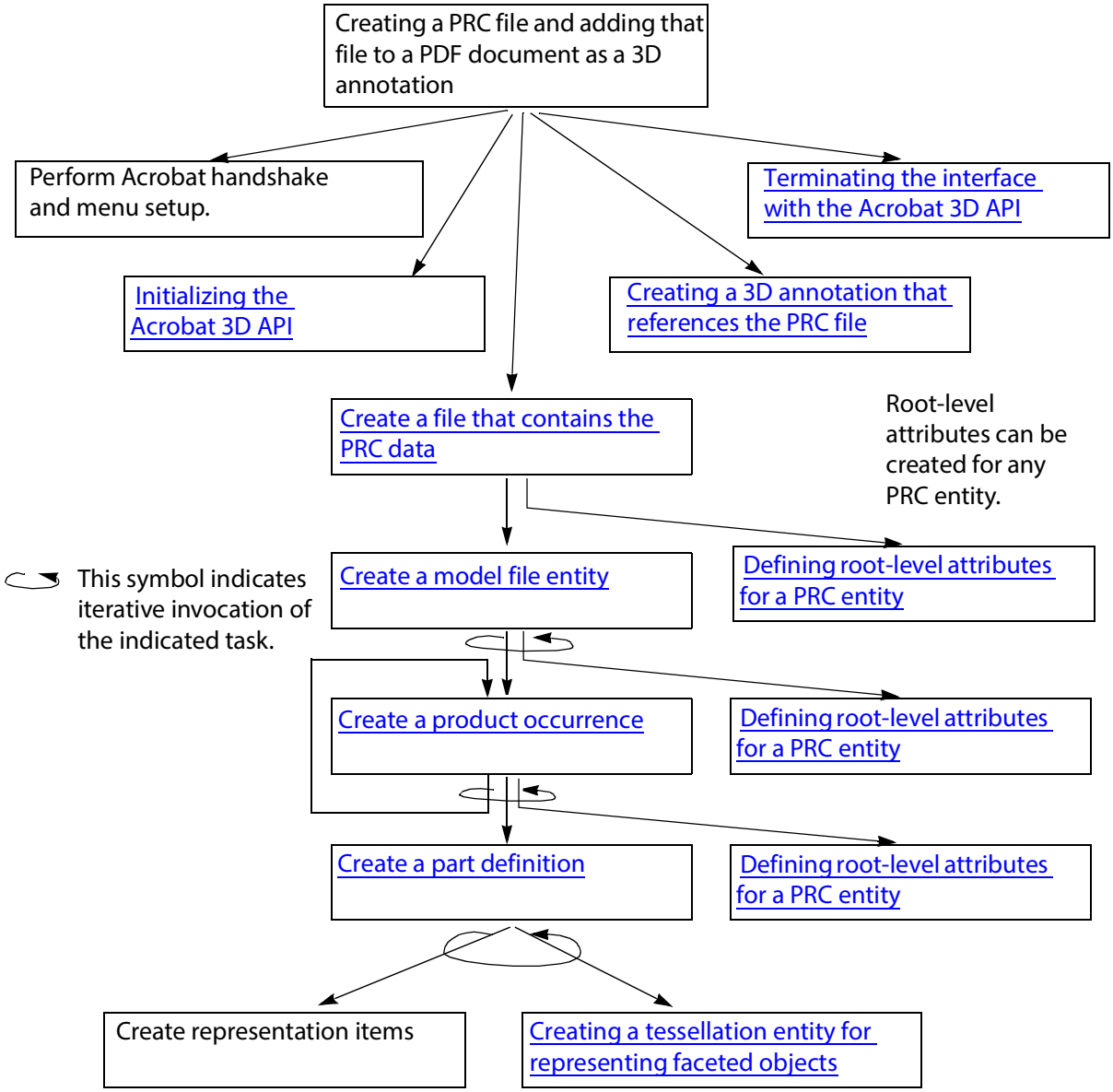
This section explains how to create a PRC file that represents a circular band (a co-edge) that is UV-mapped to a clear cylinder surface (pictured below). More specifically, the solid in this 3D image contains a single face, which in turn contains a cylinder and a loop. The loop contains a single co-edge that contains two circular curves. The co-edge is UV-mapped to the cylinder. The following images show the resulting image from different perspectives. To open a PDF document that contains the PRC content pictured below, click here.





The following diagram shows the tasks required to create each PRC entity to represent the image pictured above. This diagram is intended to portray a bottom-up process, where upper-level functions are not complete until their lower-level functions are completed. That is, the most basic PRC entities (such as vertices, edges and coedges) are created first, followed by the more complex PRC entities that consist of those basic PRC entities and other data. The last PRC entity to be created is the model file.

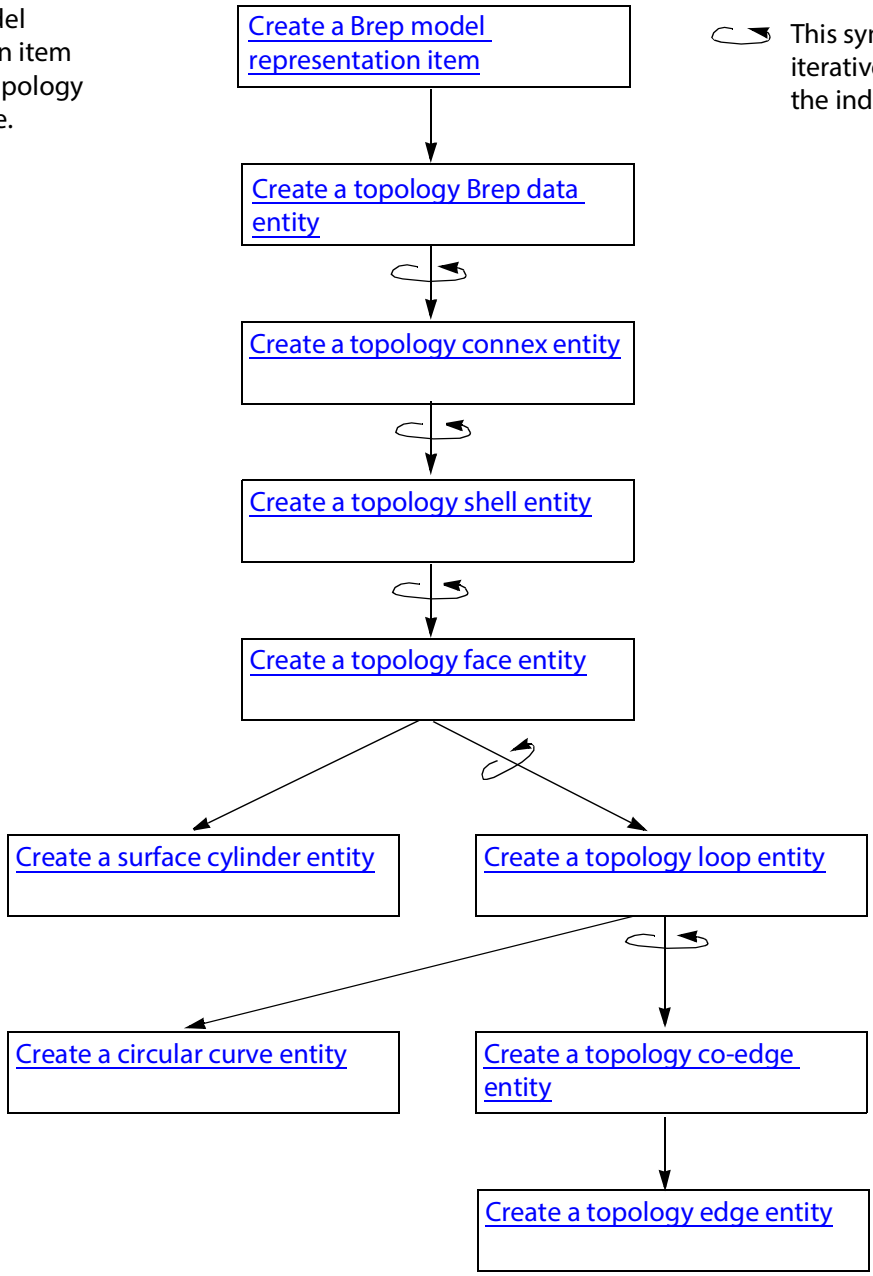
Structural diagram for creating a PRC file



This diagram is continued in Structural diagram for creating a Brep model representation within a part definition entity.

Structural diagram for creating a Brep model representation within a part definition entity

The Brep model representation item defines the topology for a 3D shape.



Handling errors

For information about handling errors, see Handling errors.

Creating a model file entity and exporting it to a physical file

This section describes how to initiate the process of creating a model file and how to export that data to a physical file.

Create a file that contains the PRC data

1. Create a pointer that references a populated model file entity. (See Create a model file entity.) In the following example, `createModelFile` is a private function that returns a pointer to a populated model file.

```
A3DAsmModelFile* pModelFile = createModelFile();
```

2. Create the model file entity. (See Create a model file entity.)

3. Save the contents of the model file to a physical file by invoking the `A3DAsmModelFileWriteToFile` function.

```
if (p != NULL) {
   if (_access(pcPRCName, 00) != -1)
_unlink(pcPRCName);
   ASInt32 iRet = A3DAsmModelFileWriteToFile(p, NULL, pcPRCName);
```

4. Delete the model file populated in Step 1.

```
A3DAsmModelFileDelete(pModelFile);
```

Note: The Acrobat 3D API functions in this chapter are described as though implicit linking were possible. Wherever you see a direct call to one of these functions, you should instead use the pointer to the function. See Implementing external linking.

Creating structure PRC entities

This section explains how to create the PRC entities that represent the structure of the PRC file, such as the model file, product occurrences, and part definitions. The names of structure module entities have the form `A3DAsm`*`Entity_name`*.

Create a model file entity

5. If the model file references multiple product occurrences, create an array to accommodate a pointer that references each product occurrence.

6. For each product occurrence in the model file, create a pointer that references a populated product occurrence entity. (See Create a product occurrence.) In the following example, `createOccurrence` is a private function that returns a pointer to a populated product occurrence.

```
A3DAsmProductOccurrence* p = createOccurrence();
```

7. Declare and initialize a model file data structure and set the values of its members, as follows:

   - `m_uiPOccurrencesSize` represents the number of child product occurrences.
   - `m_dUnit` is a multiple of millimeters that specifies the units used by the 3D data in the structure element.
   - `m_ppPOccurrences` references the array of product occurrence pointers.

```
A3DAsmModelFileData sData;
A3D_INITIALIZE_DATA(sData);
sData.m_uiPOccurrencesSize = 1;
sData.m_dUnit = 1.0;
sData.m_ppPOccurrences = &p;
```

8. Encapsulate the model file data by invoking the `A3DAsmModelFileCreate` function. The first argument is a pointer to the model file data, and the second argument is a pointer to the model file entity.

```
ASInt32 iRet = A3DAsmModelFileCreate(&sData, &pModelFile);
```

9. Optionally, associate the model file entity with root-level attributes (see Defining root-level attributes for a PRC entity).

Create a product occurrence

1. Declare a null pointer to a product occurrence.

```
A3DAsmProductOccurrence* pProductOccurrrence = NULL;
```

2. Create a pointer to a populated part definition. (See Create a part definition.) In the following example, `createPart` is a private function that returns a pointer to a populated part definition entity.

```
A3DAsmPartDefinition* p = createPart();
```

3. Set the `m_pPart` member to reference the part created in Step 2.

```
A3DAsmProductOccurrenceData sData;
A3D_INITIALIZE_DATA(sData);
sData.m_pPart = p;
```

4. Package the product occurrence data as a PRC entity by invoking the `A3DAsmProductOccurrenceCreate` function. The first argument is a pointer to the product occurrence data, and the second argument is an indirect pointer to the product occurrence.

```
ASInt32 iRet =
   A3DAsmProductOccurrenceCreate(&sData, &pProductOccurrence);
```

5. Define the cascaded attributes for the product occurrence. (See Defining root-level attributes for a PRC entity.)

Create a part definition

1. Declare a null pointer to a part definition.

```
A3DAsmPartDefinition* pPartDefinition = NULL;
```

2. Declare and initialize a part definition data structure, as shown in the following example.

```
A3DAsmPartDefinitionData sData;
A3D_INITIALIZE_DATA(sData);
```

3. For each representation item referenced by this part definition, create a pointer to that representation item and populate it. (See Create a Brep model representation item.) In the following example, `createRIBrep` is a private function that returns a pointer to a populated Brep representation item.

```
A3DRiRepresentationItem* p = createRIBrep();
```

4. Set the `m_uiRepItemsSize` member of the part definition to the number of representation items.

5. Set the `m_ppRepItems` member of the part definition to the array of pointers for the representation items. The following example describes a part definition that references a single representation item:

```
sData.m_uiRepItemsSize = 1;
sData.m_ppRepItems = &p;
```

6. Package the part definition data as a PRC entity by invoking the `A3DAsmPartDefinitionCreate` function. The first argument is a pointer to the product occurrence data, and the second is the pointer to the part definition created in Step 1.

```
ASInt32 iRet = A3DAsmPartDefinitionCreate(&sData, &pPartDefinition);
```

Creating representation item PRC entities

The Acrobat 3D API Reference groups the PRC entities that specify individual objects present in the CAD file into the representation item module. Representation items define a particular aspect of the geometric data. The Brep model representation item is one of the PRC entities available for packaging distinct 3D objects. Some of the other representation items include set, point set, poly Brep model, and polywire.

Create a Brep model representation item

1. Declare a pointer to a Brep model entity.

```
A3DRiBrepModel* pBrepModel = NULL;
```

2. Create a pointer to a populated topology Brep data entity. (See Create a topology Brep data entity.) The `createTopoBrep` function in the following example is a private function that returns a reference to a populated topology Brep data entity.

```
A3DTopoBrep* p = createTopoBrep();
```

3. Declare and initialize a Brep model data structure, and set its member values. In the following example, the `m_pBrepData` member references the Brep data entity created in the previous step, and the `m_bSolid` member is set to `FALSE`, indicating that the Brep model is a shell.

```
A3DRiBrepModelData sData;
A3D_INITIALIZE_DATA(sData);
sData.m_pBrepData = p;
sData.m_bSolid = FALSE;
```

4. Package the Brep model data as a PRC entity by invoking the `A3DRiBrepModelCreate` function. The first argument is a pointer to the Brep data structure, and the second is a pointer to the Brep model created in Step 1.

```
ASInt32 iRet=A3DRiBrepModelCreate(&sData, &pBrepModel);
```

Creating topology PRC entities

The Acrobat 3D API Reference groups the PRC entities that specify topology into the topology module. The topology module uses a hierarchy of faces to define the shapes of 3D objects.

Create a topology Brep data entity

1. Create a null pointer to a topology Brep data entity.

```
A3DTopoBrepData* pTopoBrepData = NULL;
```

Tip: Do not confuse the `A3DTopoBrepData` entity with the `A3DTopoBrepDataData` structure. The former references a PRC entity (an opaque type), while the latter reveals the data in that entity.

2. For each topology connex data in the Brep data entity, create a pointer that references a populated topology connex entity. (See Create a topology connex entity.) In the following example,

`createTopoConnex` is a private function that returns a reference to a populated topology connex entity.

```
A3DTopoConnex* p = createTopoConnex();
```

3. If the Brep representation item references multiple topology connex entities, allocate memory for an array of references to the topology connexes in the Brep data entity.

4. Declare and initialize a `A3DTopoBrepDataData` structure. Set the `m_ppConnexes` member to reference the array of topology connex entities, and set the `m_uiConnexSize` member to the number of entries in the array.

```
A3DTopoBrepDataData sData;
A3D_INITIALIZE_DATA(sData);

sData.m_ppConnexes = &p;
sData.m_uiConnexSize = 1;
```

5. Package the `A3DTopoBrepDataData` structure as a PRC entity by invoking the `A3DAsmPartDefinitionCreate` function. The first argument is a pointer to the Brep-data data structure (`A3DTopoBrepDataData`), and the second is the pointer to the Brep data entity created in Step 1.

```
ASInt32 iRet = A3DTopoBrepDataCreate(&sData, &pTopoBrepData);
```

Create a topology connex entity

1. Declare a pointer to a topology connex entity.

```
A3DTopoConnex* pTopoConnex = NULL;
```

2. For each topology shell in the topology connex entity, create a pointer that references a populated topology shell entity. (See Create a topology shell entity.) In the following example, `createTopoShell` is a private function that returns a reference to a populated topology shell entity.

```
A3DTopoShell* p = createTopoShell();
```

3. If the topology connex entity references multiple topology shell entities, allocate memory for an array to accommodate each topology shell entity pointer.

4. Declare and initialize an `A3DTopoConnexData` structure. Set the `m_ppShells` member to reference the array that contains the pointers, and set the `m_uiShellSize` member to the number of pointers.

```
A3DTopoConnexData sData;
A3D_INITIALIZE_DATA(sData);

sData.m_ppShells = &p;
sData.m_uiShellSize = 1;
```

5. Package the topology connex data as a PRC entity by invoking the `A3DTopoConnexCreate` function. The first argument is a pointer to the Brep-data data structure (`A3DTopoBrepDataData`), and the second is the pointer to the topology connex entity created in Step 1.

```
ASInt32 iRet = A3DTopoConnexCreate(&sData, &pTopoConnex);
```

Create a topology shell entity

1. Declare a null pointer to a topology shell entity.

```
A3DTopoShell* pTopoShell = NULL;
```

2. If the topology shell references multiple topology face entities, create an array to accommodate each pointer.

3. For each topology face in the topology shell, create a pointer that references a populated topology face entity. (See Create a topology face entity.) In the following example, `createTopoFace` is a private function that returns a reference to a populated topology shell entity.

```
A3DTopoFace* p = createTopoFace();
```

4. Declare and initialize an `A3DTopoShellData` structure. Set the `m_ppFaces` member to reference the array of topology face entities, and set the `m_uiFaceSize` member to the number of topology shell face entities.

```
A3DTopoShellData sData;
A3D_INITIALIZE_DATA(sData);

sData.m_ppFaces = &p;
sData.m_uiFaceSize = 1;
```

5. Set the `A3DTopoShellData` structure's `m_pucOrientationWithShell` member to reference an ASUns8 integer that specifies the orientation of the surface normal with respect to the shell normal. If the shell is closed and otherwise arbitrary, the shell normal points outside the material. The following example specifies that the surface normal has the same orientation as the face and shell.

   Note: Despite the simplicity of the values it expresses, the `m_pucOrientationWithShell` is a pointer to an ASUns8 integer; it is not itself an ASUns8 integer.

```
ASUns8 orient = 1;
sData.m_pucOrientationWithShell = &orient;
```

6. Package the topology shell data as a PRC entity by invoking the `A3DTopoShellCreate` function. The first argument is a pointer to the topology shell data, and the second is the pointer to the topology shell entity created in Step 1.

```
ASInt32 iRet = A3DTopoShellCreate(&sData, &pTopoShell);
```

Create a topology face entity

The example described here UV-maps onto a cylinder surface a topology loop representing a circular band.

1. Declare a null pointer to a topology face entity.

```
A3DTopoFace* pTopoFace = NULL;
```

2. If the topology face references multiple loop entities, declare an array to accommodate each pointer.

3. For each loop in the topology face, create a pointer that references a populated loop entity. (See Create a topology loop entity.) In the following example, `createTopoLoop` is a private function that returns a reference to a populated loop entity.

```
A3DDouble outerradius = 0.5;
A3DDouble innerradius = 0.4;
A3DTopoLoop* loops[2];
loops[0] = createTopoLoop(outerradius);
loops[1] = createTopoLoop(innerradius);
```

4. Declare and initialize an `A3DTopoFaceData` structure. Set the `m_ppLoops` member to reference the array of loop entity pointers, and set the `m_uiLoopSize` member to the number of loop entities in the array. Set the `m_uiOuterLoopIndex` member to the array index of the topology loop that describes the outer loop. If unknown, set that member to `A3D_LOOP_UNKNOW_OUTER_INDEX`.

   In the following example, `m_uiOuterLoopIndex` indicates that the outer loop is the first entry in the `loops` array.

```
A3DTopoFaceData sData;
A3D_INITIALIZE_DATA(sData);
sData.m_ppLoops = loops;
sData.m_uiLoopSize = 2;
sData.m_uiOuterLoopIndex = 0;
```

5. Declare a pointer that references a populated surface base entity. (See Create a surface cylinder entity.) Set the `m_pSurface` member of the `A3DTopoFaceData` structure to reference the populated surface. In the following example, `createSurface` is a private function that returns a reference to a populated cylinder surface entity. The returned value is cast as a pointer to a surface base.

```
A3DSurfBase * p = createSurface();
sData.m_pSurface = p;
```

6. Package the topology face data as a PRC entity by invoking the `A3DTopoFaceCreate` function. The first argument is a pointer to the topology face data, and the second is the pointer to the topology face entity created in Step 1.

```
ASInt32 iRet = A3DTopoFaceCreate(&sData, &pTopoFace);
```

Create a topology loop entity

A loop is a connected series of co-edges and describes the boundary of a face. Generally, loops are closed, having no start or end point.

1. Declare a null pointer to a topology loop entity.

```
A3DTopoLoop* pTopoLoop = NULL;
```

2. If the topology loop references multiple co-edge entities, declare an array to accommodate each pointer.

3. For each co-edge in the loop, do the following:
   - Create a pointer that references a populated circle entity with a given radius. (See Create a circular curve entity.)
   - Create a pointer that references a populated co-edge entity with a given curve. (See Create a topology co-edge entity.)

     In the following example, `createCircle` is a private function that takes a radius and returns a reference to a populated circular curve entity, and `createTopoCoEdge` is a private function that takes a pointer to a circle entity and returns a reference to a populated co-edge entity. The circle-entity describes a curve.

```
A3DCrvBase* p = createCircle(radius);
A3DTopoCoEdge* q = createTopoCoEdge(p);
```

4. Declare and initialize an `A3DTopoLoopData` structure. Set the `m_ppCoEdges` member to reference the array of co-edge entity pointers, and set the `m_uiCoEdgeSize` member to the number of entities in the array.

```
A3DTopoLoopData sData;
A3D_INITIALIZE_DATA(sData);
sData.m_ppCoEdges = &q;
sData.m_uiCoEdgeSize = 1;
```

5. Set the `A3DTopoLoopData` structure's `m_ucOrientationWithSurface` member to specify the orientation of the loop relative to the surface normal vector. The following example specifies that the orientation is perpendicular to the surface normal vector (or parallel with the surface).

```
sData.m_ucOrientationWithSurface = 1;
```

6. Package the topology face data as a PRC entity by invoking the `A3DTopoLoopCreate` function. The first argument is a pointer to the topology loop data, and the second is the pointer to the topology loop entity created in Step 1.

```
ASInt32 iRet=A3DTopoLoopCreate(&sData, &pTopoLoop);
```

Create a topology co-edge entity

A coedge is a directed edge. The two co-edges related to an edge point in the same or opposite directions along the edge. Each co-edge is associated with a loop of one of the two neighboring/adjacent faces.

1. Create a pointer to a populated `A3DTopoEdge` entity. (See Create a topology edge entity.)

2. Declare and initialize a topology co-edge data entity.

3. Set the `m_pUVCurve` member of the data entity to reference the circle entity created earlier (see Create a topology loop entity).

   Set the `m_pEdge` member to the edge entity created in Step 1.

   Note: To set the value of the `m_pUVCurve` member using meaningful data, you need your own modeler.

```
A3DTopoCoEdgeData sData;
A3D_INITIALIZE_DATA(sData);
sData.m_pUVCurve = p;
sData.m_pEdge = q;
```

4. Set the `m_ucOrientationWithLoop member to indicate the relative orientation of the loop.`

```
sData.m_ucOrientationWithLoop = 1; /* Same orientation as the adjacent co-edge
sData.m_ucOrientationUVWithLoop = 1;
```

5. Package the co-edge data as a PRC entity by invoking the `A3DTopoEdgeCreate` function. The first argument is a pointer to the topology loop data, and the second is the pointer to the topology edge entity created in Step 1.

```
ASInt32 iRet = A3DTopoCoEdgeCreate(&sData, &pp);
```

Create a topology edge entity

1. Declare a null pointer to a topology edge entity.

```
A3DTopoEdge* pTopoEdge = NULL;
```

2. Declare and initialize a topology edge data structure.

```
A3DTopoEdgeData sData;
A3D_INITIALIZE_DATA(sData);
```

3. Populate the members with data.

4. Package the topology face data as a PRC entity by invoking the `A3DTopoEdgeCreate` function. The first argument is a pointer to the topology loop data, and the second is the pointer to the topology edge entity created in Step 1.

```
ASInt32 iRet = A3DTopoEdgeCreate(&sData, &pTopoEdge);
```

Creating geometry PRC entities

The Acrobat 3D API Reference describes the PRC entities that specify geometry in the Geometry module. This module contains these submodules:

Curves: Entities that represent different kinds of curves, such as NURBS, elliptic, and helical.

Surfaces: Entities that represent different kinds of surfaces, such as NURBS, spheric, and toric.

Common Structures for Geometric Entities: Entities that supplement data in the Curves and Surfaces modules.

This section describes the creation of a cylinder surface on which a circular co-edge is UV-mapped.

A circular curve includes a coordinate system and parameterization settings. By default, the reference domain is [0, 2PI]. A circular curve uses an `A3IntervalData` entity as the linear domain and an `A3DMiscCartesianTransformationData` structure as the coordinate system. New parameterization is computed from two coefficients applied to the reference parameterization [0, 2PI].

Create a surface cylinder entity

1. Declare a null pointer to an `A3DSurfCylinder` entity.

```
A3DSurfCylinder* pSurfCylinder = NULL;
```

2. Declare and initialize a surface cylinder data structure, and initialize the structures it contains.

```
A3DSurfCylinderData sData;
A3D_INITIALIZE_DATA(sData);
A3D_INITIALIZE_DATA(sData.m_sParam);
A3D_INITIALIZE_DATA(sData.m_sTrsf);
```

3. Set the UV-parameterization data of the surface cylinder data structure, which specifies how the circular curve is mapped to the cylinder. The following example specifies the trimming and orientation of the mapping:

   - Minimum and maximum extents in the UV domain (`sData.m_sParam.m_sUVDomain`) establish the trimming contour in the final parameterization space. Each extent is a 2D vector. The following example clips the texture represented by the co-edge to the area described by [0.0, 0.0] and [360.0 40.0].
   - Coefficients establish parameterization along the U and V axes. In this example, U values are expressed as inverted degrees (1.0/degrees). V values are expressed as inverted lengths (1.0/length).

```
sData.m_sParam.m_sUVDomain.m_sMin.m_dX = 0.0;
sData.m_sParam.m_sUVDomain.m_sMin.m_dY = 0.0;
sData.m_sParam.m_sUVDomain.m_sMax.m_dX = 360.0;
```

```
sData.m_sParam.m_sUVDomain.m_sMax.m_dY = 40.0;
// Parameters go from -1 to +1 in both directions
sData.m_sParam.m_dUCoeffA =
   1.0 / (sData.m_sParam.m_sUVDomain.m_sMax.m_dX / 2);
sData.m_sParam.m_dUCoeffB = -1.0;
sData.m_sParam.m_dVCoeffA =
   1.0 / (sData.m_sParam.m_sUVDomain.m_sMax.m_dY / 2);
   sData.m_sParam.m_dVCoeffB = -1.0;
sData.m_sParam.m_bSwapUV = FALSE;
sData.m_sTrsf.m_ucBehaviour = kA3DTransformationIdentity;
```

4. Set other member values of the surface cylinder data structure. In the following example, the radius of the cylinder is set to 10.0. The unit of measure (as multiples of millimeters) was previously set for the model file entity. (See Create a model file entity.)

```
sData.m_dRadius = 10.0;
```

5. Package the cylinder data as a PRC entity by invoking the `A3DSurfCylinderCreate` function. The first argument is a pointer to the surface cylinder data, and the second is the pointer to the surface cylinder entity created in Step 1.

```
ASInt32 iRet = A3DSurfCylinderCreate(&sData, &pSurfCylinder);
```

Create a circular curve entity

1. Declare a null pointer to a circular curve entity.

```
A3DCrvCircle* pCrvCircle = NULL;
```

2. Declare and initialize a circular curve data structure, and initialize the structures it contains.

```
A3DCrvCircleData sData;
A3D_INITIALIZE_DATA(sData);
A3D_INITIALIZE_DATA(sData.m_sParam);
A3D_INITIALIZE_DATA(sData.m_sTrsf);
```

3. Set the values of the circular curve data structure. The following example sets the radius of the circle to a value originally specified during loop creation. (See Create a topology face entity.)

```
sData.m_dRadius = radius;
```

4. Set the values of the parameterization structure within the circular curve data structure. In this example, the interval defines the circular arc as the entire circle and the co-efficients set the parameterization across the entire length of the circular arc.

```
sData.m_sParam.m_dCoeffA = 1.0;
sData.m_sParam.m_dCoeffB = 0.0;
sData.m_sParam.m_sInterval.m_dMin = 0.0;
sData.m_sParam.m_sInterval.m_dMax = 360;
```

5. Set the values of the Cartesian transformation structure within the circular curve data structure. The following example sets the behavior of the Cartesian transformation data to the identity transformation.

```
sData.m_sTrsf.m_ucBehaviour = kA3DTransformationIdentity;
```

6. Package the curve data as a PRC entity by invoking the `A3DCrvCircleCreate` function. The first argument is a pointer to the curve circle data, and the second is the pointer to the circular curve entity created in Step 1.

```
ASInt32 iRet = A3DCrvCircleCreate(&sData, &pCrvCircle);
```

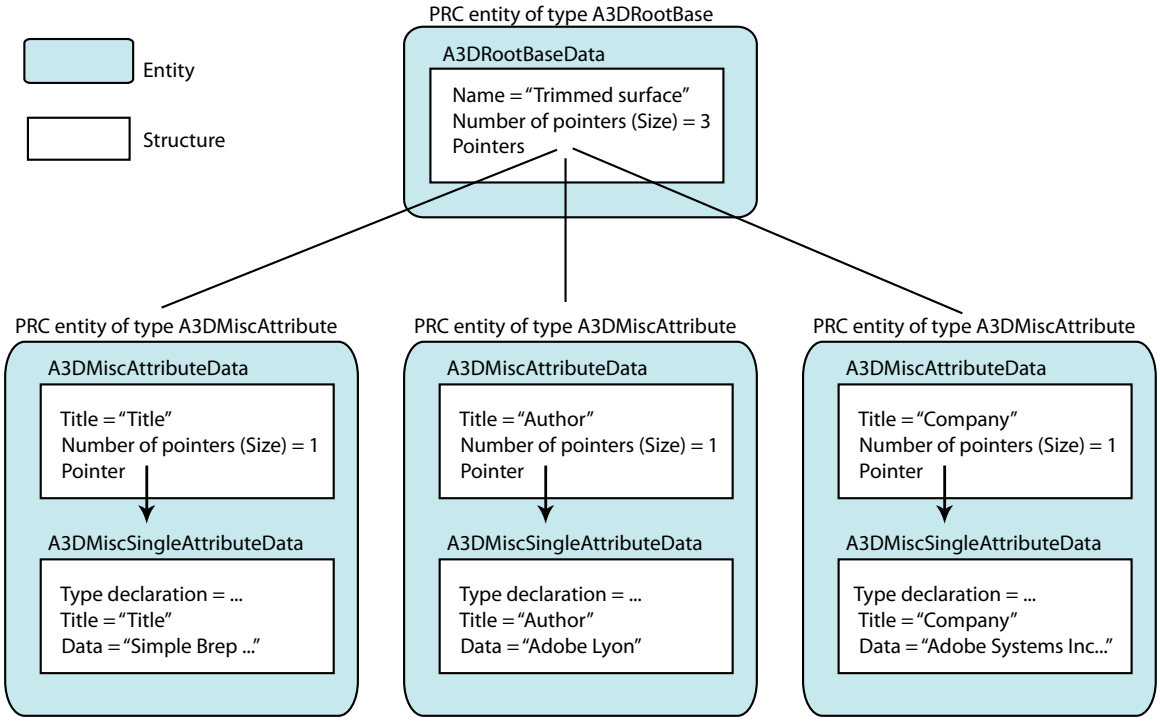
Defining root-level attributes for a PRC entity

The Acrobat 3D API lets you create a set of attributes named *root-level attributes* that can be applied to any PRC entity. These attributes are packaged as an `A3DRootBase` entity that is referenced from the PRC entity they describe.

- `A3DRootBaseData`. This structure has a name and can have pointers to multiple `A3DMiscAttribute` entities.
- `A3DMiscAttributeData`. This structure has a name and can reference a single `A3DMiscSingle` structure.
- `A3DMiscSingleAttributeData`. This structure is used for modeler data, such as string or time. It has a title, a type designation, and data. The name in the parent `A3DMiscAttributeData` must match the title in the `A3DMiscSingleAttributeData`. This structure is terminal.

Note: See the Acrobat 3D API Reference for limitations on including modeler data in the root-level attributes for a PRC entity.

Here is a diagram that shows the structure of the `A3DRootBase` entity that can be associated with any PRC entity. The `A3DMiscSingleAttributeData` structures are used only for modeler attributes. The sample code that comes after the following task descriptions exemplifies this structure.



Define miscellaneous attributes that omit modeler data

1. Determine the string to use at the top level of the attributes and (optionally) the strings to use at subordinate levels.

2. Declare an array to accommodate pointers to the `A3DMiscAttribute` entities that you will create in a subsequent step.

3. For each subordinate-level string, declare and initialize an `A3DMiscAttributeData` structure. Set the `m_pcTitle` member to the subordinate-level string. Leave the `m_uiSize` and `m_pSingleAttributesData` members set to 0 because this structure omits modeler data.

4. Package each `A3DMiscAttributeData` structure as an `A3DMiscAttribute` entity by invoking the `A3DMiscAttributeCreate` function. Save the entity pointer in the array created in Step 2.

5. Declare and initialize an `A3DRootBaseData` structure. Set its `m_pcName` member to the top-level string. Set the `m_ppAttributes` member to reference the array that contains pointers to the `A3DMiscAttribute` entities.

6. Invoke the A3DRootBaseSet function to package the `A3DRootBaseData` structure as an `A3DRootBase` entity associated with a PRC entity. The first argument in this call is a pointer to the entity, and the second is a pointer to the `A3DRootBaseData` structure.

Define miscellaneous attributes that include modeler data

1. Determine the strings to use for modeler data and the type represented by each string. Here are the possible types and the enumerations that identify them:
   - `kA3DModellerAttributeTypeInt` integer (32 bits)
   - `kA3DModellerAttributeTypeReal` floating point
   - `kA3DModellerAttributeTypeTime` integer (32 bits), which uses the same conventions as the `time_t` C datatype.
   - `kA3DModellerAttributeTypeString` UTF-8 character string

2. Determine the titles to use for each piece of modeler data.

3. Declare an array to accommodate pointers to the `A3DMiscAttributeData` entities that you will create in a subsequent step.

4. For each modeler attribute associated with the root base entity, do the following:
   - Declare and initialize an `A3DMiscSingleAttributeData` structure. Set the `m_eType` member to the type of the data, set the `m_pcData` member to a string that represents the data, and set the `m_pcTitle` member to a title that identifies the data.
   - Declare and initialize an `A3DMiscAttributeData` structure. Set the `m_pcTitle` member to the same title used in the `A3DMiscSingleAttributeData` structure. Set the `m_uiSize` member to 1, and the `m_pSingleAtttributesData` member to reference the `A3DMiscSingleAttributeData` structure.

   Note: Each `A3DMiscSingleAttributeData` structure that references another `A3DMiscSingleAttributeData` structure must reference only one such structure and must have the same title as that structure.

5. Package each `A3DMiscAttributeData` structure as an `A3DMiscAttribute` entity by invoking the `A3DMiscAttributeCreate` function. Save the entity pointer in the array created in Step 3.

6. Declare and initialize an `A3DRootBaseData` structure. (See Step 5 in Define miscellaneous attributes that omit modeler data.)

7. Package the `A3DRootBaseData` structure as an `A3DRootBase` entity. (See Step 6 in Define miscellaneous attributes that omit modeler data.)

The following sample code creates base root attributes that describe three sets of modeler data. For information about the relationships between the structures and entities that represent this data, see the diagram provided in Defining root-level attributes for a PRC entity.

```
A3DVoid setAttributes(A3DEntity* p)
{
   A3DMiscSingleAttributeData Single;
   A3D_INITIALIZE_DATA(Single);

   Single.m_eType = kA3DModellerAttributeTypeString;
   Single.m_pcTitle = "Title";
   Single.m_pcData = "Simple Brep building demonstration";

   A3DMiscAttributeData sAttribs;
   A3D_INITIALIZE_DATA(sAttribs);

   sAttribs.m_pcTitle = Single.m_pcTitle;
   sAttribs.m_pSingleAttributesData = &Single;
   sAttribs.m_uiSize = 1;
   ASInt32 iRet = A3DMiscAttributeCreate(&sAttribs, &pAttr[0]);

   Single.m_pcTitle = "Author";
   Single.m_pcData = "Adobe Lyon";
   sAttribs.m_pcTitle = Single.m_pcTitle;
   iRet = A3DMiscAttributeCreate(&sAttribs, &pAttr[1]);

   Single.m_pcTitle = "Company";
   Single.m_pcData = "Adobe Systems Incorporated";
   sAttribs.m_pcTitle = Single.m_pcTitle;
   iRet = A3DMiscAttributeCreate(&sAttribs, &pAttr[2]);

   A3DRootBaseData sRootData;
   A3D_INITIALIZE_DATA(sRootData);

   sRootData.m_pcName = "Trimmed surface";
   sRootData.m_ppAttributes = pAttr;
   sRootData.m_uiSize = 3;
   iRet = A3DRootBaseSet(p, &sRootData);
}
```

Creating a 3D annotation that references the PRC file

Within an existing PDF document, create a 3D annotation. (See "Creating annotations" on page 194.) Use the file created in "Creating a model file entity and exporting it to a physical file" on page 232to create the 3D stream dictionary. Set the 3D stream dictionary `Type` attribute to `3D` and the `Subtype` attribute to `PRC`, as shown in the following example.

```
CosObj attrObj = CosStreamDict(stm3D);
CosDictPutKeyString(attrObj, "Type",
   CosNewNameFromString(cosDoc, false, "3D"));
CosDictPutKeyString(attrObj, "Subtype",
   CosNewName(cosDoc, false, ASAtomFromString("PRC")));
```

```
CosDictPutKeyString(cosAnnot, "3DD", stm3D);
```

Creating a tessellation entity for representing faceted objects

This section explains how to add tessellation base data to an existing representation item. The Acrobat 3D SDK defines a dedicated type you can use to create faceted entities.

Create the tessellation base data

1. Create a `A3DTessBaseData` structure and set its member values to represent the facets. The following explanation assumes facets are defined as triangles, although you could define facets having more than three points.
   - `m_uiCoordSize`. Set this member to the number of coordinates required to represent all facets in the object. This value is calculated as 9 * `facetSize`. The 9 multiplier is the number of points in each facet (3) times the number of coordinates in each point (3). The `facetSize` variable represents the number of facets in the object.
   - `m_pdCoords`. Set this member to an array of *N* `A3DDouble` entries, where *N* is the value calculated for the `m_uiCoordSize` member.
2. Add coordinates for each vertex of each facet to the array referenced by the `m_pdCoords` member. The X, Y, and Z coordinates for each vertex are represented in sequential entries in the array, as shown in this example:

```
sTessBaseData.m_pdCoords[j++] = x;
sTessBaseData.m_pdCoords[j++] = y;
sTessBaseData.m_pdCoords[j++] = z;
```

Create the tessellation facet data

1. Create an `A3DTess3DData` structure and set its members to represent the facet normals:
   - `m_uiNormalSize`. Set this member to the size of the array required to represent the normals for all facets in the objects. This is calculated as 3 * `facetSize`. The 3 multiplier is the number of cordinates to represent each vector.
   - `m_pdNormals`. Set this member to an array of *N* `A3DDouble` entries, where *N* is the value calculated for the `m_uiNormalSize` member.
   - `m_uiTriangulatedIndexSize`. Set this member to the number of entries required to represent the indexes associated with each facet in the objects. Each facet can have one to four indexes that reference texture coordinates, one index that references a facet normal, and three indexes that reference vertices. Normals are read from the `m_pdNormals` array, and textures are read from the `m_pdTextureCoords` array.
   - `m_puiTriangulatedIndexes`. Set this member to an array of *N* `ASUns32` entries, where *N* is the number of entries in the `m_uiTriangulatedIndexSize` member. This array contains the indexes of the points (`m_pdCoords`), normals (`m_pdNormals`), and texture arrays for each face's triangulated representation. The entries in this array correspond to the contents in the `A3DTessFaceData` entity.
2. Add the indexes for each facet, considering the number of entries to represent each facet. Each facet contains the following sequential representations in this array:

- (Optional) Index of the first texture in the array referenced by the `m_pdTextureCoords` member. This entry can be followed by three more entries for additional texture indexes.
- Index of the facet normal in the array referenced by the `m_pdNormals` member.
- Index of the first facet vertex in the array referenced by the `m_pdCoords` member.
- Index of the second facet vertex in the array referenced by the `m_pdCoords` member.
- Index of the third facet vertex in the array referenced by the `m_pdCoords` member.

In the following example for STL data, `facetSize` is set to 0 for the first facet represented by the array and to 1 for the second facet in the array.

```
sTess3DData.m_puiTriangulatedIndexes[4 * facetSize] = i;
sTess3DData.m_puiTriangulatedIndexes[4 * facetSize + 1] = j;
sTess3DData.m_puiTriangulatedIndexes[4 * facetSize+2] = j + 3;
sTess3DData.m_puiTriangulatedIndexes[4 * facetSize+3] = j + 6;
```

3. Add the coordinates of each facet normal to the array referenced by the `m_pdNormals` member. The X, Y, and Z coordinates for each vector are represented in sequential entries in the array, as shown in this example:

```
sTess3DData.m_pdNormals[i++] = x;
sTess3DData.m_pdNormals[i++] = y;
sTess3DData.m_pdNormals[i++] = z;
```

4. For each face, create an `A3DTessFaceData` entity and set its member values as follows:
   - `m_usUsedEntitiesFlags`. Set this member to an enumeration that specifies the entities used in the current face tessellation. For example, the `kA3DTessFaceDataTriangleOneNormal` enumeration specifies that the face data is a triangle with a single normal facet.
   - `m_uiSizesTriangulatedSize`. Set this member to the number of entries in the array referenced by the `m_puiSizesTriangulated` member.
   - `m_puiSizesTriangulated`. Set this value to the repetition sequence applied to the `m_usUsedEntitiesFlags` member. A single entry with a value of 1 indicates that the first flag in `m_usUsedEntitiesFlags` applies to all face data.
   - `m_uiStartTriangulated`. Set this value to the starting index for the wire in the array of point indexes of the `A3DTess3DData` entity.

5. Set the `A3DTess3DData` structure's `m_psFaceTessData` member to reference an array that in turn references each face that applied to the facets. Set the `m_uiFaceTessSize` member to the number of faces in this array.

6. Declare a pointer to a null `A3DTess3D` entity.

7. Package the tessellation data by invoking the `A3DTess3DCreate` function, providing the location of the `A3DTess3DData` structure as the first argument and the location of the `A3DTess3D` pointer as the second argument.

```
A3DTess3D* pTess3D = NULL;
A3DTess3DCreate(&sTess3DData, &pTess3D);
```

8. Package the tessellation base data with in the `A3DTess3D` entity by invoking the `A3DTessBaseSet` function.

```
A3DTessBaseSet(&pTess3D, &sTessBaseData);
```

9. Add the `A3DTess3D` data to a representation item. (See Creating representation item PRC entities.)

The following code sample creates an A3D tessellation entity from a file that conforms to the STL format. It packages this entity with a poly Brep model representation item. STL is a file format native to the stereolithography CAD software created by 3D Systems.

```
// Create structure for facet data
A3DTessBaseData sTessBaseData;
A3D_INITIALIZE_DATA(sTessBaseData);
sTessBaseData.m_uiCoordSize = 9 * facetSize;
sTessBaseData.m_pdCoords =
   (A3DDouble*)allocCllBck(sTessBaseData.m_uiCoordSize * sizeof(A3DDouble));

// Create structure for normal vector data
A3DTess3DData sTess3DData;
A3D_INITIALIZE_DATA(sTess3DData);
sTess3DData.m_uiNormalSize = 3 * facetSize;
sTess3DData.m_pdNormals =
   (A3DDouble*)allocCllBck(sTess3DData.m_uiNormalSize * sizeof(A3DDouble));

// Total amount of indexes is calculated as follows:
// (one normal + one triangle (3 points)) * Number of facets
sTess3DData.m_uiTriangulatedIndexSize = 4 * facetSize;
sTess3DData.m_puiTriangulatedIndexes =
   (ASUns32*)allocCllBck(sTess3DData.m_uiTriangulatedIndexSize *
     sizeof(ASUns32));

// Set up variables used in reading STL data.
A3DUTF8Char name[100];
A3DUTF8Char dummy[100];
A3DDouble x, y, z;
ASInt32 iRet = 0;
ASUns32 i = 0;
ASUns32 j = 0;
const A3DUTF8Char* pcName = NULL;

rewind(stlFile);
curr = 0;
facetSize = 0;
memset(sBuffer, 0, A3D_MAX_BUFFER);
while(fgets(sBuffer, A3D_MAX_BUFFER, stlFile) > 0) {
   sscanf(sBuffer, "%s", key);
   if(!_stricmp(key, "solid")) {
      iRet = sscanf(sBuffer, "%s %s", key, name);
      if (iRet != 2)
         sprintf(name,"unnamed");
   } else if (!_stricmp(key, "facet")) {
      iRet = sscanf(sBuffer,"%s %s %lf %lf %lf",key,dummy,&x,&y,&z);
      sTess3DData.m_puiTriangulatedIndexes[4 * facetSize] = i;
      sTess3DData.m_puiTriangulatedIndexes[4 * facetSize + 1] = j;
      sTess3DData.m_puiTriangulatedIndexes[4 * facetSize + 2] = j + 3;
      sTess3DData.m_puiTriangulatedIndexes[4 * facetSize + 3] = j + 6;
      sTess3DData.m_pdNormals[i++] = x;
      sTess3DData.m_pdNormals[i++] = y;
      sTess3DData.m_pdNormals[i++] = z;
      facetSize++;
   } else if (!_stricmp(key, "vertex")) {
```

```
      iRet = sscanf(sBuffer, "%s %lf %lf %lf", key, &x, &y, &z);
      sTessBaseData.m_pdCoords[j++] = x;
      sTessBaseData.m_pdCoords[j++] = y;
      sTessBaseData.m_pdCoords[j++] = z;
    }
    curr += strlen(sBuffer);
    prgIncCllBck(curr);
    memset(sBuffer, 0, A3D_MAX_BUFFER);
}
fclose(stlFile);
prgEndCllBck();

// one can consider that STL consists in one single face made of total amount
of triangles
A3DTessFaceData sTessFaceData;
A3D_INITIALIZE_DATA(sTessFaceData);
sTessFaceData.m_usUsedEntitiesFlags = kA3DTessFaceDataTriangleOneNormal;
sTessFaceData.m_uiSizesTriangulatedSize = 1; /* size of the next array */
sTessFaceData.m_puiSizesTriangulated = &facetSize;
sTessFaceData.m_uiStartTriangulated = 0;

// so, only one face
sTess3DData.m_uiFaceTessSize = 1;
sTess3DData.m_psFaceTessData = &sTessFaceData;
A3DTess3D* pTess3D = NULL;
A3DTess3DCreate(&sTess3DData, &pTess3D);
A3DTessBaseSet(pTess3D, &sTessBaseData);

// create a PolyBrepModel representation item
A3DRiPolyBrepModel *pRiPolyBrepModel = NULL;
A3DRiPolyBrepModelData sPolyBrepModelData;
A3D_INITIALIZE_DATA(sPolyBrepModelData);
sPolyBrepModelData.m_bIsClosed = TRUE;
A3DRiPolyBrepModelCreate(&sPolyBrepModelData, &pRiPolyBrepModel);

// assign the tessellation to the PolyBrepModel
A3DRiRepresentationItemData sRiData;
A3D_INITIALIZE_DATA(sRiData);
sRiData.m_pTessBase = pTess3D;
A3DRiRepresentationItemSet(pRiPolyBrepModel, &sRiData);
```

20 Handling Exceptions

This chapter explains how to handle exceptions that are raised by Acrobat core API methods. Acrobat core API methods generally do not return error codes; instead, they raise exceptions when errors occur. Exceptions are handled by exception handlers. It is recommended that you create your own exception handlers to trap and handle exceptions.

Creating exception handlers

Exception handlers typically perform object cleanup (such as releasing memory) when an exception occurs. Your exception handler can either deal with the exception or pass it along to the next handler on the exception handler stack.

You use the `DURING`, `HANDLER`, and `END_HANDLER` macros to define exception handlers. Application logic that may raise an exception appears between the `DURING` and `HANDLER` macros. Application logic that handles exceptions appears between the `HANDLER` and `END_HANDLER` macros.

If the method raises an exception, the handler code is executed; otherwise, it is not executed. When an exception occurs, your handler can access the exception error code by using the `ERRORCODE` macro. The value returned by the `ERRORCODE` macro does not change until another exception is raised. (See the Acrobat and PDF Library API Reference.)

The following code example declares an error handler that is active during a call to the `AVDocOpenFromFile` method. If an exception occurs, an error message is displayed to the user. For information about opening a PDF document by using the `AVDocOpenFromFile` method, see "Opening PDF documents" on page 64.

Example: *Creating an exception handler*

```
DURING
   AVDoc avd = AVDocOpenFromFile(asp, NULL, (char *)NULL);
HANDLER
   AVAlertNote("Error opening file");
END_HANDLER
```

Note: For code brevity, most of the code examples located in this guide do not include `DURING`, `HANDLER`, and `END_HANDLER` macros to handle exceptions. You should include these macros in your own custom code.

Returning a value from an exception handler

To return a value from a `DURING HANDLER` block, do not use a return statement. Instead, use the `E_RETURN(x)` or `E_RTRN_VOID` macros. The `E_RETURN(x)` macro returns a value and the `E_RTRN_VOID` macro does not return a value (this is equivalent to a `void` function).

The `E_RETURN(x)` macro must not invoke a function that might raise an exception, because the macro pops an exception frame off the stack before evaluating the expression to return. If this evaluation raises an exception, it does not call your handler, but instead calls the next handler up the stack. For example, consider the following application logic:

```
E_RETURN(myfn());
```

This is not recommended if there is any possibility that `myfn` could raise an exception. If you need to invoke a function, it is best to do it this way:

```
result = myfn();
E_RETURN(result);
```

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

Raising exceptions

Although many Acrobat core API methods raise exceptions, some methods do return error codes or `NULL` if something goes wrong. Inside a `DURING HANDLER` block, if your plugin detects one of these cases, it can invoke 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.

You can also use the `RERAISE` macro when you do not want your exception handler to handle an exception, but want to pass the exception to the next exception handler on the stack.

Note: Your plugin should use the `ASRegisterErrorString` method to define its own exceptions.

Exception handling scenarios

This section discusses the following potential problems with exception handling:

- Goto statements
- Nested exception handlers in a single function
- Register usage

Using goto statements

It is recommended that you not use a `goto` within a `DURING HANDLER` block. Jumping outside a `DURING HANDLER` block disrupts the stack frame, as shown in the following example:

```
DURING
   goto error;
HANDLER
END_HANDLER
error:
```

This action leads to unpredictable results because the top stack frame has not been removed. As a result, the frame is incorrect. Instead, use the `ASRaise` method and then use the `goto` statement within the `HANDLER END_HANDLE` block, as shown in the following example:

```
DURING   ...
   ASRaise(myspecialerrorcode);
HANDLER
if ERRORCODE == myspecialerrorcode
   goto error;
END_HANDLER
error:
```

For information about the `goto` statement, see the Acrobat and PDF Library API Reference.

Using nested exception handlers

Avoid using nested exception handlers within a single function. The exception handling macros change the call stack, and nesting them can disrupt the stack. Your plugin can safely nest an exception handler if the nested handler is in another function called inside the `DURING HANDLER` block, as shown in the following example.

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

If you insist on nesting exception handlers in a single function, do not return from the inner exception handler (either through a call to return in a handler or `E_RETURN` from body code). This action leaves the exception stack out of sync with the call stack. Any errors raised in body code surrounded by the outer exception handler will restore the incorrect calling environment and lead to unpredictable results, as shown in the following example.

```
{
DURING /* Places one frame on the exception stack */
pdoc = AVDocGetPDDoc(avdoc);
   DURING /* Places a second frame on the stack */
      rootBm = PDDocGetBookmarkRot(pdDoc);
      if (!PDBookmarkIsValid(rootBm)){
      E_RTRN_VOID
      /*
      Returning here interferes with the exception stack
      because two frames have been placed on the stack
      and E_RTRN_VOID only clears one of them before
      returning
      */
}
pdAction = PDBookMarkGetAction(parentBm);
HANDLER
      AVAlertNote("Bad AVDoc");
      return (1);
      /*
      Returning here interferes with the exception stack
      because there is still a frame on the stack from
      the outer DURING macro and it will not be cleared
```

```
     before the function returns
     */
END_HANDLER
HANDLER
   AVAlertNote("Bad PDDoc");
END_HANDLER
}
```

Using register variables

The `DURING` and `HANDLER` macros use the standard C `setjmp/longjmp` mechanism. The `DURING` macro calls `setjmp`. An exception results in a `longjmp` to the context that was saved by the most recent `setjmp`. When a `longjmp` occurs, all registers, including those containing variables the compiler optimized into register variables, are restored to the values they held when the `setjmp` occurred.

As a result, the state of local variables that have been optimized into registers is unpredictable when the exception handler is invoked. To avoid this situation, declare all variables that are set in the main body of the code and used in the exception handler or beyond (if the handler lets execution continue) as volatile. This ensures that they are never optimized into register variables, but are always referenced from memory.

When using the `volatile` statement, place the keyword in the correct location, such as the following:

```
volatile myStruct* p = 0;
```

The previous statement declares the instance of the structure to be volatile. The next statement declares the pointer itself to be volatile:

```
myStruct* volatile p = 0;
```

In general, the second version is the one to use.

21 Working with Acrobat Extended APIs

Some of Acrobat's default plugins expose their APIs for use by third parties. These are Acrobat's extended APIs. These APIs are specified in the Acrobat and PDF Library API Reference.

Search extended API

The Adobe Acrobat Search plugin allows users to perform text searches in PDF documents. It adds menus, menu items, a toolbar button, and a Search panel to the Acrobat viewer.

The Search plugin exports a host function table (HFT) that contains methods that can be used by other plugins. The HFT's name is Search, and its version number is 0. To use the Search plugin's HFT, a plugin must include the header file SrchClls.h. The plugin must also import the HFT using the `ASExtensionMgrGetHFT` method and assign the HFT returned by this call to a plugin-defined global variable named `gSearchHFT`. (See "Working with Host Function Tables" on page 156.)

Acrobat 6.0 and later has two versions of the Search plugin:

- The Search plugin uses a search engine licensed from Lextek International. Lextek can be contacted at http://www.lextek.com.
- The Search5 plugin uses a search engine licensed from Verity, Inc. Verity can be contacted at http://www.verity.com.

You can perform the following tasks with either version of the Search plugin:

- Create or delete indexes
- Determine what indexes are available
- Send queries to an existing index

You cannot use either version of the Search plugin to directly obtain the results of a search, for manipulation or for display in an application other than Adobe Acrobat. For information about the methods included in the Search extended API, see the Acrobat and PDF Library API Reference.

Catalog extended API

Acrobat Catalog is a plugin that allows you to create a full-text index of a set of PDF documents. A full-text index is a searchable database of all the text in the documents. After building an index, you can use the Search command to search the entire library quickly. Searches of full-text indexes created using Catalog are faster and more convenient than using the Find command.

The Catalog plugin has an HFT consisting of several methods that plugin developers can import and use. In addition, Catalog supports DDE, and broadcasts several Windows messages. For information about the methods included in the Catalog extended API, see the Acrobat and PDF Library API Reference.

PDF Consultant and Accessibility Checker extended API

Acrobat comes with a plugin called the PDF Consultant and Accessibility Checker. This plugin walks through PDF documents, visiting each object and determining its type and other statistics. It can make certain modifications or repairs to the PDF document. The objects that the consultant visits can range from simple, primitive types such as Cos strings to higher-level objects such as images. Users invoke the consultant to operate on a particular PDF document, choose which tests or repairs to run, then view the results and select repair options.

The consultant visits the objects in a PDF document according to instructional flags you pass to it. After the consultant visits an object, the object may be different. The consultant reclassifies modified objects before moving on to the next object.

As the consultant traverses a PDF document, gathering objects of interest, it can perform the following tasks:

- Walk a given hierarchy
- Keep track of cycles
- Ensure that objects are only visited once, if desired
- Recognize object types
- Keep a traversal stack list

Acrobat agents

The consultant accomplishes its task by using agents, which are pieces of code you design to gather the statistics and recommend to the consultant the necessary repairs of the document. Separate agents handle each area of analysis and repair. The agents inform the consultant of the particular types of objects in which they are interested by registering with the consultant. When the consultant has one or more agents registered, it hands each object of the requested type(s) in the current document to each of the agents that requested that type. The consultant gives objects to each agent in turn, depending on the order in which they registered.

The consultant must intelligently determine the type of each object it comes across (both direct and indirect), so it can pass appropriate objects to the agents, or replace or remove ones that it has been instructed to handle itself. The consultant communicates directly with agents, keeping lists of which agents are interested in which objects, and obtaining instructions from the agent as to an object's visitation status.

Agents can perform their own repairs and modifications to the PDF document, and can return a corrected object to serve as a replacement for the object the consultant originally passed to it. Agents can also modify the Cos graph themselves (including adding or removing Cos objects or modifying the contents such as keys or array elements).

The consultant keeps a list of each object (starting with the object which began the traversal) that it visits on its way to any given object. Agents must be careful not to make any modifications that would affect any of the objects in the list, which is referred to as the traversal stack. For this reason, agents can specify a post-processing callback that the consultant invokes once it has finished traversing the entire document.

Reclassifying and revisiting

If an agent or the consultant modifies an object, the consultant reclassifies that object, possibly changing its type.

Agents also pass visitation flags to the consultant that determine how object types should be visited. Limiting the traversal is important, as PDF documents are graphs, arbitrarily complex, and often there are many ways to visit a single object. If the consultant reclassifies an object, it may also change the way that object is revisited. You must keep this in mind as you develop your agents.

Agent architecture

Your agent code will primarily consist of a structure, as defined in the ConsExpt.h header file. Acrobat provides a C++ wrapper class to facilitate writing agents; you can derive an agent class from this base class.

How the consultant works

The consultant completes a full, non-recursive traversal of the Cos graph that comprises a PDF document, keeping track of cycles as it goes. Note that there is no guarantee that objects will be visited in any particular order, only that the consultant will visit all objects (except isolated objects such as the `DocInfo` object or previously orphaned objects) at least once, provided no agents modify the graph such that graph paths are removed or redirected.

Removing or modifying objects

If an agent removes, replaces, or modifies an object, the consultant passes the modified objects (if they are encountered) to other agents. For example, `DictA` refers to `DictB`.The first agent replaces all references to `DictB` with references to `DictC`, so when later agents receive `DicA` from the consultant, they will see the references to `DictC`.

Reclassifying objects

In general, the consultant reclassifies an object after an agent is finished performing operations on it. It is possible that, in the process of modifying the object, the agent may actually change the type of the object. This could mean that agents originally interested in the object may not be interested in it. So the consultant must reclassify an object after each agent has finished with it. Because the default behavior in the revisit upon reclassification mode is to revisit objects when they are reclassified, new objects added in this mode will actually be visited again if they are reclassified as the traversal continues.

Determining the higher-level type (the `PDFObjType`, as the consultant code calls it) of a given Cos object is not always easy. The consultant not only looks at the construction of objects (what keys are present in the object) but also at how the object was reached (through what particular object type and via what keys). Objects that are interpreted differently depending on how they are traversed can be properly identified.

Consultant process

The following steps describe the consultant process:

1. You create a consultant.
2. You create an agent.
3. Register your agent with the consultant, with information as to which object types are of interest.
4. The user invokes the consultant to work on a particular PDF document.
5. The consultant creates a traversal stack to keep track of where it is in walking through the PDF document.

6. The consultant begins traversing the PDF document. If agents have instructed the consultant to modify or remove the object, it does so, returning the appropriate replacement.
7. The consultant pushes the object onto the traversal stack and sends a message to the agent that the object was found.
8. The agent sends messages to the consultant about what to do to objects: replace them, remove them, revisit them later or not.
9. When the entire PDF document has been traversed, the consultant calls the agent back to perform any post-processing repairs it may want to do.
10. The consultant unregisters all agents.
11. You release the agent object.
12. You release the consultant object.

Important issues for consultant development

First, you must decide if you actually do want to use a consultant. A consultant walks through an entire PDF document. If you only need to modify a small number of objects, and you know how to locate those objects, it makes more sense to write the object-finding code yourself.

If you decide to use the consultant, here are some planning considerations:

- Avoid implementing an agent that modifies objects on the traversal stack while the consultant is still walking through the document, otherwise infinite loops and other problems can occur (see Maintaining the traversal stack).
- Decide whether the consultant or the agent does the work (see Deciding consultant or the agent does the work).
- Determine order in which agents interact with the consultant. This order is important because agents can modify objects that other agents want to view (see Avoiding agent collisions).
- Decide whether to allow the consultant to revisit objects that have multiple classifications and what conditions must exist to allow such repeat visits (see Avoiding visitation collisions).

You should make your decisions about all of these issues before you write your code. Some of these issues lead to errors that are difficult to debug, so it is best to understand them all while creating your plugin.

Maintaining the traversal stack

The consultant keeps track of the objects it has visited in the PDF document in the traversal stack. If an agent modifies an object such that it affected the traversal stack, the entire process is derailed. The consultant may no longer know if it had visited an object, which could cause infinite loops, multiple, unnecessary visitations, or objects that remain unvisited.

It is extremely important that the integrity of the traversal stack remain undamaged. You must design your agent carefully so as to avoid this problem. You can use the postprocessing step of your agent to handle many repair tasks, thereby avoiding dealing with objects still on the traversal stack.

Deciding consultant or the agent does the work

If the consultant performs object modifications, it does so as it goes through its traversal. Modifications that affect an object's type or properties alter the traversal stack and corrupt the traversal process. For these kinds of modifications, set up an agent to perform the tasks in the postprocessing step.

For example, suppose an agent wants to remove annotations while there are form widgets present in the document. There are a few ways the agent can remove the annotations while the consultant is working, but they all have problems:

- Invoking the agent for all annotations and removing them at the Cos level does not clean up the forms tree if there are Widget Annots in the document.
- Invoking the agent for all annotations and using the `PDPageAnnotRemove` method modifies the page object, which may still be in the traversal stack.

The best solution in this case is to enumerate all of the `Annot` objects by having the consultant look for `Annot` objects and keep a list of them, then let the agent invoke `PDPageAnnotRemove` on them in the post-processing step.

Avoiding agent collisions

When running multiple agents on a document, the order in which you register your agents is the order in which the consultant will hand them objects. If your earlier agents modify objects, they may change the objects in such a way that they are missing important information or are of a different type than they were originally. For example, one agent may consider it correct to remove a given field of an object, while another would complain that the field was not present and would want to add it. If the first agent modified the object type, subsequent agents would no longer think they were interested in it, and their processing would not take place. You must group your agents so that you do not run multiple agents with conflicting goals at the same time.

A rarer problem could occur with self-referential objects. For example, if `DictA` contains a reference to itself and the first agent replaces `DictA` with `DictB` (which would still contain a reference to `DictA`), another agent cannot work with `DictB` until the internal reference is changed. But if you are running the agents concurrently, there will be a collision. This would be a case best handled by the consultant.

Avoiding visitation collisions

Objects that have multiple classifications can be reached from multiple paths. In such cases you may allow the consultant to revisit such objects if, and only if, they were reclassified on a new path. However, you must take care not to allow revisitation under other circumstances, or the consultant could miss objects, which would defeat the reason for using a mode that considers object classification.

Importing the consultant HFTs into a plugin

The consultant exports its functions using an HFT. The variable name your plugin uses for the HFT must be of type HFT and named `gConsultantHFT`. The consultant's HFT allows you to create consultants. The consultant exports an HFT that deals with the general operation of the consultant, including the creation and deletion of consultant objects and agent registration. You must load the consultant plugin before the HFTs plugins can import it. Importing the consultant's HFT is the same as importing any other plugin's HFT. (See "Importing an existing HFT" on page 164.)

To access the HFT, you must include the ConsHFT.h file into your project. In a plugin, the `PluginImportReplaceAndRegister` method should contain the code that imports the HFT.

Example: *Importing consultant HFTs*

```
HFT gConsultantHFT= (HFT)NULL;
ACCB1 ASBool ACCB2 PluginImportReplaceAndRegister(void)
{
   ASBool bRetVal = false;

   //Import the Consultant's main HFT
   gConsultantHFT = Init_PDFConsultantHFT; // Macro in ConsHFT.h
   if(gConsultantHFT != (HFT)NULL)
      bRetVal = true;
   else
   //Put in error message about the absence of the Consultant HFT
   return bRetVal;
};
```

The consultant defines the following methods for HFT usage:

- `ConsultantCreate`
- `ConsultantDestroy`
- `ConsultantTraverseFrom`
- `ConsultantRegisterAgent`
- `ConsultantSetStart`
- `ConsultantNextObj`
- `ConsultantGetPercentDone`
- `ConsultantGetNumDirectVisited`
- `ConsultantGetNumIndirectVisited`
- `ConsultantSuspend`
- `ConsultantResume`
- `ConsStackGetCount`
- `ConsStackIndexGetObj`
- `ConsStackIndexGetTypeCount`
- `ConsStackIndexGetTypeAt`
- `ConsStackIndexIsDict`
- `ConsStackIndexIsArray`
- `ConsStackIndexGetDictKey`
- `ConsStackIndexGetArrayIndex`
- `PDFObjTypeGetSuperclass`
- `ConsultantGetNumUniqueIndirectsVisited`

Creating and destroying consultants

The consultant's HFT allows you to create a consultant for your own use. Once you have finished writing your agent class, you are ready to register it with the consultant and begin processing documents. You should keep your agent separate from the consultant object—that is, do not make the consultant object a member of your agent class. Use a plugin as the owner for both the consultant and your agent object.

Because there is some memory overhead in creating a consultant, you should only create a `Consultant` object when it is required, not before. If your target application is a plugin, the most logical place to perform all operations is in the menu item execute procedure. Whether or not it makes sense to destroy the `Consultant` object after each execution of the menu item depends on your project.

The consultant HFT provides the functions `ConsultantCreate` and `ConsultantDestroy`, for creating and destroying `Consultant` objects. It also provides the Consultant data type, an opaque type for passing handles to `Consultant` objects. The `ConsultantCreate` method returns variables of that type and requires them as parameters to all other HFT functions having the `Consultant` prefix.

After each run the consultant unregisters all the agents that were registered with it; however the memory for the `Consultant` object itself remains, and the object must be explicitly destroyed to free the memory. Depending on the duties you assign your consultant, you may want to destroy it after each execution of the menu item that launches it, or you may wish to keep it running.

Registering agents with consultants

In order to modify or analyze documents, you must register your agent with the consultant by invoking the `ConsultantRegisterAgent` method.

Once the agent is registered with the consultant, it remains registered until a call to `ConsultantTraverseFrom` is made. You must re-register agents before each successive call to `ConsultantTraverseFrom`.

When you register an agent, you supply a rule (one of the `RegAgentFlag` values) for revisitation of objects as the consultant runs through the document from the starting object. The following code example registers an agent with a consultant.

Example: *Registering an agent with a consultant*

```
//Declare volatile consultant because it is inside a DURING bloc
Consultant volatile hConsultant = (Consultant)NULL;

DURING
   AVDoc hAVDoc = AVAppGetActiveDoc();
   miAssert(hAVDoc != ( AVDoc )NULL );
   if( hAVDoc != ( AVDoc )NULL )
      {
      //Create a Consultant object
      hConsultant = ConsultantCreate(
      DumpAllObjectsAgentPercentDone );
      miAssert( hConsultant != ( Consultant )NULL );

      if( hConsultant != ( Consultant )NULL )
      {
         //Get the current document root
         PDDoc hPDDoc = AVDocGetPDDoc(hAVDoc);

         //Create an agent and register it
         gDumpAllObjectsAgent = new DumpAllObjectsAgent(hPDDoc);

         if((gDumpAllObjectsAgent == (DumpAllObjectsAgent*)NULL)
         || (gDumpAllObjectsAgent->IsValid() == false))
         {
```

```
        ASRaise(GenError(genErrNoMemory) );
        }
        else
        {
        ConsultantRegisterAgent(hConsultant,*gDumpAllObjectsAgent,
        REG_REVISITRECLASS_ALL );

        //Start the consultant
        ConsultantTraverseFrom(hConsultant,
        CosDocGetRoot(PDDocGetCosDoc(hPDDoc)),PT_CATALOG);
        }
     }
}
HANDLER
... Destroy Consultant...Free Memory...
END_HANDLER
```

Starting the consultant

The `ConsultantTraverseFrom` method instructs the consultant to begin traversing a document, starting at a specific Cos object. The Cos object should be the catalog of a currently open document. The `ConsultantTraverseFrom` method has no return value and instead raises an exception if an error occurs. The following code example demonstrates how to use the traversal stack manipulation functions.

Example: *Using the consultant traversal stack*

```
char* GetTraversalString(ConsStack stack, char *traversalString,
ASUns32 strLen)
{
ASUns32 Index, NumItems, CurStrLen;
char StringUns32[16]; traversalString[0] = '\0';
CurStrLen = strlen(traversalString);

//Get the number of items in the current traversal
NumItems = ConsStackGetCount(stack);

for(Index = 0; (Index < NumItems) && (CurStrLen < strLen); Index++)
{
   if((CurStrLen += strlen(TRAVERSAL_SEP)) < strLen)
      strcat(traversalString, TRAVERSAL_SEP);

   //Add the parent key, if this stack entry has one */
   if(ConsStackIndexIsDict(stack, Index))
      {
      char* strParentKey =   ASAtomGetString(ConsStackIndexGetDictKey(stack,
      Index));
      if((CurStrLen += strlen(strParentKey)) < strLen)
         strcat(traversalString, strParentKey);
      }
//Add the parent index, if this stack entry has one
else if(ConsStackIndexIsArray(stack, Index))
{
   sprintf(StringUns32, "%u",
   ConsStackIndexGetArrayIndex(stack, Index));
```

```
   if((CurStrLen += (strlen(StringUns32) + 2)) < strLen)
   {
      strcat(traversalString, "[");
      strcat(traversalString, StringUns32);
      strcat(traversalString, "]");
      }
   }
}
return traversalString;
}
```

Consultant object type identification

One of the main features the PDF Consultant and Accessibility Checker framework gives you is the use of its identification engine. This engine can look at Cos objects in a PDF file and, based on properties of the objects and of the object's parents, assign PDF object type identifiers to them.

Each Cos object has a simple Cos type and attributes, in the scheme of the document as a whole each object serves a particular purpose. The PDF object type assigned to each object represents that object's role in the PDF document.

Some PDF object types represent higher-level, conceptually-familiar objects like `PT_PAGE` (which indicates that the object is a page in the document), while others (like `PT_AADICTIONARY`) are a bit more obscure, particularly to those who are not familiar with the PDF document format. PDF object types are represented using the enumerated type `PDFObjType`, which is defined in ConsObTp.h. A good way to see all of the various PDF object types that the consultant can identify is to look at the constants defined in that file.

Some object types (in particular many simpler objects such as strings and numbers) are not assigned a particular type. The consultant can identify those objects that are of most use to you. If the consultant cannot identify a specific object, it assigns the identity of `PT_UNKNOWN` to the object. Just because the consultant assigns this value to an object does not mean the object is foreign or invalid (although it can potentially mean that), it may simply mean that the object type is not particularly significant in the realm of the PDF document format, and thus the consultant does not know about it.

To allow for greater agent flexibility, the consultant understands PDF object type subclasses and superclasses. Certain PDF object types are members of more generic classes of PDF object type. Agents can often make use of this information, so the consultant assigns object types that are actually arrays of types.

The consultant assigns to an object the most specific classification as well as the more generic classes of which the object is a member. Agent structures include a field called `WantSubclasses` that indicates whether or not the agent wants to be called for all the interesting objects' subclasses as well as their directly interesting types.

For example, the `PT_ANNOTATION` object type has a number of more specific subclasses such as `PT_LINKANNOTATION`, `PT_LINEANNOTATION`, and so on. If an agent requests only objects of type `PT_ANNOTATION`, and its `WantSubclasses` member is `false`, it may not be called back for very many objects. If the `WantSubclasses` member is `true`, then the consultant will invoke the agent back for objects of all specific types of annotations as well as those classified only as `PT_ANNOTATION`. This also means that when an agent retrieves the type of an object, it must specify which type it wants. The types in the array that is the classification of the object always go from the most specific (at index 0) to the least specific (the last index in the array).

Creating an agent class

A minimal `Agent` class needs only to define the functions defined as virtual in the `ConsultantAgentObject` class declared in ConsExpt.h. The following example shows this definition.

Example: *Creating an agent class*

```
#include "ConsExpt.h"
class DumpAllObjectsAgent : public ConsultantAgentObj
{
protected:
//Data members
FILE* m_DumpFile;
const static PDFObjType s_hAgentObjects[ ];
const static ASUns32 s_iNumAgentObjects;
public:

//Constructor / destructor
DumpAllObjectsAgent(PDDoc hPDDoc);
virtual ~DumpAllObjectsAgent(void);

//Required methods
virtual void ConsAgentPostProcess(void);
virtual ASInt32 ObjFound(CosObj Obj, const PDFObjType*
pObjTypeHierarchy,
const ASUns32 SizeObjHierarchy,
TraversalStack Stack,CosObj* pObjToReturn);
};
```

Creating agent constructors

In order to write an `Agent` class derived from the `ConsultantAgentObj` base class, you must invoke the base constructor in the derived classes construction list. The base constructor requires a constant array of so-called objects of interest (of type `PDFObjType`) as well as the length of the array (as `ASUns32`) to be passed as parameters. It is up to you as to where and how the array of types is stored; however, the storage must persist, as the base class saves only a pointer to the data. This has important implications for authoring agents; the derived class cannot initialize the data in its own constructor since the base constructor is called first.

The following example shows an example constructor. In the Agent example the array types and array length are static data members of the `Agent` class. In larger-scale systems it is better to create a host object for the agent that is responsible for determining the proper objects to include in the array and for passing them on to the `Agent` constructor. The list of object types is passed on to the consultant when `ConsultantRegisterAgent` is invoked.

Example: *Creating agent constructors*

```
//Define static const data to be passed to parent class constructor
const ASUns32 DumpAllObjectsAgent::s_iNumAgentObjects = 1;
const PDFObjType
DumpAllObjectsAgent::s_hAgentObjects[DumpAllObjectsAgent::
s_iNumAgentObjects] = {DT_ALL};

//Derived Agent class constructor */
```

```
DumpAllObjectsAgent::DumpAllObjectsAgent( PDDoc hPDDoc ) :
ConsultantAgentObj( &s_hAgentObjects[ 0 ], s_iNumAgentObjects )
{
Open Temporary File and Initialize Data Members ...}
}
```

Recognizing objects of interest

Agents register a list of objects with the consultant in which they are interested. When the consultant classifies an object as any of the types the agent registered with, the consultant calls the `ObjFound` callback function, a virtual function in the `ConsultantAgentObj` base class.

- The parameters the consultant passes to this function allow the function to set up a return value with information about the current object, its parents, and the state of the consultant traversal stack.
- The return value from the callback is an `OR` of bit flags that instruct the consultant on handling the current object.

In the Creating agent constructors example, an `Agent` constructor simply gathers information about each object encountered and outputs it to a file. It does not need to have the consultant make any modifications to the document. Therefore, in the definition of the `ObjFound` callback function, the return value is always `OD_NOCHANGE` and the object returned in `pObjToReturn` is simply the same object that was found. In many cases it makes the most sense for an agent to make all document modifications itself, without the consultant's replace and remove facilities. In these cases you must take special care not to modify objects that are currently on the consultant's traversal stack.

The `DumpAllObjects` plugin demonstrates that `PDFConsultant` agents can access any Cos object from any point in the document. The plugin writes information about certain Cos objects to an output file, called AllObjects.txt.

The `ObjFound` callback function of the `DumpAllObjects` agent writes to a file the Cos object traversal path that it took to reach a specific Cos object. The function calls `GetTraversalString`, which describes, with respect to other objects, where a given object lives in the document. For example, the following shows the format of a traversal path of a text annotation:

```
18 0 obj PT_TEXTANNOTATION | PT_ANNOTATION | ->AcroForm->Fields->[0]->
P->Annots->[1]
```

The consultant looks at all Cos objects. To simplify the output, the `DumpAllObjects` agent only involves the most common Cos objects—`CosString`, `CosDict`, `CosArray`, and `CosStream`.

Post processing stage

The second and final required function definition in any `ConsultantAgentObj` derived class is the `PostProcess` callback. This function is called when the consultant has finished its traversal and is preparing to unregister agents to prepare for the next possible run. This callback takes no parameters and returns no values (see ConsAgentPostProcessCallback). There are also no restrictions on what types of operations the Agent can perform on the document in this function.

The `PostProcess` callback function is the place to perform any operations that may otherwise damage the consultant's traversal by modifying objects up the consultant's current traversal stack.

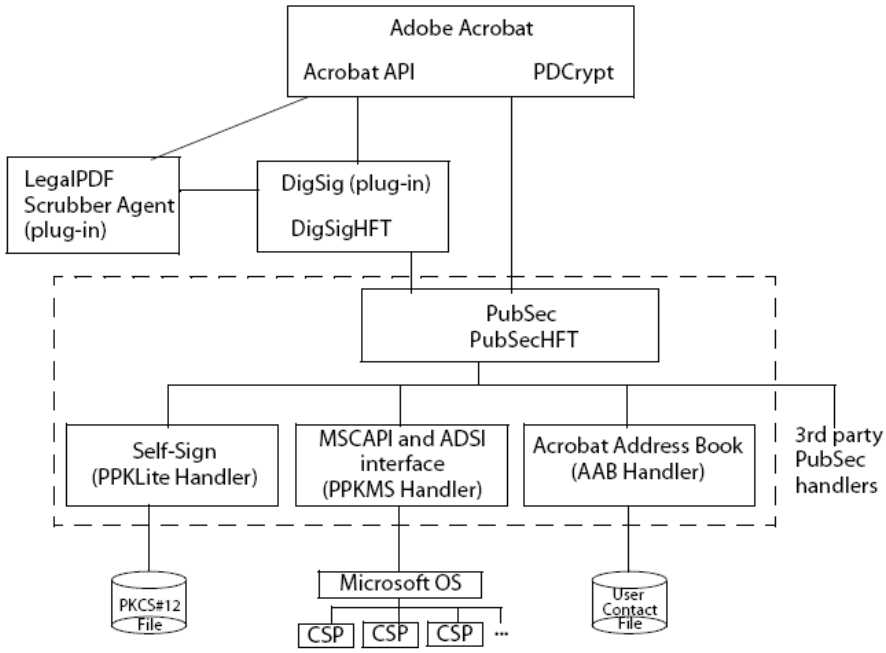
Digital signature extended API

Digital signatures allow a person to attest to something about a document by signing their name to it. An Acrobat signature in a document is bound to that document in such a way that altering the signed document or moving the signature to a different document invalidates the signature.

A single document may be signed more than once, and changes may occur between signings. Acrobat's digital signatures link each signature with a particular state of the document. All changes append the PDF changes to the fully-preserved base PDF document. The ability to do serial signatures of protected documents is unique to Acrobat, and draws heavily on the PDF file design for an appended save.

Adobe Acrobat implements digital signatures using plugins that can handle both generic functions common to all digital signatures, and also specific kinds of signatures (signing methods), such as public-private key (PPK), handwriting, retinal scans, fingerprints, and so on.

The following diagram shows the security plugin relationships.



For information about the APIs that make up the Digital signature extended API, see the Acrobat and PDF Library API Reference.

The PubSec layer

The PubSec layer, introduced in Acrobat 6.0, is an interface for Acrobat public-key security handlers. PubSec forms a high-level interface to the digital signature facility; the PubSec code uses DigSig for digital signature operations, but provides many additional benefits. Developers are encouraged to use the PubSec HFT rather then DigSig HFT.

PubSec methods enable you to perform the following tasks:

- Count and close encrypted documents.
- Validate a specific signature field.
- Access and create digests for data buffers.

- Import and export certificate data, and manage the certificates in the Acrobat Address Book (AAB).
- Manage signature appearances (DSAP files).
- Register and unregister handlers. Handlers can register as PubSec handlers to provide the following cryptographic services:
  - Do private-key signing and signature validation
  - Act as a cryptographic source for decrypting using private keys
  - Act as a directory source for certificate-based identity authentication

Handlers can call back into the PubSec HFT for various services. Most calls to PubSec pass an opaque state object called a `PSEngine`. You specify a default engine upon registering the handler, and the default engine can make use of the security UI dialog boxes provided by PubSec and DigSig.

To register a handler with PubSec, perform the following tasks:

1. Implement the callbacks you need to provide customized functionality. Many of the callbacks for PubSec can be specified as NULL, in which case PubSec provides default behavior. It is recommended that you use the default behavior when possible.
2. Fill in the handler structure with pointers to your callback implementations (PubSecHandler).
3. Register the handler with PubSec by invoking the `PSRegisterHandler` method.

Digital signature components

Digital signatures contain two parts:

- The signature field dictionary which is the PDF dictionary structure that stores information about the signature.
- The signature annotation with its associated appearance (including the background and layout of name, time, and so on). A blind digital signature does not have an associated appearance.

Acrobat's digital signature plugin creates these two parts when the user chooses to sign a document. Your plugins do not have to handle deleting the signature, as the DigSig plugin does that transparently.

Digital signature scenarios

Acrobat supports three digital signature scenarios. Acrobat's Digital Signature plugin handles the first case, and allows other plugins to further handle the second and the third cases.[The following steps look as if they are part of a procedure, but I don't think they are. If they're simply three scenarios, they should be bulleted items, 5Heads, or perhaps reworked as inline headings with text. The other situation where we use a numbered list is if order is important, but I don't think that is the situation here either. The Adobe editorial style guide says "A numbered list should be sequential. If the items in the list are not sequential, don't number them.]

- If the user creates a signature field and does not specify a default signing method, DigSig handles that case with no communication to your plugins:
  - DigSig creates the signature field dictionary.
  - DigSig creates the signature annotation dictionary.
  - DigSig creates the (blank) signature appearance dictionary.

- The Forms plugin also creates Signature fields. If the user creates a signature field and specifies a default method, Forms calls DigSig to fill in default values:
  - DigSig creates the signature field dictionary, the signature annotation dictionary, and the (blank) signature appearance dictionary.
  - DigSig calls the DSDefaultValueProc callback that your plugin provides. This callback must create the default signature value dictionary and create the /DV key in the signature field dictionary to point to it.
- If the user asks to sign a specific signature field using the plugin, DigSig invokes callbacks into your plugin during a four-step sequence. Your plugin must register these callbacks during the plugin initialization phase. The four callbacks required for this scenario are:
  - `dsNewSigData`
  - `dsCommitSign`
  - `dsFinshSign`
  - `dsFreeSigData`

Initializing the digital signature plugin

When Acrobat is started, all plugins go through a three-step initialization process that allows plugins to establish communication among themselves without being dependent on the order of loading. Plugins that interact with Acrobat's digital signature plugin (DigSig) use the following initialization sequence:

1. The DigSig plugin exports its HFT under the name DigSigHFT.
2. To work with DigSig, your plugin must import the DigSig HFT.
3. To work with DigSig, your plugin must create a `DigSigHandlerRec` structure, assign the relevant methods, and then invoke the `DigSigRegisterFilter` method to register the structure.

When the user opens a document, the digital signature plugin notifies your plugin of the new document by invoking the `DSDocOpenProc` method. You can allocate some storage or choose to automatically validate any of their respective signatures in the document.

Auto-validation may produce significant delays if it must read all of a large document from a CD-ROM or over a network, or if it must access a signature registry or authority over a network. Therefore, Adobe software only accesses signatures at user request.

When the user closes a document, the digital signature plugin invokes `DSDocCloseProc`.

Understanding the process

The steps in this section are suggestions that describe the interactions of a digital signature plugin (the SignDoc sample plugin provided with this SDK is a more complete example).

Dialogs and signature gathering

The digital signature plugin invokes your `dsNewSigDataProc` callback method, a callback that begins the signature gathering process:

- Your plugin interacts with the user, and allows the user to cancel if they want to do so.

- Your plugin acquires the signature itself in a method-specific way. All information is saved in memory, without altering the document itself.
- If `dsNewSigData` does not cancel, DigSig prepares the document for saving. First, it calls `dsUnValidateSig` on every signature in the document to put any overprinting or underprinting in canonical form. It then counts how many pages and fields have changed since any prior signature and records this.
- For a first signature, the digital signature plugin displays the Save As dialog box, allowing the user to select file name, optimization, and encryption. The user may cancel. Other than fatal errors, such as out-of-disk-space, this is the last chance to stop the process.

Saving a document

The following steps describe how the digital signature plugin saves a document:

1. The digital signature plugin invokes your `DSCommitSignProc` callback method to update the document with the actual signature. Your `DSCommitSignProc` callback must perform the following tasks:
   - Create the signature dictionary, possibly using information in the signature field /DV dictionary, perhaps using the `/ByteRange` and `/Contents` keys.
   - Point /V in the signature field dictionary to this. Then create the /AP /N value in the signature annotation dictionary, using a method-specific visible representation of the signature, including a symbol signifying "unvalidated signature."
   - Optionally allocate dynamic storage for a marked array, an array of marked COS objects that it cares about.
   - Return a marked array that includes at least the /ByteRange and /Contents value objects.
2. The digital signature plugin inserts the /Changes array from step 1.
3. The digital signature plugin saves the PDF document to a file. For each Cos object in the marked array, DigSig records the object's byte offset and length in the file as written. The saved file may have objects encrypted by the Acrobat standard encryption handler, if the user so chooses.
4. The first time a document is signed, the digital signature plugin may rename the file and may invoke the Optimizer, Linearizer, and Garbage Collector. Upon return from the save, all Cos objects are invalid, including those in the marked array.

   All PD-level objects except the `PDDoc` are invalid. Signing methods must not depend on saving any such state between `dsCommitSign` and `dsFinishSign`. In particular, the byte offsets and lengths in the marked array are valid upon entry to `doSign`, but the Cos objects are not. The order of entries is unchanged, however, these Cos objects will be rewritten as CosNull before invoking *dsFinishSign*.

Finishing the process

The following describes how the digital signature plugin finishes the process of a signing a document:

1. Invokes `dsFinishSign`, passing back in the marked array. Your `DSFinishSignProc` callback method must perform the following tasks:
   - Calculate the `/ByteRange` that it desires, using the byte offsets and lengths in the marked array.
   - Overwrite the marked `/ByteRange` value in the saved file, using the `DigSigOverwriteIntArray` or `DigSigOverwriteBytes` callback.
   - Overwrite any other marked Cos objects it wants to.

- Calculate any document digest that it desires, using the `DigSigFileGetEOF`, `DigSigFileSetPos`, and `DigSigFileRead` callbacks; or it may use the `DigSigMD5ByteRange` callback.
- Obscure or encrypt this digest in a method-specific way.
- Overwrite the marked `/Contents` value in the saved file, using `DigSigOverwriteHexstring` or `DigSigOverwriteBytes`.
- Optionally delete dynamic storage for the marked array returned by the plugin.

2. Invokes `dsFreeSigData`, which may free up any remaining storage.

Revalidating signatures

If the user reopens the file, the signatures must be validated. If the user asks to validate one or more signature fields, the digital signature plugin sequences through them one at a time. Your `DSValidateSignProc` callback method must perform the following tasks:

- Recalculate any document digest that it desires, using the `DigSigFileGetEOF`, `DigSigFileSetPos`, and `DigSigFileRead` callbacks; or it may use the `DigSigMD5ByteRange` callback.
- Compare this result to the stored one, and do any other method-specific checks it desires.
- Optionally do a validation against some stored (network) registry.
- Update the `/AP /N` value in the signature annotation dictionary to show doublechecked/pass/fail symbol.
- Return doublechecked/pass/fail.

The user may open more than one document at a time, and may switch between open documents.

Additional available callbacks

The user may ask to show a signature panel containing summary information for each signature in an open document. If multiple documents are open, there may be multiple panels, or a single panel may be repainted as the user switches between documents. DigSig manages updating the panel(s), but may call the respective method plugin for each signature to get information to display on the panel. For each signature, the signature panel has two levels of detail:[Same comment as on page 15 about the following not being a sequential list.]

1. CLOSED displays a doublechecked/pass/fail/unknown/blank icon and a line of text for each signature field in the document. The default text is the name of the person signing and the date and time of signing, displayed in a language-independent way.

2. The digital signature plugin calls `dsGetValidState` to choose which icon to show.

3. OPEN displays an icon and line of text for each signature, then indented lines of further text, currently consisting of the name of the signer, date and time of signing, location of signing, reason for signing, and signing method.

4. The digital signature calls `dsGetValidState` to choose which icon to show.

5. Your plugin may update the signature panel for a document asynchronously (it may be doing validation as a background or idle-loop task). To do this, use the `DigSigUpdatePanel` callback.

Additional plugin support

Whenever a signature is created or verified, the plugin may optionally alter the appearance of the signature in the document, for the purpose of displaying or printing. For example, it could change an overprinted question mark on an unverified signature to an underprinted logo for a verified signature. To help with this, DigSig provides an HFT callback `DigSigGetStdXObj` that returns an XObject for a blank appearance, a question mark, or a cross. These are suitable as targets of the Do operator in a signature's appearance stream.

To avoid saving a signature to a file with an appearance of valid (rather than unvalidated), just before each file save, DigSig loops through all the signature fields and calls the specific method's `dsUnValidateSig` entry. This routine restores the signature's appearance to the unvalidated state.

The AcroForms Widget Annot handler calls into DigSig using four entries. These calls all reflect user actions taken in the document view, not the Signatures panel view.

When the user selects an annotation by tabbing to it or by clicking it with the mouse, and that annotation is for a signature field, AcroForms calls DigSigDraw. If the annotation is selected, then `bIsSelected` is true. When the user tabs to a signature annotation and activates it by hitting the spacebar or enter key, this is equivalent to a left mouse click.

AcroForms calls `DigSigKeyDown`. The parameters parallel those of `AVAnnotHandlerDoKeyDownProc`. When the user left-clicks inside a signature annotation, AcroForms calls `DigSigClick`. The parameters parallel those of `DoClickProcType`.

When the user right-clicks inside a signature annotation, AcroForms calls DigSigRightClick.

Rollback support

There is a constraint on the values in the /ByteRange array. This constraint allows DigSig to implement rollbacks prior to signatures.

The largest offset + length value in the /ByteRange array for a given signature must be equal to the length of the PDF file containing that signature; that is, it must equal offset + 1 of the "F" in the %%EOF at the end of the file.

In addition, the following constraints also apply:

- All offsets must be in the range 0..2147483647.
- All lengths must be in the range 1..2147483647.
- Offset[n+1] must be strictly greater than offset[n] + length[n].

Forms extended API

The Acrobat Forms plugin exports its own Host Function Table (HFT), whose methods can be used by other plugins. To use the Acrobat Forms plugin's HFT, a plugin must:

- Include the FormsHFT.h header file (which includes AF_ExpT.h and AF_Sel.h).
- Import the HFT using the `ASExtensionMgrGetHFT` method. A convenient way to perform this task is to use the `Init_AcroFormHFT` macro defined in FORMSHFT.H.

```
#define Init_AcroFormHFT
ASExtensionMgrGetHFT(ASAtomFromString(AcroFormHFT_NAME),
AcroFormHFT_LATEST_VERSION)
```

- Assign the HFT returned by this call to a plugin-defined global variable named `gAcroFormHFT`.

Data may be imported and exported into Acrobat Forms in forms data format (FDF). FDF is used to submit form data to a server, as well as to receive the response and incorporate it into a form. FDF is based on PDF and uses the same syntax and set of basic object types as PDF. It also has the same file structure, except that the cross-reference table is optional. See the *PDF Reference* for more information about the structure of a PDF document.

For information about the APIs included in the Forms extended API, see the Acrobat and PDF Library API Reference.

Weblink extended API

A link in a PDF document that references a URL is referred to as a Weblink.

The Acrobat Weblink plugin exports its own Host Function Table (HFT), whose methods can be used by other plugins. The HFT's name is defined in the WLHFTNAME macro, and its version number is `WEB_LINK_HFT_LATEST_VERSION`.

To use the Weblink plugin's HFT, a plugin must include the header file WeblinkHFT.h. The plugin must also import the HFT using `ASExtensionMgrGetHFT` and assign the HFT returned by this call to a plugin-defined global variable named `gWLHFT`. The easiest way to do this is to use the `Init_gWLHFT` macro defined in the header files.

For information about the APIs included in the Weblink extended API, see the Acrobat and PDF Library API Reference.

Weblink services

The Weblink plugin provides the following services:

- Maintenance of links (editing and storage of URLs associated with links, and so on)
- Manipulation of links (appropriate cursor changes and dynamic display of URL destinations)
- Selection of the external web browser
- Manipulation of the Adobe standard web driver
- Basic progress status services (progress monitor, wait cursor, and so on)

The Weblink plugin includes a standard driver, known as the Adobe Standard Web Driver. It allows support for transport mechanisms or web browsers to be added at a later time.

The Standard Web Driver uses DDE messages and Apple events to communicate with a web browser. It supports a protocol that consists of a suite of verbs—some going to and some coming from—the web browser. These verb definitions are provided so that web browsers can implement this protocol to be compatible with the Adobe standard web driver. Each verb is specified in terms of the platform-specific implementation: DDE for Windows and Apple events for Mac OS. The standard driver's use of each verb is also described. Browsers that wish to use their own protocol may do so by writing a custom driver.

The Weblink plugin communications software in the Weblink driver is independent of the Acrobat mechanism for handling links (the PDF implementation of URLs). This separation improves portability by isolating the highly platform-specific interapplication communication messages. Even on a given platform, there is no standard among web browsers for handling interapplication communication, and the actual transport mechanism may vary over time. By separating out the transport code, the Weblink plugin

remains portable across platforms, across different vendors' implementations of web browsers, and across different versions of web browsers from the same vendor.

Writing a custom driver

A driver is an Acrobat core plugin, written like any other plugin. A driver must register itself with the Weblink plugin during the import, replace, and register phases of the plugin initialization process by invoking `RegisterWebDriver`. You pass this method a `WebDriverVector` structure containing a version number and six pointers to functions that your driver provides to handle web-browser-specific tasks.

A driver is responsible for performing the following tasks:

- Connecting with external services (either directly or through an external application)
- Communicating with external services
- Associating a base URL with a given document
- Identifying external browsers that are compatible with the driver

In a typical session, the following actions can occur:[Same question as page 15. Is the following numbered list sequential?]

1. The user starts Acrobat.
   - The Weblink plugin publishes a Host Function Table (HFT) during the `exportHFTsCallback` phase of initialization.
   - During the `importReplaceAndRegisterCallback` phase, all drivers in turn invoke `RegisterWebDriver` in the Weblink plugin's HFT to register themselves as available.
   - During the `initCallback` phase, the Weblink plugin, if possible, selects an appropriate driver and notifies it that it is the active driver.
2. The user opens a PDF document with Weblinks and clicks a Weblink.
   - The Weblink plugin extracts the URL from the link and passes it to the driver.
   - The driver packages the URL into an interapplication communication (IAC) message and sends it to an external web browser (launching the browser application, if necessary).
   - The external web browser brings itself to the foreground unless the URL's MIME type is application/pdf.
3. The web browser retrieves the document and packages an IAC message.
   - The driver accepts the IAC message from the browser and opens the PDF document by using the `AVDocOpenFromFile` method. The driver should associate the URL with the document.
   - To resolve relative links, Weblink prepends either a base URL with the document, or if there is no base URL, the appropriate portion of the URL of the document the link is in.

Spelling extended API

Acrobat provides a Spelling plugin, which exports a Host Function Table (HFT) implementing a spell-check API for use by plugin developers.

To use the spelling HFT, a plugin must include the header file SpellerHFT.h, which includes Speller_Sel.h.

The following is a typical sequence of calls made by a plugin using the Spelling HFT. During its `importReplaceAndRegister` callback, the plugin should:

- Import the HFT, using `ASExtensionMgrGetHFT`, and assign the HFT returned by this call to a plugin-defined global variable named `gSpellerHFT`. The easiest way to do this is to use the Init_SpellerHFT macro defined in SpellerHFT.h.
- Allocate and initialize one `SpellCheckParam` block for each spelling domain the client will add.
- Add zero or more domains using the `SpellAddDomain` call.

During execution, a plugin performs the following tasks:

1. Respond to the following callbacks for each domain:
   - `SCEnableProc` is called by Spelling to ask if this domain has anything that needs to be checked
   - `SCGetTextProc` is called to get a text buffer to be checked.
   - `SCCompletionProc` is called after the user has modified the text buffer.
2. The client may call other Spelling HFT services during execution even if it did not add a domain.

During its `unloadCallback`, a plugin should perform the following tasks:

1. Remove all spelling domains added during initialization using the `SpellRemoveDomain` method.
2. Free all memory associated with `SpellCheckParam` block(s) (`scInBuffer`, `scOutBuffer`, and `scClientData`).
3. Free the `SpellCheckParam` block(s).

Several of the Spelling API methods (`SpellCheck`, `SpellCheckText`, and `SpellCheckWord`) take input strings as parameters, and several methods return strings as output parameters.

Input strings are either big-endian Unicode strings with the bytes 0xFE 0xFF prepended, or strings with `PDFDocEncoding`. In either case a string is expected to have the appropriate null-termination. If a string is UCS-2 it may have embedded language and country information.

Output text is in big-endian UCS-2 format with the bytes 0xFE 0xFF prepended. This string can be converted to a host encoded string by using the `ASTextFromPDText` and `ASTextGetEncodedCopy` methods.

```
char **altArray = NULL;
ASInt32 altCount = 0;
ASBool status = SpellCheckWord(acd, cWord, NULL, 0, &altArray, &altCount);
if (altCount) {
   ASText ast = ASTextFromPDText(altArray[1]);
   char* altWord = ASTextGetEncodedCopy(ast, (ASHostEncoding)
   PDGetHostEncoding() );
}
```

AcroColor extended API

AcroColor is an HFT that allows you to access the Adobe Color Engine (ACE), which provides color profile management for Acrobat and for other Adobe applications. Plugins can import the AcroColor HFT to use the color management methods.

The AcroColor extended API is the only extended API that is not installed as a plugin. It is part of the Acrobat core, but is considered an extended API. The AcroColor APIs, unlike the other extended APIs, can be used by the PDF Library.

The AcroColor HFT encapsulates color management into a set of convenient objects and functions. The objects represent basic color-management entities:

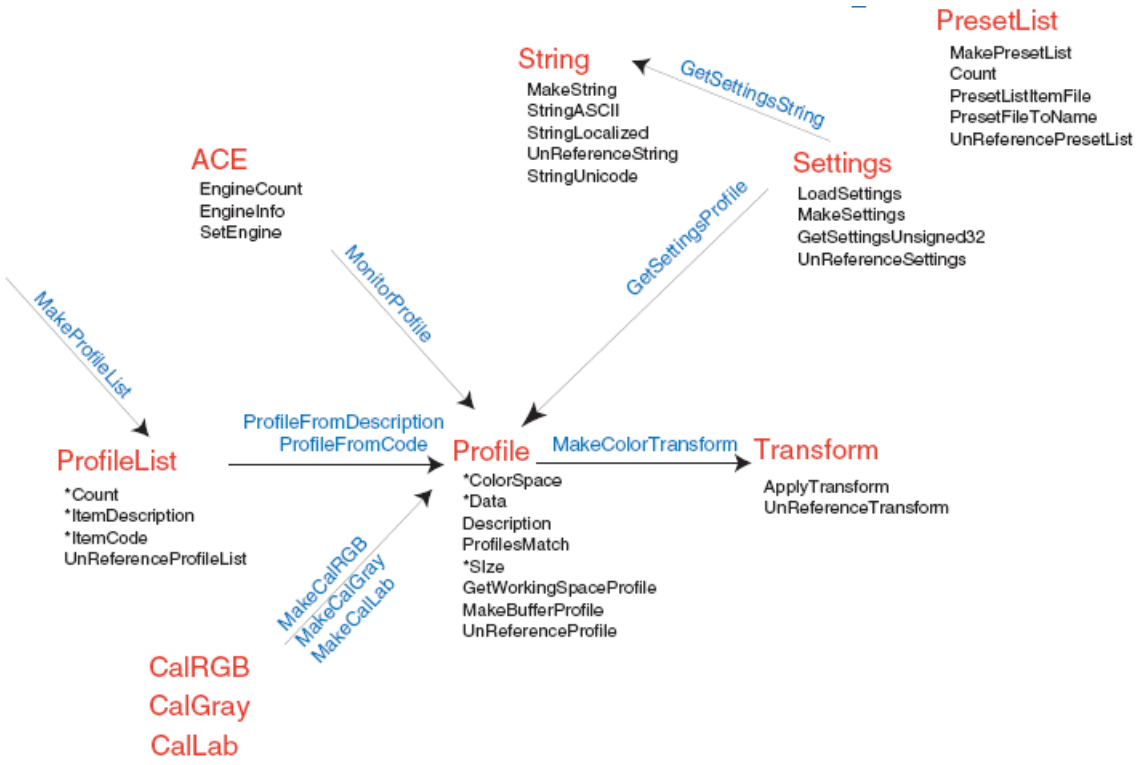
- The color management engine, or ACE, which is used by the underlying software to control a color management session.
- Device-specific ICC color profiles, which provide specific mapping between standard color specifications and specific values for particular output devices that produce those colors. Additional support objects include profile lists.
- Color spaces for the different kinds of color production (such as grayscale, RGB, and CMYK). Additional support objects include calibrated color spaces for standard color specifications.
- Transformations between profiles or color spaces.
- Color settings, as listed in the Acrobat Preferences. Color settings files contain, for instance, references to color profiles, and apply across Adobe products. Additional support objects include a string object and preset lists of settings.

You can create an ICC color profile from available data (`ACMakeBufferProfile`), or use profiles that are installed on the system (`ACGetWorkingSpaceProfile`), or stored in color settings files (`ACGetSettingsProfile`).

You can extract information directly from profiles, such as a string to use in the UI (`ACProfileDescription`). However, the most important thing you do with color profiles is use them to make transformations (`ACMakeColorTransform`). You can then apply it (`ACApplyTransform`) to transform a set of image data from one profile to another, so that it appears with the same colors on a different display device.

AcroColor objects are reference-counted. Each object type has an unreference method (such as `ACUnReferenceProfile`). Whenever you create one of these objects, you are responsible for using the corresponding unreference method to release it when you are finished with it.

The following diagram shows objects and their relationships.



Color conversion operations

The Color Conversion API has been extended in Acrobat 8.0 to include new APIs that enable you to add color conversion operations to your plugin or PDF Library applications. By using this extended API, you can convert a page based on a list of conversion actions. Each conversion action contains a set of matching fields (object attribute or type, color space, rendering intent), what the action should do if an object matches the matching fields, and conversion parameters (rendering intent, black point compensation, and target color space).

The list of conversion actions is evaluated in order. For example, a list could contain the following actions:

- Convert JPEG images to CMYK.
- Convert all images to CMYK.
- Convert line art using saturation intent.

Object attributes

An object located within a PDF document can contain the following attributes:

- Image (for example, JPEG/JPEG2K, lossless)
- Line art (for example, fill or stroke)
- Text
- Smooth shade
- Transparency
- Overprinting

Color space attributes

The following list describes color space attributes:

- Color space (RGB, CMYK, grayscale, Lab)
- Calibrated
- Device (for example, not calibrated)
- Special (Separation/DeviceN)
- Indexed
- NChannel
- Alternate color space
- Base of indexed color space

Conversion actions

The following is a list of conversion actions:

- Convert to a color space
- Preserve the object as it is
- Alias a separation to a different one
- Decalibrate the object, if possible (for example, replace calibrated spaces with device). This does not work with Lab color spaces.
- Downconvert from NChannel to DeviceN

Action modifiers

The following action modifiers apply if the action converts the object:

- Render Intent:
  - Override the color conversion with one of the ICC intents
  - Use document intent
- Preserve black
- Black point compensation: on or off
- Embed or do not embed the target profile if the object was converted

Ink aliasing

Along with the list of actions, there is a list of inks, such as specific colorants, which can control whether a particular ink is converted to process or aliased to another colorant.

Data structures

The AcroColor extended API contains methods, such as `PDDocColorConvertPage`, that accept data structure instances as arguments. These data structures consist of a list of action records and a list of inks. Each action record specifies attributes, color spaces, and rendering intent, along with an action. That is, what to do with the particular object if a match is located. The ink list defines ink aliasing or conversion to process for particular named colorants.

The following list specifies the data structures that you use to work with the AcroColor extended API:

PDColorConvertAction: Defines a color conversion action for a combination of attributes, color space, and rendering intent.

PDColorConvertParams: Represents a list of actions that will be performed.

For information about these data structures and their data members, see the Acrobat and PDF Library API Reference.

Data enum values

Some data structure members require enum values as values. For example, the `mAction` member, that belongs to the `PDColorConvertAction` data structure, requires a `PDColorConvertActionType` value. The following list specifies the data enum values that you use to work with the AcroColor extended API:

PDColorConvertObjectAttributeFlags: Specifies object attributes.

PDColorConvertSpaceTypeFlags: Specifies color space attributes.

PDColorConvertActionType: Specifies action types that occur when an object is matched.

For information about these enum values, see the Acrobat and PDF Library API Reference.

Converting a document to RGB

You can use the AcroColor extended API to convert a document to RGB by performing the following steps:

1. Create an instance of the `PDColorConvertParams` data structure.
2. Create an `AC_Profile` object. This object is used to assign a value to the `mConvertProfile` data member that belongs to the `PDColorConvertAction` data structure. When you are done with this object, invoke the `ACUnReferenceProfile` method to release it from memory.
3. Invoke the `ACProfileFromCode` method and pass the following arguments:
   - The address of the `AC_Profile` object.
   - The value `AC_Profile_AppleRGB` (this is an `AC_ProfileCode` value)
4. Create a `PDColorConvertActionType` variable and assign it the value `kColorConvConvert`. This variable is used to assign a value to the `mAction` data member that belongs to the `PDColorConvertAction` data structure.
5. Create an instance of the `PDColorConvertAction` data structure and assign the following values to its data members:

   mMatchAttributesAny: Assign `-1`

   mMatchSpaceTypeAny: Assign `-1`

   mMatchIntent: Assign `AC_UseProfileIntent` (an `AC_RenderIntent` value)

   mConvertProfile: Assign the `AC_Profile` object

   mEmbed: Assign `true`

   mPreserveBlack: Assign `false`

   mUseBlackPointCompensation: Assign `true`

   mAction: Assign the `PDColorConvertActionType` variable

6. Assign the following values to the PDColorConvertParams data members:

   mActions: Assign the instance of the `PDColorConvertAction` data structure to this data member

   mNumActions: Assign `1`

   mInks: Assign `NULL`

   mNumInks: Assign `0`

7. Invoke the `PDDocColorConvertPage` method and pass the following arguments:
   - A `PDDoc` object that represents the document in which to convert a page. (See "Creating a PDDoc object" on page 76.)
   - The instance of the `PDColorConvertParams` data structure that describes how color conversions are performed.
   - An `ASInt32` value that specifies the page number to convert. This value is a 0-based index.
   - An `ASProgressMonitor` object that represents the progress monitor callback. You can pass `NULL` if you do not want to provide a progress monitor callback.
   - The data element to pass to the progress monitor callback. You can pass `NULL` if you do not want to provide a progress monitor callback.
   - A `PDColorConvertReportProc` object that represents the reporting callback. You can pass `0` to indicate that there is no reporting callback.
   - The data element to pass to the reporting callback. You can pass `NULL` if you do not want to provide a reporting callback.
   - The address of an `ASBool` variable. If a conversion is made to the specified page, `true` is assigned.

The following code example converts a page in a PDF document to Apple RGB.

Example: *Converting a page in a PDF document to Apple RGB*

```
//Define the color parameters
PDColorConvertParams myColorParams;

//Define the color actions
PDColorConvertAction myAction;

//Declare an AC_Profile object
AC_Profile prof;

//Define AppleRGB as the profile to use
ACProfileFromCode(&prof, AC_Profile_AppleRGB);

//Declare a PDColorConvertActionType variable
PDColorConvertActionType actionType =  kColorConvConvert;

//Populate the PDColorConvertAction data members
myAction->mMatchAttributesAny = -1;
myAction->mMatchSpaceTypeAny = -1;
myAction->mMatchIntent= AC_UseProfileIntent;
myAction->mConvertProfile=prof ;
myAction->mEmbed = true;
myAction->mPreserveBlack = false;
myAction->mUseBlackPointCompensation= true;
```

```
myAction->mAction = actionType;

//Populate the PDColorConvertParams pointer
myColorParams->mActions=myAction;
myColorParams->mNumActions=1;
myColorParams->mInks= NULL;
myColorParams->mNumInks=0;

//Convert the second page to Apple RGB
PDDocColorConvertPage (theDoc, myColorParams, 1, NULL, NULL,0,NULL,false);

//Deallocate the AC_Profile object
ACUnReferenceProfile(prof);
```

For information about the APIs included in the AcroColor extended API, see the Acrobat and PDF Library API Reference.

PDF Optimizer API

PDF Optimizer API is part of the AV layer and is exported using the AcroView HFT. You use this API to work with the PDF Optimizer tool, which optimizes an active PDF document and then saves it using the `PDDocSaveWithParams` method to a specified location. The PDF Optimizer API is available with Acrobat Pro and Acrobat Pro Extended, but not with Acrobat Standard or with Adobe Reader.

Using this API, you can reduce the size of bulky PDF files and run Distiller optimizations on PDF files without having to print them. Avoiding the print route enables you to retain bookmarks, tags, links, and so on. You can also make PDF files compatible with specific versions of Acrobat.

You can invoke the `AVDocSaveOptimized` method to run the PDF Optimizer tool on a specified PDF document. An optimized document is created using the settings specified in the `PDFOptParams` structure. The optimized document is saved to disk at the location specified in the parameter's structure. If the operation is successful, the active document is closed and the optimized document is opened for viewing. If the operation fails, the active document remains open.

The `AVDoc` object passed to the proc should not be dirty. PDF Optimizer is unavailable in external windows like those of a web browser, so the `AVDoc` object should not be from a document open in an external window. The document should not be of a version greater than the default PDF version of the Acrobat application.

When you invoke the `AVDocSaveOptimized` method, pass the following arguments:

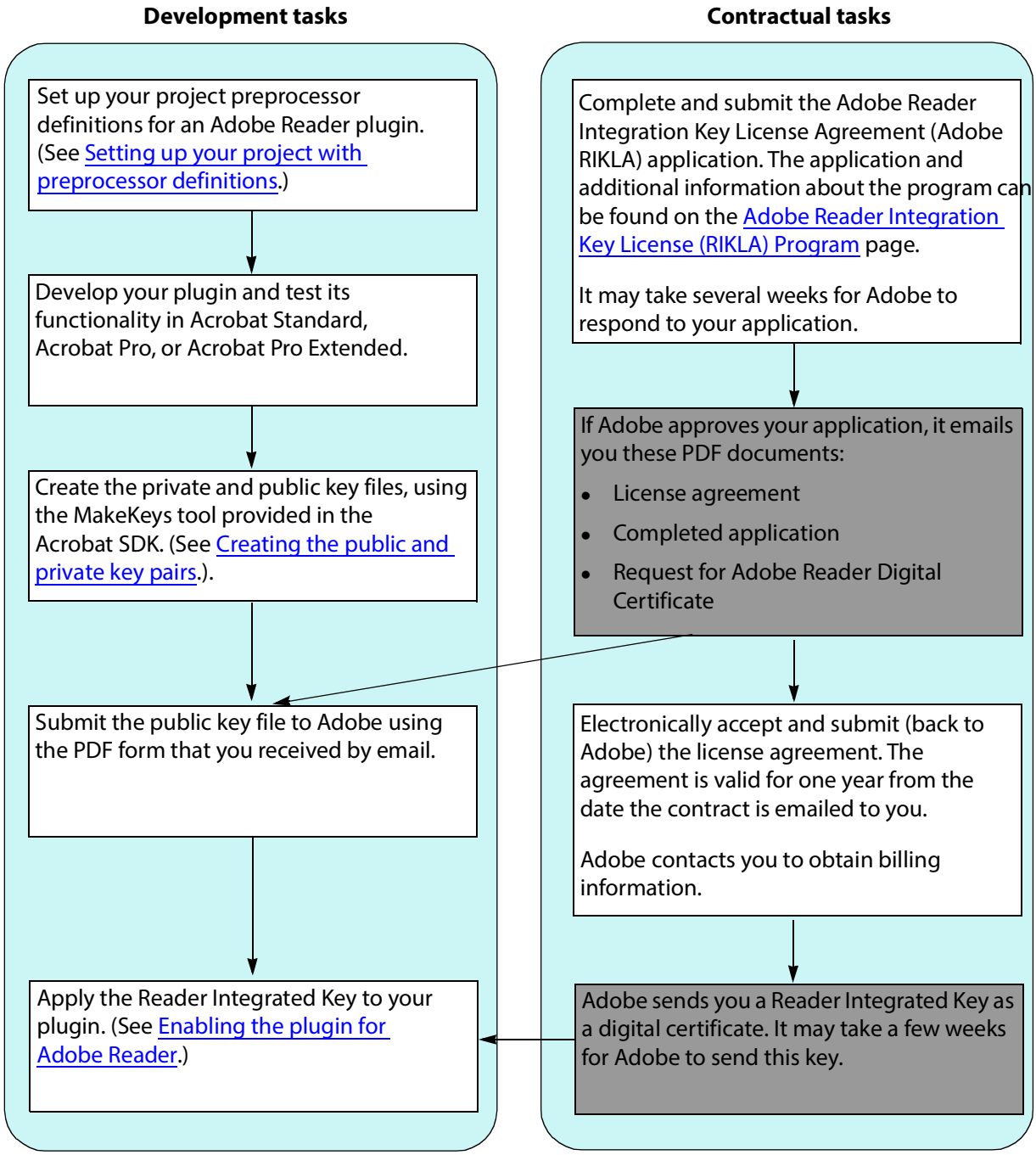
- An `AVDoc` object that represents the PDF document to optimize.
- An instance of a `PDFOptParams` data structure. (See the Acrobat and PDF Library API Reference.)

Creating an Adobe Reader plugin

Developing for Reader requires that you:

- Set up definitions for your plugin project
- Obtain the public key that Adobe uses to create your Adobe Integrated Key
- Apply that key to a plugin.

The following diagram illustrates the development and contractual tasks required to develop and enable a plugin to work with Adobe Reader. The gray-filled boxes indicate tasks that Adobe performs.



Setting up your project with preprocessor definitions

Before building an Adobe Reader plugin, you must set up your project properties with preprocessor definitions for `READER_PLUGIN`.

A common cause of difficulty when Reader-enabling a plugin is that the plugin attempts to acquire one or more Host Function Tables (HFTs) during startup that are not available within Adobe Reader. To avoid this problem on Windows and Mac OS, ensure that the `READER_PLUGIN` symbol is defined. With this symbol

defined, only those HFTs available in Adobe Reader are acquired. (See "Working with Host Function Tables" on page 156.)

To define the READER_PLUGIN symbol in a Mac OS project:

Add the `READER_PLUGIN` symbol to the Xcode project-level build settings by selecting the plugin project in the Groups & Files pane and opening the Project Inspector. On the Build tab, under the GCC 4.0 Preprocessing settings, add `READER_PLUGIN=1` to the Preprocessor Macros setting.

To define the READER_PLUGIN symbol in a Visual C++ project (Windows):

To define the READER_PLUGIN symbol in a Win32 project:

1. Locate the `AcroDspOptions.rsp` file in the SDK's `/PluginSupport/Samples` directory.
2. Open the `AcroDspOptions.rsp` file in a text editor.
3. Change ACRO_SDK_LEVEL to 0x000A0000 from 0x00090000
4. Remove the `AcroColorHFT` define by removing the following line from the file:

   ```
   /D PI_ACROCOLOR_VERSION=0x00060000
   ```

5. Add the READER_PLUGIN define by adding the following to the file:

   ```
   /D READER_PLUGIN
   ```

6. Save the `AcroDspOptions.rsp` file.
7. Open and rebuild your Visual C++ plugin project.

Note: This change affects all SDK plugins built after the change is made.

Creating the public and private key pairs

Use the MakeKeys tool to create the public and private key pairs for your plugin.

The MakeKeys tool is a command line (Terminal tool). You can find this tool in the PluginSupport/Tools/Reader-enabling Tools directory in the Acrobat SDK. You can also download a ZIP file that contains this tool from http://www.adobe.com/go/rikla_tools.

To create public and private key pairs:

1. Open a Command Prompt (Windows) or Terminal (Mac OS).
2. Go to the directory that contains the MakeKeys tool.
3. Run the MakeKeys tool, providing it with the following parameters:
   - `-kp` option followed by the name of the file in which to store the key pair. The directory must already exist. The file name cannot exceed the platform limitations on file name lengths.
   - `-pk` option followed by the name of the file in which to store the public key. The directory must already exist. The file name cannot exceed the platform limitations on file name lengths.

- A random string that you create. The MakeKeys tool uses the string to generate a random number. The random string must contain only letters and digits. There is no limit on the length of the string.

The following example command line shows the options and parameters you supply to the MakeKeys tool. The italics in the command line indicate placeholders that you must replace with meaningful values.

```
MakeKeys -kp KeyPairFileName -pk PubKeyFileName RandomString
```

The size of the resulting key pair file is approximately 450 bytes. The size of the public key pair file is approximately 98 bytes.

4. Submit the public key file to Adobe using the PDF form that you received by email, as described in Developing and enabling an Adobe Reader plugin.

Enabling the plugin for Adobe Reader

After receiving the Reader Integrated Key from Adobe, use the SignPlugin tool to enable the plugin for Adobe Reader.

The SignPlugin tool is a command line (Terminal tool). You can find this tool in the PluginSupport/Tools/Reader-enabling Tools directory in the Acrobat SDK. You can also download a ZIP file that contains this tool from http://www.adobe.com/go/rikla_tools.

To enable the plugin for Adobe Reader:

1. Open a Command Prompt (Windows) or Terminal (Mac OS).
2. Go to the directory that contains the SignPlugin tool.
3. Run the SignPlugin tool, providing it with the following parameters:
   - `-kp` option followed by the location of the file that contains the key pair. This file was produced by the MakeKeys tool. (See Creating the public and private key pairs.)
   - `-cf` option followed by the location of the file that contains the Reader Integrated Key provided by Adobe. (See Developing and enabling an Adobe Reader plugin.)
   - Location of the plugin.

   The following example command line shows the options and parameters you supply to the SignPlugin tool. The italics in the command line indicate placeholders that you must replace with meaningful values.

   ```
   SignPlugin -kp keypairFileName -cf Reader_Integrated_Key_FileName
   /MyDirectory/MyPlugin
   ```

   If SignPlugin is successful, it returns the message: "Plugin.acroplugin successfully Reader Enabled."

The plugin can now be loaded by Adobe Reader. If you encounter any difficulties, see "Troubleshooting an Adobe Reader plugin" on page 283.

Note: This procedure must be repeated each time the plugin is built.

The SignPlugin has been been upgraded to a 64 Bit executable to support running on Mac OS Catalina.

Tip: On Mac OS, to easily copy paths to the command line, drag files from the Finder window to the Terminal window.

Troubleshooting an Adobe Reader plugin

There are some issues that can cause a plugin not to load in Adobe Reader. The most common issues are documented here. If the problem persists, contact Acrobat Developer Support.

Plugin appears to be ignored by Adobe Reader

Several problems can cause this symptom:

- The plugin was compiled without the `READER_PLUGIN` symbol being defined (Windows and Mac OS platforms only).

  A common cause of difficulty when enabling a Adobe Reader plugin is that the plugin attempts to acquire one or more HFTs during startup that are not available in Adobe Reader. To avoid this problem on Windows and Mac OS, ensure that the `READER_PLUGIN` symbol is defined. (See Setting up your project with preprocessor definitions.)

  With this symbol defined, only those HFTs available in Adobe Reader are acquired. For information about an HFT, see "Working with Host Function Tables" on page 156.

- The Certified Plugins Only preference is set (Windows and Mac OS platforms only).

  Adobe Reader recognizes only Adobe plugins as certified.

  Ensure that the Certified Plugins Only option is not selected in the Options Preferences dialog box.

- The plugin was not correctly enabled.

  The resources file sent by Adobe may have been corrupted during delivery. Verify that the file sizes match those documented in the previous sections. If the public/private key pair or public key files are corrupt, you must regenerate new files and request a new Reader Integrated Key from Adobe. If the file that contains the key is corrupt, you must request a new Reader Integrated Key from Adobe, using the existing public key file.

Adobe Reader error messages

The following are the most commonly received error messages:

- "There was an error while loading the plugin name.api. The plugin is incompatible with this version of the viewer."
- "There was an error while loading the plugin name.api. The plugin failed to initialize."
- "There was an error while loading the plugin '*your plugin name*'. Two plugins are attempting to register with the same name."

The first error is displayed if the plugin returns `false` from the `PISetupSDK` method (defined in PIMain). The method returns `false` if the plugin attempts to acquire an HFT that is not available. (See "Working with Host Function Tables" on page 156.)

23 Plugins

The Acrobat core and extended APIs allow you to write plugins that integrate with Adobe Reader. For detailed information on the API architecture, methods, and usage, see Overview and the Acrobat and PDF Library API Reference.

Reader enablement

Any plugins written for Adobe Reader must be Reader-enabled, which means that you will need to obtain permission and licensing from Adobe Systems.

When developing a Reader-enabled plugin, follow the steps described in Creating an Adobe Reader Plug-In to make specific changes to your plugin code in order for Adobe Reader to recognize and load it. For information on obtaining a license key for enabling your Reader plugin, see the site for the Acrobat DC Reader Integration Key License (RIKLA) Program at http://www.adobe.com/go/rikla_program.

A Reader-enabled plugin is a dynamically linked extension to Adobe Reader created using C/C++ APIs, and can be developed for any supported platform:

- DLLs on Windows (using the extension `.api`)
- Shared libraries (code fragments) on Mac OS X

Note: Don't confuse *Reader enablement* with *rights-enabled PDF*. Reader enablement enables a plugin to work with Adobe Reader. A rights-enabled PDF turns on additional user features in Adobe Reader, such as the ability to save forms offline.

APIs available for Adobe Reader

Host Function Tables (HFTs) are tables of function pointers, essentially providing a means by which plugins call methods in Adobe Reader. The following HFTs are available for development with Adobe Reader:

- `AcroSupport`
- `AcroView`
- `AcroViewSweetPea`
- `ASExtra`
- `Catalog`
- `Cos`
- `PDModel`
- `ASExtra`
- `PDSRead`
- `AcroSupport`
- `Core`
- `Forms`
- `TTS`
- `DigSigHFT`
- `AcroHLS`
- `PubSecHFT`
- `Search`
- `WebLink`

HFTs are described in Understanding Plug-ins and Working with Host Function Tables. For information about the HFTs and the APIs, see the Acrobat and PDF Library API Reference. For details regarding the additional usage rights that may be applied to a PDF document, see Understanding your target application.

Index

U

V

X

Z