Adobe Acrobat 7.0.5

Acrobat Interapplication Communication Overview

July 27, 2005

Adobe Solutions Network — http://partners.adobe.com

Contents

Preface

Adobe® Acrobat® provides support for interapplication communication (*IAC*) through Apple® events and AppleScript on Macintosh® platforms and through OLE and DDE on Windows® platforms. This support allows programs to control Acrobat and Adobe Reader® in much the same way a user would. You can also use the IAC support to render a PDF file into any specified window instead of the Acrobat or Reader window. The IAC support methods and events serve as wrappers for some of the core API calls in the Acrobat SDK. Thus, IAC supports enterprise workflows by making it possible to control Acrobat and Reader, display PDF documents in other applications, and manipulate PDF data from other applications.

What Is In This Document

This document explains the IAC support concepts, such as objects and commands universally understood by applications. This document is divided into chapters that parallel the different technologies:

- The overall IAC architecture in Chapter 1, "Architecture"
- Apple Events in Chapter 2, "Apple Event Support."
- OLE in Chapter 3, "OLE Support."
- DDE in Chapter 4, "DDE Support."

You only need to read the chapters that are relevant to your own platforms and technologies.

In addition, the appendices provide useful information for migrating your IAC applications to Microsoft® Visual Studio .NET and taking advantage of the new Automation support using ATL interfaces.

Prerequisites

You should already be familiar with at least one of these technologies:

- Apple events
- AppleScript
- DDE
- OLE

If you are not, see the list of documents that describe them in Other Useful Documentation.

You should also know something of the Acrobat core API. The IAC capabilities are actually a subset of those provided in the Acrobat core API; many of the IAC messages are similar to core API methods.

Related Documents

The Acrobat SDK includes many other books that you might find useful. If for some reason you did not install the entire SDK onto your system and you do not have all of the documentation, please visit the Adobe Solutions Network web site (http://partners.adobe.com/asn/) to find the books you need.

Developer Documentation

The *Acrobat SDK User's Guide* describes the capabilities of the Acrobat SDK, and provides a general overview of its usage

Code Samples

The Acrobat SDK contains several examples of using IAC. See the *Guide to SDK Samples* for a list of all the samples.

Interapplication Communication Documentation

Acrobat Interapplication Communication Reference provides detailed information on the Apple Event, DDE, and OLE support in Acrobat.

Core API Documentation

Acrobat and PDF Library API Overview gives an overview of the objects and methods in the core API.

Acrobat and PDF Library API Reference describes in detail the objects, methods and callbacks in the core API.

File Format Documentation

PDF Reference provides the authoritative, detailed description of the industry-standard PDF file format.

Platform-Specific Documentation

Inside Macintosh: Interapplication Communication, ISBN 0-201-62200-9, Addison-Wesley. This contains information on Apple events and scripting.

AppleScript Language Guide, ISBN 0-201-40735-3, Addison-Wesley. This contains more information on the AppleScript language.

Apple Event Registry: Standard Suites, by Apple Developer Technical Publications, Part number 030-1958-A. This contains more information on the core and required Apple events.

OLE 2 Programmer's Reference Volumes One and Two, ISBN 1-55615-628-6 and ISBN 1-55615-629-4, Microsoft Press. Volume One contains information on OLE 2.0; Volume Two covers OLE Automation.

Conventions Used in This Book

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

Font	Used for	Examples
monospaced	Paths and filenames	`C:\templates\mytmpl.fm`
	Code examples set off from plain text	These are variable declarations: `AVMenu commandMenu,helpMenu;`
monospaced bold	Code items within plain text	The **`GetExtensionID`** method ...
	Parameter names and literal values in reference documents	The enumeration terminates if **`proc`** returns **`false`**.
monospaced italic	Pseudocode	`ACCB1 void ACCB2 ExeProc(void)` `{ do something }`
	Placeholders in code examples	`AFSimple_Calculate(cFunction, cFields)`
blue	Live links to Web pages	The Adobe Solutions Network URL is: http://partners.adobe.com/asn/
	Live links to sections within this document	See Using the SDK.
	Live links to code items within this document	Test whether an **`ASAtom`** exists.
bold	PostScript language and PDF operators, keywords, dictionary key names	The **setpagedevice** operator
	User interface names	The **File** menu
italic	Document titles that are not live links	*Acrobat Core API Overview*
	New terms	*User space* specifies coordinates for...
	PostScript variables	*filename* **deletefile**

1 Architecture

The Acrobat core API exposes most of its architecture in C, although it is written to simulate an object-oriented system with nearly fifty objects. The IAC interface for Apple events and OLE automation exposes a smaller number of objects, which closely map those in the Acrobat API and can be accessed through Visual C++ .NET, Visual Basic .NET, and Visual C# .NET. This chapter provides an overview of those objects. You can read the *Acrobat Interapplication Communication Reference* for more detailed information of how to use IAC objects on your platform.

NOTE: DDE does not organize IAC capabilities around objects, but instead uses DDE messages to Acrobat.

Interapplication Communication Objects

You can think of the Acrobat API as having two distinct layers that use IAC objects:

- The Acrobat application (AV) layer.
- The Portable Document (PD) layer.

The AV layer enables you to control how the document is viewed. For example, the view of a document object resides in the layer associated with Acrobat. The PD layer provides access to the information within a document, such as a page. From the PD layer you can perform basic manipulations of PDF documents, such as deleting, moving or replacing pages, as well as changing annotation attributes. You can print PDF pages, select text, access manipulated text, and create or delete thumbnails.

In addition, it is also possible to treat a PDF document as an ActiveX document and implement convenient PDF browser controls through the **`AcroPDF`** object, which provides you with the ability to load a file, move to various pages within a file, and specify various display and print options. A detailed description of its usage is provided in OLE Automation Support.

In addition to the convenient **`AcroPDF`** object, IAC provides you with control over the application's user interface and appearance of its window by either using its PD layer object, **`PDPage`**, or by using its AV layer object, **`AVDoc`**. The **`PDPage`** object has a method called **`Draw`** which exposes Acrobat's rendering capabilities. If you need more fine-grained control, you can create your application with the **`AVDoc`** object, which has a function called **`OpenInWindow`** that can display text annotations and active links in your application's window.

Accessing the AV and PD Layers

The **Apple Events**, **Applescript**, and **OLE Automation 2.0** systems each refer to the objects with a different syntax. The following subsections provide tables that explain the objects and class names for each system. If you are using **Apple Events**, you will use the name of the object in a `CreateObjSpecifier` statement. If you are using **AppleScript**, you will use the object name in a `set ... to` statement. If you are using **OLE**, you will use the name in either a Visual Basic .NET or Visual C# .NET `CreateObject` statement or in an MFC `CreateDispatch` statement. If the name of the object appears in the table as "None," there is no corresponding class name for the object in that system.

Acrobat Application Layer

Table 1.1 displays IAC objects in the *Acrobat Application Layer*. The first three objects are the primary source for controlling the user interface.

TABLE 1.1 Acrobat Application Layer Objects

Object	Description	Apple Event Class Name	OLE Automation Class Name
`AVApp`	The top-level object, representing Acrobat. You can control the appearance of Acrobat, whether or not an Acrobat window appears, and the size of the application window. Your application has access to the menu bar and the toolbar through this object.	`Application`	`AcroExch.App`
`AVDoc`	Represents a window containing an open PDF file. Your application can use this to cause Acrobat to render into its window so that it closely resembles Acrobat's window. You can also use this object to select text, find text, or print pages. This object has several bridge methods to access other objects.	`Document`	`AcroExch.AVDoc`
`AVPageView`	Controls the contents of the AVDoc window. Your application can scroll, magnify, go to next, go to previous, or go to an arbitrary page. This object also holds the history stack.	`PDF Window`	`AcroExch.AVPageView`
`AVMenu`	Represents a menu in Acrobat. You can count or remove menus. Each menu has a language-independent name used to access it.	`Menu`	None
`AVMenuItem`	Represents a single item in a menu. You can execute or remove menu items. Every menu item has a language-independent name used to access it.	`Menu item`	None
`AVConversion`	A format in which to save the document.	`conversion`	None

Portable Document Layer

Table 1.2 displays the IAC objects in the Portable Document layer.

TABLE 1.2 Portable Document Layer Objects

Object	Description	Apple Event Class Name	OLE Automation Class Name
`PDDoc`	The underlying PDF representation of a document. Using this object, your application can perform operations such as deleting and replacing pages. You can create and delete thumbnails with this object, as well as set and retrieve document information fields. Note: With Apple Events the first page of a document is page 1; with OLE Automation the first page is page 0.	`Document`	`AcroExch.PDDoc`
`PDPage`	**`PDDoc`** objects are composed of **`PDPage`** objects. You can use this object to render Acrobat to your application's window. You can also access page size and rotation, set up text regions, create and access annotations through this object. Note: For Apple Events the first page of a document is page 1; for OLE automation it is page 0.	`page`	`AcroExch.PDPage`
`PDAnnot`	Used to manipulate link and text annotations. You can set and query about the physical attributes of the annotation. You can perform a link annotation with this object. Note: Apple Events have two additional, related objects: **`PDTextAnnot`**, a text annotation, and **`PDLinkAnnot`**, a link annotation.	`annotation`	`AcroExch.PDAnnot`
`PDBookmark`	This object represents bookmarks in the PDF document. You cannot directly create a bookmark, but if you know a bookmark's title, you can change its title or delete it.	`bookmark`	`AcroExch.PDBookmark`

TABLE 1.2 Portable Document Layer Objects

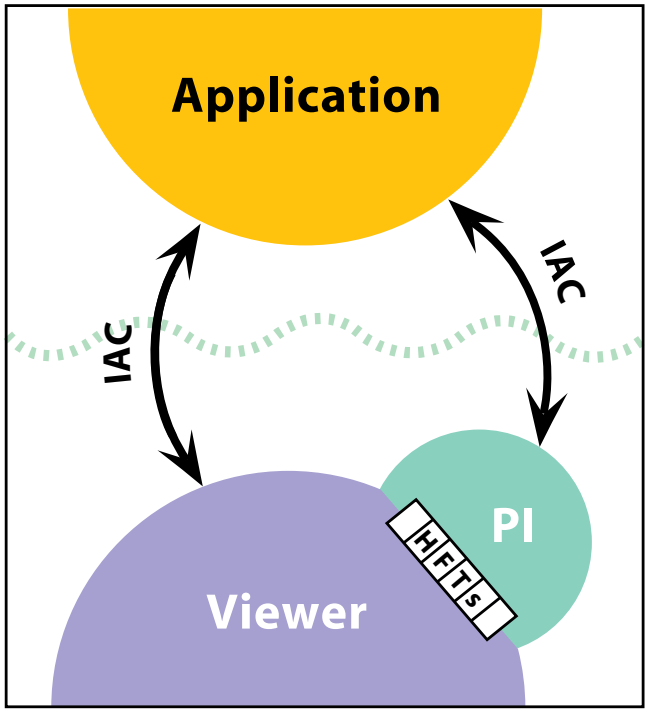
Object	Description	Apple Event Class Name	OLE Automation Class Name
`PDTextSelect`	Serves two purposes: If selected texts exists within an **`AVDoc`** object, your application can access the words in that region through this object, or you can cause text to appear selected.	None	`AcroExch.PDTextSelect`

Using Plug-ins for Interapplication Communication

The Role of Plug-ins in Extending IAC Interfaces

You can extend the functionality of the IAC interfaces by writing plug-ins that use core API objects that are not already part of the IAC support system. Figure 1.1 shows the software architecture needed to establish a connection.

FIGURE 1.1 Using Plug-Ins for Interapplication Communication



Using Acrobat JavaScript in InterApplication Communication

The `JSObject` interface provides you with convenient access to the powerful features offered by Acrobat JavaScript. It is recommended that you take advantage of this interface wherever possible, and its usage is explained in OLE Support.

2 Apple Event Support

Acrobat supports Apple Events and a number of Apple Event objects on the Macintosh platforms. IAC support includes some of the objects and events described in *Apple Event Registry: Standard Suites*, as well as Acrobat-specific objects and events. This chapter lists each of the supported events and objects. (You can find information on Apple Events supported by the Acrobat Search plug-in by referring to the *Acrobat and PDF Library API Reference*. Other plug-ins supporting additional Apple Events are described in the *Acrobat and PDF Library API Overview*.)

When programming for Macintosh platforms, it is advised that you use AppleScript with Acrobat whenever possible. There are some Apple Events not available through AppleScript; these can be handled with C or other programming languages. When programming in C, use the declarations in `AcroAETypes.h`. For a complete description of the parameters, see the *Acrobat Interapplication Communication Reference*.

Acrobat Objects In Apple Events

Acrobat presents these objects to the Apple Event interface:

- **`annotation`**: an annotation on a page of a document. The **`Text Annotation`** and **`LinkAnnotation`** classes are two specific annotation types.
- **`application`**: represents Acrobat itself.
- **`bookmark`**: a bookmark within a PDF document.
- **`conversion`**: a format in which to save a document. Any installed conversion (such as TIFF or JPEG) is supported, and Acrobat provides extra support for **`EPS Conversion`** and **`PostScript Conversions`**.
- **`document`**: represents a single open document.
- **`Link Annotation`**: a link annotation within the PDF document.
- **`menu`**: a menu in Acrobat.
- **`menu item`**: a single item in a menu within the PDF document.
- **`page`**: represents a single page in a PDF document.
- **`PDF Window`**: represents the view of a document in Acrobat's window.
- **`Text Annotation`**: a text annotation within the PDF document.

Acrobat Support For Apple Events

Acrobat supports four categories of Apple Events:

- *Required Events*: Events the Finder sends to all applications
- *Core Events*: Events common to a wide variety of applications, though not universally applicable to all applications.
- *Acrobat-specific Events*: Events that are specific to Acrobat.
- *Miscellaneous Apple Events*: Events that don't fall into one of the above categories

NOTE: Acrobat supports all of the above events, but Adobe Reader supports only the four required events: **`open`**, **`run`**, **`print`** and **`quit`**.

Required Events

The suite of required Apple Events consists of four events sent by the Finder to all applications:

- **`open`**: opens a file.
- **`run`**: launches an application and invokes its standard startup procedures.
- **`print`**: prints one or more files.
- **`quit`**: terminates an application. See the version of the quit Event in the core suite for a variant that accepts options.

Core Events

Acrobat supports the following subset of the core suite of Apple Events:

- **`close`**: closes a document.
- **`count`**: counts the number of elements of a particular class in an object.
- **`delete`**: deletes one or more objects.
- **`exists`**: tests whether a specified object exists.
- **`get`**: retrieves the value of a property of an object.
- **`make`**: creates a new object.
- **`move`**: moves a page object.
- **`open`**: opens one or more documents.
- **`quit`**: terminates Acrobat.
- **`save`**: saves a document to a file, optionally specifying the format in which to save the document.
- **`set`**: assigns one or more values to one or more variables.

Acrobat-specific Events

This suite contains Acrobat-specific Events and objects. Apple encourages the use of an application's signature as the name of its class for application-specific Apple Events, so **CARO** is the name of the class for Acrobat-specific Apple Events, as shown below. AppleScript does not need this information.

```
#define kAEAcrobatViewerClass 'CARO'
```

The supported Events in this suite are:

- **bring to front**: brings the specified document's window to the front.
- **clear selection**: clears the document's current selection, if any.
- **close all docs**: closes all documents.
- **create thumbs**: creates thumbnail images for all pages in the document
- **delete pages**: deletes a range of pages in the document.
- **delete thumbs**: deletes all thumbnails from the document.
- **execute**: executes the specified menu item as if the user had selected it.
- **find next note**: finds and selects the next text note in a document.
- **find text**: finds text in a document.
- **get info**: retrieves the value of a key in the document's **Info** dictionary.
- **go backward**: goes to the previous view in the stored view history.
- **go forward**: goes to the next view in the stored view history.
- **goto**: displays a specified page.
- **goto next**: displays the page after the one currently displayed in the **PDF Window**.
- **goto previous**: displays the page before the one currently displayed in the **PDF Window.**
- **insert pages**: inserts one or more pages from one document into another.
- **is toolbutton enabled**: tests whether the specified toolbutton is enabled.
- **maximize**: sets the document's window size to its maximum or original size.
- **perform**: executes a bookmark's or link annotation's action.
- **print pages**: prints one or more pages of a document.
- **read page down**: scrolls forward through the document by one screen.
- **read page up**: scrolls backward through the document by one screen.
- **remove toolbutton**: removes a button from the toolbar.
- **replace pages**: replaces one or more pages in a document with pages from another document.
- **scroll**: scrolls the page view by the specified amount.
- **select text**: selects the specified text.

- **`set info`**: sets the value of a key in the document's **Info** dictionary.
- **`zoom`**: changes the zoom level of the specified **`PDF Window`**.

Miscellaneous Apple Events

Acrobat provides the following Apple Event:

- **`do script`**: perform the specified Acrobat JavaScript script.

3 OLE Support

This chapter describes OLE 2.0 support in Adobe Acrobat for Microsoft Windows. Acrobat applications are OLE servers and also respond to a variety of OLE automation messages. Since Acrobat is an OLE server, you can embed PDF documents into documents created by an application that is an OLE client.

For complete descriptions of the Acrobat parameters associated with OLE automation methods, see the OLE automation sections of the *Acrobat Interapplication Communication Reference*.

NOTE: The header files containing values of various constants, needed by C and C++ programmers to use the OLE automation support described in this chapter, are located in the SDK IAC directory. Visual Basic .NET and Visual C# .NET users do not need these header files, though it may be useful to refer to them in order to verify the constant definitions.

Differences Among the Acrobat Applications

Acrobat supports all of the OLE automation methods listed in this chapter and in the *Acrobat Interapplication Communication Reference*. Adobe Reader does not support OLE automation, except for the PDF browser controls provided in the **`AcroPDF`** object.

How To Tell If an Acrobat Application Is Running

Use the Windows **`FindWindow`** method with the Acrobat class name. You can use **Spy++** to determine the class name for the version of the application.

OLE Server Support

Acrobat provides the appropriate OLE interfaces to be an OLE server. You can embed PDF documents or link them to OLE containers. However, Acrobat does not perform in-place activation.

NOTE: Acrobat 7.0 supports dual interfaces, so the methods all have a return type of **HResult**.

OLE Automation Support

Dual Interfaces

The automation interfaces for Acrobat 7.0 have been changed from MFC Disp interfaces to Dual interfaces. Thus, methods previously returning **LONG** values in place of boolean values in Acrobat 6.0 now return **BOOL** values (-1 for success, 0 for failure). The return values for such methods are equivalent to **VARIANT_TRUE** and **VARIANT_FALSE**.

For example, in Acrobat 6.0, a client using the **CAcroPDDoc** object's **Open** method would contain the following code:

```
int ret = pdfDoc.Open(inputFile);
```

Now, in Acrobat 7.0, the client would contain the following code:

```
bool ret = pdfDoc.Open(inputFile);
```

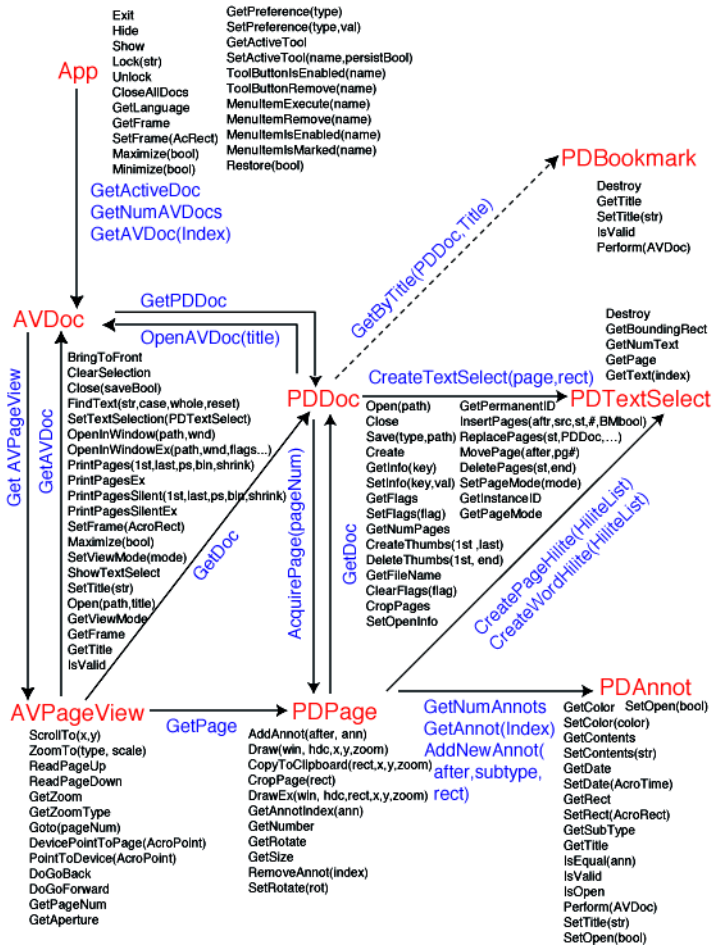
NOTE: It is only necessary to migrate previously written projects if they will refer to the Acrobat 7.0 type libraries.

OLE Objects and Methods

OLE automation support is provided by a set of classes in Acrobat's API.

Figure 3.1 is a graphical representation of the objects and methods that are used in OLE. The arrows indicate *bridge methods*, which are methods that can get an object from a related object of a different layer. For instance, you may want to get the **PDDoc** associated with a particular **AVDoc** object. In OLE automation, you can use the **GetPDDoc** method in the **AcroExch.AVDoc** object.

FIGURE 3.1 OLE Objects and Methods



What's Possible: Examples and Approaches

For OLE automation, Acrobat provides three capabilities: rendering PDF documents, remotely controlling the application, and implementing PDF browser controls. You can accomplish these tasks in a number of ways.

Rendering PDF Documents

You can render PDF files on the screen in two ways:

- Using an interface similar to Acrobat's user interface. The **AVDoc** object's **OpenInWindowEx** method opens a PDF file in your application's window. This window has vertical and horizontal scroll bars, as well as buttons on the window's perimeter for setting the zoom factor. Users interacting with this type of window find its operation similar to that of working in Acrobat, though not everything works the same way. For instance, links are active and the window can display any text annotation on a page.
- Using the **Page** object's **DrawEx** method. You provide a window and an **HDC,** as well as a zoom factor. Acrobat renders the current page into your window. The application must manage scroll bars and other items in the user interface. This approach works unless your application requires an **AVDoc** representation, in which case you should use the **AVDoc** object's methods.

The SDK provides two samples that illustrate these mechanisms. The `ActiveView` and `StaticView` samples use the **OpenInWindowEx** and **DrawEx** methods respectively.

NOTE: In the current implementation of Acrobat, when a user quits an application using OLE automation, this can have an effect on Acrobat itself as well as a web browser displaying PDF files:

- If there are no PDF documents open in Acrobat, the application will quit.
- If a web browser is displaying a PDF file, the display will go blank. (The user can refresh the page to re-display it.)

Remotely Controlling the Application

There are two ways to remotely control Acrobat:

- Given the exported interfaces, you can write an application that manipulates various aspects of PDF files, such as pages, annotations, and bookmarks. Thus, your application might use **AVDoc**, **PDDoc**, **Page**, and **annotation** methods, and might not provide any visual feedback that requires rendering into its application window.
- You can launch Acrobat from your own application, which has set up an environment for the user. Your application can cause Acrobat to open a file, set the page location, zoom factor, and possibly even select some text. This would be useful, for example, as part of a help system.

PDF Browser Controls

You may use the **AcroPDF** library to display a PDF document in applications using simplified browser controls. In this case the PDF is treated as an ActiveX document, and the interface is available in Adobe Reader.

The document may be loaded via the **AcroPDF** object's **LoadFile** method. Once this has been accomplished, you may implement browser controls to determine which page to display as well as the display, view, and zoom modes, whether to display bookmarks, thumbs, scrollbars, and toolbars, print pages using various options, and highlight a text selection.

Development Environments

You have a choice of environments in which to integrate with Acrobat: Visual Basic .NET, Visual C# .NET and Visual C++ .NET.

If possible, use Visual Basic .NET or Visual C# .NET. The run-time type checking offered by the **CreateObject** call in Visual Basic allows quick prototyping of an application, and in both cases the implementation details are simplified.

NOTE: In these examples you will see strings with "**AcroExch.App**" and strings with "**Acrobat.CAcroApp**". The first is the form for the external string used by OLE clients to create an object of that type. The second is the form that is included in developer type libraries.

Example 3.1 shows a Visual Basic subroutine to view a given page:

EXAMPLE 3.1 Visual Basic View Page

```
Sub Goto(Int where)
     Dim app as Object, avdoc as Object, pageview as Object

     Set app = CreateObject("AcroExch.App")
     Set avdoc = app.GetActiveDoc
     Set pageview = avdoc.GetAVPageView
     pageview.Goto(where)
End Sub
```

Example 3.2 shows the same route in Visual C++:

EXAMPLE 3.2 Visual C++ View Page

```
void goto(int where)
{
     CAcroApp app;
     CAcroAVDoc *avdoc = new CAcroAVDoc;
     CAcroAVPageView pageview;
     COleException e;
     app.CreateDispatch("AcroExch.App");
     avdoc->AttachDispatch(app.GetActiveDoc, TRUE);
     pageview->AttachDispatch(avdoc->GetAVPageView, TRUE);
     pageview->Goto(where);
}
```

Example 3.3 shows how to use PDF browser controls to view a page:

EXAMPLE 3.3 Visual Basic .NET using AcroPDF Browser Controls

```
Friend WithEvents AxAcroPDF1 As AxAcroPDFLib.AxAcroPDF
Me.AxAcroPDF1 = New AxAcroPDFLib.AxAcroPDF
'AxAcroPDF1
Me.AxAcroPDF1.Enabled = True
Me.AxAcroPDF1.Location = New System.Drawing.Point(24, 40)
Me.AxAcroPDF1.Name = "AxAcroPDF1"
Me.AxAcroPDF1.OcxState = CType(
        resources.GetObject("AxAcroPDF1.OcxState"),
        System.Windows.Forms.AxHost.State
)
Me.AxAcroPDF1.Size = New System.Drawing.Size(584, 600)
Me.AxAcroPDF1.TabIndex = 0
AxAcroPDF1.LoadFile("http://myURL/myFile.pdf")
AxAcroPDF1.setCurrentPage(TextBox2.Text)
```

The Visual Basic .NET examples are simpler to read, write, and support, and are similar in terms of implementation detail to the code required for Visual C# .NET. In Visual C++, the **`CAcro`** classes hide much of the type checking that must be done. Using OLE automation objects in Visual C++ requires understanding the **`AttachDispatch`** and **`CreateDispatch`** methods of the **`COleDispatchDriver`** class. See Using Visual Basic and Visual C# .NET with the Acrobat SDK for more information.

Using Visual Basic and Visual C# .NET with the Acrobat SDK

The only requirement for using the OLE objects made available by Acrobat is to have the product installed on your system and the appropriate type library file included in the project references for your Visual Basic project. The Acrobat type library file is named `Acrobat.tlb`. This file is included in the `InterAppCommunicationSupport\Headers` folder in the SDK. Once you have the type library file included in your project, you can use the object browser to browse the OLE objects. It is not sufficient to install just an ActiveX control or DLL to enable OLE Automation. You must have the full Acrobat product installed to use OLE automation.

Getting Started

This overview and the *Acrobat Interapplication Communication Reference* both document the objects and methods available. These documents (as well as the API) were designed with C programming in mind and programming with the API requires some familiarity with C concepts.

The best resources for a VB or C# .NET programmer, besides the object browser, are the sample projects. These samples demonstrate use of the Acrobat OLE objects and contain comments describing the parameters for the more complicated methods.

If you are a VB .NET programmer, it will be helpful to include the `iac.bas` module in your project (included in the headers folder).

Necessary C Knowledge

Although you do not need the header files provided in the SDK, these files may be used to find the values of various constants, such as **`AV_DOC_VIEW`**, that are referenced in the documentation. The file `iac.h` contains most of these values.

Some of the methods, such as **`OpenInWindowEx`**, can be confusing when utilized in VB .NET. **`OpenInWindowEx`** takes a **`long`** for the **`openflags`** parameter. The options for this parameter provided in the *Acrobat Interapplication Communication Reference* are:

- **`AV_EXTERNAL_VIEW`** — Open the document with the toolbar visible.
- **`AV_DOC_VIEW`** — Draw the page pane and scrollbars.
- **`AV_PAGE_VIEW`** — Draw only the page pane.

If you were developing in the C, these strings would be replaced by a numeric value prior to compilation; passing these strings to the method would not raise an error. When programming in VB .NET, these strings correspond to constant variables defined in `iac.bas`.

In some situations, you will need to apply a bitwise **OR** to multiple values and pass the resultant value to a method. An example of this is using **PDDocSave**. The **ntype** parameter of the **PDDocSave** method is a bitwise **OR** of the following flags as defined in `iac.h`:

```
/* PDSaveFlags — used for PD-level Save
** All undefined flags should be set to zero.
** If either PDSaveCollectGarbage or PDSaveCopy are used, PDSaveFull
must be used. */
typedef enum {
     PDSaveIncremental = 0x0000,  /* write changes only */
     PDSaveFull = 0x0001,         /* write entire file */
     PDSaveCopy = 0x0002,         /* write copy w/o affecting current
                                     state */
     PDSaveLinearized = 0x0004,   /* writes the file linearized */
     PDSaveCollectGarbage = 0x0020  /* perform garbage collection on
                                       unreferenced objects */
} PDSaveFlags;
```

For example, if you would like to fully save the PDF file and linearize it within a VB .NET application, pass **PDSaveFull + PDSaveLinearized** (both defined in `iac.bas`) into the **ntype** parameter; this is the equivalent of a binary **OR** of the **PDSaveFull** and **PDSaveLinearized** parameters. In many instances, the numeric values are spelled out in comments in the VB .NET sample code; however, knowledge of why the methods are structured in this way and how they are utilized in C can be useful to VB and C# .NET programmers.

Understanding the Acrobat OLE Interfaces

OLE 2.0 support includes several classes whose names begin with "**CAcro**" (such as **CAcroApp** and **CAcroPDDoc**) to simplify driving Acrobat. Several files in the SDK encapsulate the definitions of these classes.

The **CAcro** classes are defined in the Acrobat type library `acrobat.tlb`. The **OLEView** tool in Visual Studio allows you to browse registered type libraries. Use `acrobat.tlb` when defining OLE automation for a project in Microsoft Visual C++. The files `acrobat.h` and `acrobat.cpp` are included in the Acrobat SDK, and implement a type-safe wrapper to the Acrobat automation server.

NOTE: Do not modify the `acrobat.tlb`, `acrobat.h`, and `acrobat.cpp` files in the SDK; these define Acrobat's OLE automation interface.

The **CAcro** classes inherit from the MFC **COleDispatchDriver** class. Understanding this class makes it easier to write applications that use the **CAcro** classes and their methods.

The **COleDispatchDriver** class implements the client side of OLE automation, providing most of the code needed to access automation objects. It provides several wrapper functions: **AttachDispatch**, **DetachDispatch**, and **ReleaseDispatch,** as well as several convenience functions: **InvokeHelper**, **SetProperty**, and **GetProperty**. You employ some of these methods when you use the Acrobat-provided automation objects. Other methods are used in Acrobat's implementation of these objects.

The next section discusses how to use the classes exported by `acrobat.cpp`, and shows when to call the **CreateDispatch** and **AttachDispatch** methods.

See the *Acrobat Interapplication Communication Reference* for details on the **CAcro** classes and their methods.

Using the COleDispatchDriver Class With Acrobat

COleDispatchDriver is essentially a "class wrapper" for **IDispatch**, which is the OLE interface by which applications expose methods and properties so that other applications written in Visual Basic and C# .NET can use the application's features. This provides OLE support for Acrobat applications.

Example 3.4 is a section of code from `acrobat.h` that declares the **CAcroHiliteList** class. **CAcroHiliteList** is a subclass of the **COleDispatchDriver** class, which means that it shares all the instance variables of **COleDispatchDriver**. One of these variables is **m_lpDispatch**, which holds an **LPDISPATCH** for that object. An **LPDISPATCH** is a **long** pointer to an **IDispatch**, which can be considered an opaque data type representing a dispatch connection. **m_lpDispatch** can be used in functions that require an **LPDISPATCH** argument.

EXAMPLE 3.4 CAcroHiliteList Class Declaration

```
class CAcroHiliteList : public COleDispatchDriver
{
public:
     CAcroHiliteList() {}// Calls COleDispatchDriver default constructor
     CAcroHiliteList(LPDISPATCH pDispatch) :
COleDispatchDriver(pDispatch) {}
     CAcroHiliteList(const CAcroHiliteList& dispatchSrc) :
COleDispatchDriver(dispatchSrc) {}

// Attributes
public:

// Operations
public:
     bool Add(short nOffset, short nLength);
};
```

Here is the related implementation section of the **Add** method from ACROBAT.CPP:

```
bool CAcroHiliteList::Add(short nOffset, short nLength)
{
     bool result;
     static BYTE parms[] =
           VTS_I2 VTS_I2;
     InvokeHelper(0x1, DISPATCH_METHOD, VT_I4, (void*)&result, parms,
           nOffset, nLength);
     return result;
}
```

When the **Add** method is called (such as with this code from Example 3.5),

```
hilite->Add(0, 10);
```

the **InvokeHelper** function gets called. This **COleDispatchDriver** method takes a variable number of arguments. It eventually calls the Acrobat implementation for **CAcroHiliteList** object's **Add** method. This happens across the virtual OLE "wires" and takes care of all the OLE details. The end result is that a page range is added to the **CAcroHiliteList** object.

Example 3.5 is an implementation of a method adapted from the **ActiveView** sample in the SDK:

EXAMPLE 3.5 Using C OleDispatchDriver Class

```
// This code demonstrates how to highlite words with
// either a word or page highlite list
void CActiveViewDoc::OnToolsHilitewords()
{
     CAcroAVPageView pageView;
     CAcroPDPage page;
     CAcroPDTextSelect* textSelect = new CAcroPDTextSelect;
     CAcroHiliteList* hilite = new CAcroHiliteList;
     char buf[255];
     long selectionSize;

     if ((BOOL) GetCurrentPageNum() > PDBeforeFirstPage) {

        // Obtain the AVPageView
        pageView.AttachDispatch(m_pAcroAVDoc->GetAVPageView(),TRUE);

        // Create the Hilite list object
        hilite->CreateDispatch("AcroExch.HiliteList");
        if (hilite) {

     // Add the first 10 words or characters of that page to the hilite list
           hilite->Add(0,10);
           page.AttachDispatch(pageView.GetPage(), TRUE);

           // Create text selection for either page or word hilite list
           textSelect->AttachDispatch(page.CreateWordHilite(hilite->m_lpDispatch));
```

```
            m_pAcroAVDoc->SetTextSelection(textSelect->m_lpDispatch);
            m_pAcroAVDoc->ShowTextSelect();

            // Extract the number of words and the first word of text selection
            selectionSize = textSelect->GetNumText();
            if (selectionSize)
               sprintf (buf, "# of words in text selection: %ld\n1st word in text
selection = '%s'", selectionSize, textSelect->GetText(0));
            else
               sprintf (buf, "Failed to create text selection.");

            AfxMessageBox(buf);
         }
      }

      delete textSelect;
      delete hilite;
}
```

In the above sample, the objects with the prefix **CAcro** are all **CAcro** class objects—and they are also **COleDispatchDriver** objects—since all the Acrobat **CAcro** classes are subclasses of **COleDispatchDriver**.

Instantiating a class is not sufficient to use it. Before you use an object, you must *attach* your object to the appropriate Acrobat object by using one of the **Dispatch** methods of the **COleDispatchDriver** class. These functions also initialize the **m_lpDispatch** instance variable for the object.

This code from Example 3.5 illustrates how to attach an **IDispatch** that already exists:

```
CAcroAVPageView pageView;
// Obtain the AVPageView
pageView.AttachDispatch(m_pAcroAVDoc->GetAVPageView(), TRUE);
```

The **GetAVPageView** method of the **CAcroAVDoc** class returns an **LPDISPATCH**—which is what the **AttachDispatch** method is expecting for its first argument. The **BOOL** passed as the second argument indicates whether or not the **IDispatch** should be released when the object goes out of scope, and is typically **TRUE**. In general, when an **LPDISPATCH** is returned from a method such as **GetAVPageView**, you use **AttachDispatch** to attach it to an object.

The following code from Example 3.5 uses the **CreateDispatch** method:

```
CAcroHiliteList *hilite = new CAcroHiliteList;
hilite->CreateDispatch("AcroExch.HiliteList");
hilite->Add(0, 10);
```

In this case, the **CreateDispatch** method both creates the **IDispatch** object and attaches it to the object. This code works fine; however, the following code would fail:

```
CAcroHiliteList *hilite = new CAcroHiliteList;
hilite->Add(0, 10);
```

This error is analogous to using an uninitialized variable. Until the **IDispatch** object is attached to the **COleDispatchDriver** object, it is not valid.

CreateDispatch takes a string parameter, such as "**AcroExch.HiliteList**", which represents a class. The following code is incorrect:

```
CAcroPDDoc doc = new CAcroPDDoc;
doc.CreateDispatch("AcroExch.Create");
```

This fails because Acrobat won't respond to such a parameter. The parameter should be "**AcroExch.PDDoc**" instead. The following table lists all the valid strings.

TABLE 3.1 Strings For CreateDispatch

CAcroPoint: "AcroExch.Point"
CAcroRect: "AcroExch.Rect"
CAcroTime: "AcroExch.Time"
CAcroApp: "AcroExch.App"
CAcroPDDoc: "AcroExch.PDDoc"
CAcroAVDoc: "AcroExch.AVDoc"
CAcroHiliteList: "AcroExch.HiliteList"
CAcroPDBookmark: "AcroExch.PDBookmark"
CAcroMatrix: "AcroExch.Matrix"
AcroPDF: "AxAcroPDFLib.AxAcroPDF"

Returning to the code in Example 3.5:

```
CAcroPDPage page;
page.AttachDispatch(pageView.GetPage(), TRUE);
```

A **PDPage** object is needed, because the purpose of this code is to highlight words on the current page. Since it is a **CAcro** variable, it is necessary to attach to the OLE object before using its methods. **CreateDispatch** cannot be used create a **PDPage** object (since there is no "**AcroExch.PDPage**" object listed in Table 3.1). However, the **AVPageView** method **GetPage** returns an **LPDISPATCH** pointer for a **PDPage** object. This is passed as the first argument to the **AttachDispatch** method of the page object. (The **TRUE** argument indicates we want the object to be released automatically when it goes out of scope).

```
CAcroPDTextSelect* textSelect = new CAcroPDTextSelect;
textSelect->AttachDispatch
     (page.CreateWordHilite(hilite->m_lpDispatch));
m_pAcroAVDoc->SetTextSelection (textSelect->m_lpDispatch);
m_pAcroAVDoc->ShowTextSelect();
```

This code does the following:

1. Declares a text selection object **`textSelect`**.
2. Calls the **`CAcroPDPage`** method **`CreateWordHilite`**, which returns an **`LPDISPATCH`** for a **`PDTextSelect`**. **`CreateWordHilite`** takes an **`LPDISPATCH`** argument representing a **`CAcroHilite`** list. The **`hilite`** variable already contains a **`CAcroHiliteList`** object,and its instance variable **`m_lpDispatch`** contains the **`LPDISPATCH`** pointer for the object.
3. Calls the **`CAcroAVDoc`** object's **`SetTextSelection`** method to select the first ten words on the current page.
4. Calls the **`AcroAVDoc`**'s **`ShowTextSelect`** method to cause the visual update on the screen.

OLE Automation Using the JSObject Interface

Whenever possible, you should take advantage of the powerful capabilities inherent to Acrobat JavaScript by using the **`JSObject`** interface available within the **`AcroExch.PDDoc`** object. To obtain the interface, invoke the object's **`GetJSObject`** method, as shown below in Example 3.6:

EXAMPLE 3.6 Using the JSObject Interface

```
Dim gApp As Acrobat.CAcroApp
Dim gPDDoc As Acrobat.CAcroPDDoc
Dim jso As Object

Private Sub Form_Load()
    Set gApp = CreateObject("AcroExch.App")
    Set gPDDoc = CreateObject("AcroExch.PDDoc")

    If gPDDoc.Open("c:\adobe.pdf") Then
        Set jso = gPDDoc.GetJSObject
        jso.console.Show
        jso.console.Clear
        jso.console.println ("Hello, Acrobat!")
        gApp.Show
    End If
End Sub
```

Other Useful Information

This section provides other pieces of information and hints useful in developing plug-ins using OLE automation.

Using OLE Messages

The Acrobat OLE automation implementation is based on a synchronous messaging scheme. When an application sends a request to Acrobat, the application processes that request and returns control to the application. Only then can the application send Acrobat another message. If your application sends one message followed immediately by another, the second message may not be properly received (instead of generating a server busy error, it fails with no error message).

For example, this problem manifests itself with the **`AVDoc.OpenInWindowEx`** method, where a large volume of information regarding drawing position and mouse clicks is exchanged, and with the usage of the **`PDPage.DrawEx`** method on especially complex pages. With the **`DrawEx`** method, the problem arises when a **`WM_PAINT`** message is generated. If the page is complex and the environment is multi-threaded, the application may not finish drawing the page before the application generates another **`WM_PAINT`** message. Since the application is single-threaded, multi-thread applications must take care to handle this situation appropriately.

MDI Usage

Suppose you create a multiple document interface application that creates a static window into which Acrobat displays (using the **`OpenInWindowEx`** call), and this window is based on the **`CFormView`** OLE class. If another window is placed on top of that window and is subsequently removed, the Acrobat window does not repaint correctly.

To fix this, assign the **Clip Children** style to the dialog template (on which **`CFormView`** is based). Otherwise, the dialog erases the background of all child windows, including the one containing the PDF file, which wipes out the previously covered part of the PDF window.

Event Handling

When a PDF file is opened with **`OpenInWindowEx`**, Acrobat creates a child window on top of it. This allows the application to receive events for this window directly. However, an application must also handle the following events: **`resize`**, **`key up`**, and **`key down`**.

The following example from the `ActiveView` sample shows how to handle a resize event:

EXAMPLE 3.7 Handling Resize Events

```
void CActiveViewVw::OnSize(UINT nType, int cx, int cy)
{
     CWnd* pWndChild = GetWindow(GW_CHILD);
     if (!pWndChild)
          return;
     CRect rect;
     GetClientRect(&rect);
     pWndChild->
     SetWindowPos(NULL,0,0,rect.Width,rect.Height,
     SWP_NOZORDER | SWP_NOMOVE);

     CView::OnSize(nType, cx, cy);
}
```

After sending the message to the child window, it also does a resize. This results in both windows being resized, which is the desired effect.

OLE Automation Summary

This section simply lists all the OLE automation methods. For complete descriptions of the parameters associated with OLE automation methods, see the OLE automation sections of the *Acrobat Interapplication Communication Reference*.

Objects

The Acrobat application is represented as several OLE automation objects:

- **`AcroExch.App`** — The application itself.
- **`AcroExch.AVDoc`** — A document as seen in the user interface.
- **`AcroExch.PDDoc`** — The underlying PDF representation of a document. The first page in a **`PDDoc`** is page 0.
- **`AcroExch.Hilite`** — An entry in a highlight list. A highlight list is used to highlight one or more groups of characters/words on a single page.
- **`AcroExch.AVPageView`** — The area of Acrobat's window that displays the contents of a document's page.
- **`AcroExch.PDPage`** — A single page in the PDF representation of a document. The first page in a **`PDDoc`** is page 0.
- **`AcroExch.PDAnnot`** — An annotation on a page in the PDF file.
- **`AcroExch.PDBookmark`** — A bookmark in a PDF file.
- **`AcroExch.PDTextSelect`** — A selection of text on a single page that may contain more than one disjointed group of words.
- **`AxAcroPDFLib.AxAcroPDF`** — An object containing PDF browser controls. This object provides simple methods making it possible to load a file, move to various pages within a file, and specify various display and print options.

Data Types

Acrobat supports the following data types for OLE automation:

- **`AcroExch.HiliteList`** — A list of highlighted characters, which may include one or more contiguous groups of characters or words on a single page. The **`Add`** function is provided to add to a **`HiliteList`**.
- **`AcroExch.Point`** — A point, specified by its *x*– and *y*–coordinates.
- **`AcroExch.Rect`** — A rectangle, specified by the top left and bottom right points.
- **`AcroExch.Time`** — A specified time, accurate to the millisecond, including the day of the week.

Methods

Acrobat OLE automation support includes the following methods, grouped by object.

AcroExch.App

- **CloseAllDocs** — Closes all open documents.
- **Exit** — Exits Acrobat.
- **GetActiveDoc** — Gets the frontmost document.
- **GetActiveTool** — Gets the name of the currently active tool.
- **GetAVDoc** — Gets an **AVDoc** via its index within the list of open **AVDoc** objects.
- **GetFrame** — Gets the window's frame.
- **GetInterface** — Gets an IDispatch interface for a named object such as a third-party plug-in.
- **GetLanguage** — Gets a code that specifies which language Acrobat's user interface is using.
- **GetNumAVDocs** — Gets the number of open **AVDocs**.
- **GetPreferenceEx** — Gets a value in the preferences file for zoom values and colors.
- **Hide** — Hides Acrobat.
- **Lock** — Locks Acrobat.
- **Maximize** — Maximizes Acrobat.
- **MenuItemExecute** — Executes the menu item whose language-independent menu item name is specified.
- **MenuItemIsEnabled** — Determines whether the specified menu item is enabled.
- **MenuItemIsMarked** — Determines whether the specified menu item is marked.
- **MenuItemRemove** — Removes the menu item whose language-independent menu item is specified.
- **Minimize** — Minimizes Acrobat.
- **Restore** — Restores the Acrobat window to its previous state.
- **SetActiveTool** — Sets the active tool according to the specified name.
- **SetFrame** — Sets the window's frame to the specified rectangle.
- **SetPreferenceEx** — Sets a value in the preferences file for zoom values and colors.
- **Show** — Shows Acrobat.
- **ToolButtonIsEnabled** — Determines whether the specified toolbar button is enabled.
- **ToolButtonRemove** — Removes the specified button from the toolbar.
- **UnlockEx** — Unlocks Acrobat if it was previously locked.

AcroExch.AVDoc

- **`BringToFront`** — Brings the window to the front.
- **`ClearSelection`** — Clears the current selection.
- **`Close`** — Closes a document.
- **`FindText`** — Finds the specified text, scrolls so that it is visible, and highlights it.
- **`GetAVPageView`** — Gets the **`AVPageView`** associated with an **`AVDoc`**.
- **`GetFrame`** — Gets the rectangle specifying the window's size and location.
- **`GetPDDoc`** — Gets the **`PDDoc`** associated with an **`AVDoc`**.
- **`GetTitle`** — Gets the window's title.
- **`GetViewMode`** — Gets the current document view mode (pages only, pages and thumbnails, or pages and bookmarks).
- **`IsValid`** — Determines whether the **`AVDoc`** is still valid.
- **`Maximize`** — Maximizes the window if **`MaxSize`** is **`true`**.
- **`Open`** — Opens a file.
- **`OpenInWindowEx`** — Opens a PDF file and displays it in the specified window, according to the specified page number, zoom factor, and page view.
- **`PrintPages`** — Prints a specified range of pages, displaying a print dialog box.
- **`PrintPagesSilent`** — Prints a specified range of pages without displaying any print dialog box.
- **`SetFrame`** — Sets the window's size and location.
- **`SetTextSelection`** — Sets the document's selection to the specified, previously created text selection.
- **`SetTitle`** — Sets the window's title.
- **`SetViewMode`** — Sets the mode in which the document is viewed (pages only, pages and thumbnails, or pages and bookmarks).
- **`ShowTextSelect`** — Changes the view so that the current text selection is visible.

AcroExch.AVPageView

- **DevicePointToPage** — Converts the coordinates of a point from device space to user space.
- **DoGoBack** — Goes to the previous view on the view history stack, if any.
- **DoGoForward** — Goes to the next view on the view history stack, if any.
- **GetAVDoc** — Gets the **AVDoc** corresponding to the current page.
- **GetAperture** — Gets the aperture associated with the current page. The *aperture* is the rectangular region of the window in which the document is drawn, measured in device space units.
- **GetDoc** — Gets the **PDDoc** object corresponding to the current page.
- **GetPage** — Gets the **Page** objectcorresponding to the current page.
- **GetPageNum** — Gets the page number of the page. The first page in a document is page zero.
- **GetZoom** — Gets the current zoom factor, specified as a percent (for example, 100 is returned if the magnification is 1.0).
- **GetZoomType** — Gets the current zoom type.
- **Goto** — Goes to the specified page.
- **ReadPageDown** — Scrolls forward through the document by one screen area.
- **ReadPageUp** — Scrolls backward through the document by one screen area.
- **ScrollTo** — Scrolls to the specified location on the current page.
- **ZoomTo** — Zooms to the specified magnification.

AcroExch.HiliteList

- **Add** — Adds the specified highlight to the current highlight list.

AcroExch.PDAnnot

- **`GetColor`** — Gets an annotation's color.
- **`GetContents`** — Gets a text annotation's contents.
- **`GetDate`** — Gets an annotation's date.
- **`GetRect`** — Gets an annotation's bounding rectangle.
- **`GetSubtype`** — Gets an annotation's subtype.
- **`GetTitle`** — Gets a text annotation's title.
- **`IsEqual`** — Determines whether or not an annotation is the same as the specified annotation.
- **`IsOpen`** — Tests whether a text annotation is open.
- **`IsValid`** — Tests whether an annotation is still valid.
- **`Perform`** — Performs a link annotation's action.
- **`SetColor`** — Sets an annotation's color.
- **`SetContents`** — Sets a text annotation's contents.
- **`SetDate`** — Sets an annotation's date.
- **`SetOpen`** — Opens or closes a text annotation.
- **`SetRect`** — Sets an annotation's bounding rectangle.
- **`SetTitle`** — Sets a text annotation's title.

AcroExch.PDBookmark

NOTE: It is not possible to create a bookmark with OLE—only to destroy one.

- **`Destroy`** — Destroys a bookmark. It is not possible to *create* a bookmark with OLE.
- **`GetByTitle`** — Gets the bookmark that has the specified title.
- **`GetTitle`** — Gets a bookmark's title (up to 256 characters).
- **`IsValid`** — Determines whether the bookmark is still valid.
- **`Perform`** — Performs a bookmark's action.
- **`SetTitle`** — Sets a bookmark's title.

AcroExch.PDDoc

- **AcquirePage** — Acquires the specified page.
- **ClearFlags** — Clears a document's flags. The flags indicate:
  - whether the document has been modified.
  - whether the document is a temporary document and should be deleted when closed.
  - the version of PDF used in the file. This method can be used only to clear, not to set, flag bits.
- **Close** — Closes a file.
- **Create** — Creates a new **PDDoc**.
- **CreateTextSelect** — Creates a text selection from the specified rectangle on the specified page.
- **CreateThumbs** — Creates thumbnail images for the specified page range in a document.
- **CropPages** — Crops the pages in a specified page range.
- **DeletePages** — Deletes pages from a file.
- **DeleteThumbs** — Deletes thumbnail images from the specified pages in a document.
- **GetFileName** — Gets the name of the file associated with this **PDDoc**.
- **GetFlags** — Gets a document's flags. The flags indicate:
  - whether the document has been modified.
  - whether the document is a temporary document and should be deleted when closed.
  - the version of PDF used in the file.
- **GetInfo** — Gets the value of a specified key in the document's info dictionary.
- **GetInstanceID** — Gets the instance ID from the ID array in the document's trailer.
- **GetJSObject** — Gets a dual interface to the JavaScript object associated with the **PDDoc**, allowing automation clients full access to the built-in and user-defined Acrobat JavaScript methods available in the document.
- **GetNumPages** — Gets the number of pages in a file.
- **GetPageMode** — Gets a value indicating whether Acrobat is currently displaying only pages, pages and thumbnails, or pages and bookmarks.
- **GetPermanentID** — Gets the permanent ID from the ID array in the document's trailer.
- **InsertPages** — Inserts the specified pages from a source document after the indicated page within the current document.
- **MovePage** — Moves a page to another location within the same document.
- **Open** — Opens a file.
- **OpenAVDoc** — Opens a window and displays the document in it.

- **`ReplacePages`** — Replaces the indicated pages in the current document with those specified from the source document.
- **`Save`** — Saves a document.
- **`SetFlags`** — Sets a document's flags. The flags indicate:
  - whether the document has been modified.
  - whether the document is a temporary document and should be deleted when closed.
  - the version of PDF used in the file. This method can be used only to set, not clear, flag bits.
- **`SetInfo`** — Sets the value of a key in a document's info dictionary.
- **`SetOpenInfo`** — Sets the parameters that tell how a document is opened.
- **`SetPageMode`** — Sets the page mode in which a document is opened: display only pages, pages and thumbnails, or pages and bookmarks.

AcroExch.PDPage

- **`AddAnnot`** — Adds a specified annotation at a specified location in the page's annotation array.
- **`AddNewAnnot`** — Creates a new text annotation and adds it to the page.
- **`CopyToClipboard`** — Copies a PDF image to the clipboard without requiring an **`hWnd`** or **`hDC`**.
- **`CreatePageHilite`** — Creates a text selection from a list of character offsets and character counts on a single page.
- **`CreateWordHilite`** — Creates a text selection from a list of word offsets and word counts on a single page.
- **`CropPage`** — Crops the current page.
- **`DrawEx`** — Instructs Acrobat to draw into a specified window, specifying a page view and zoom factor.
- **`GetAnnot`** — Gets the specified annotation on the page.
- **`GetAnnotIndex`** — Gets the index (in the page's annotation array) of the specified annotation.
- **`GetDoc`** — Gets the **`PDDoc`** associated with the page.
- **`GetNumAnnots`** — Gets the number of annotations on the page.
- **`GetNumber`** — Gets the page number of the current page. The first page in a document is page zero.
- **`GetRotate`** — Gets the rotation value for the current page.
- **`GetSize`** — Gets a page's width and height in points.
- **`RemoveAnnot`** — Removes the specified annotation from the page's annotation array.
- **`SetRotate`** — Sets the rotation for the current page.

AcroExch.PDTextSelect

- **`Destroy`** — Destroys a text selection.
- **`GetBoundingRect`** — Gets a text selection's bounding rectangle.
- **`GetNumText`** — Gets the number of text elements in a text selection.
- **`GetPage`** — Gets the page number on which a text selection is located.
- **`GetText`** — Gets the text from the specified element of a text selection.

AxAcroPDFLib.AxAcroPDF

- **`GoBackwardStack`** — Goes to the previous view on the view stack, if it exists.
- **`GoForwardStack`** — Goes to the next view on the view stack, if it exists.
- **`GotoFirstPage`** — Goes to the first page in the document.
- **`GotoLastPage`** — Goes to the last page in the document.
- **`GotoNextPage`** — Goes to the page in the document, if it exists.
- **`GotoPreviousPage`** — Goes to the previous page in the document, if it exists.
- **`LoadFile`** — Opens and displays the specified document within the browser.
- **`Print`** — Prints the document according to the specified options in a user dialog box.
- **`PrintAll`** — Prints the entire document without a user dialog box.
- **`PrintAllFit`** — Prints the entire document without a user dialog box, and shrinks pages as needed to fit the imageable area of a page in the printer.
- **`PrintPages`** — Prints the specified pages without displaying a user dialog box.
- **`PrintPagesFit`** — Prints the specified pages without displaying a user dialog box, and shrinks pages as needed to fit the imageable area of a page in the printer.
- **`PrintWithDialog`** — Prints the document according to the specified options in a user dialog box. These options may include embedded printing and specifying which printer is to be used.
- **`SetCurrentHighlight`**— Highlights the text selection within the specified bounding rectangle on the current page.
- **`SetCurrentPage`** — Goes to the specified page within the document.
- **`SetLayoutMode`** — Sets the layout mode for the page view.
- **`SetNamedDest`** — Changes the page view to the specified named destination.
- **`SetPageMode`** — Sets the page mode to display the document only, or to additionally display bookmarks or thumbnails.
- **`SetShowScrollbars`** — Determines whether scrollbars will appear in the document view.
- **`SetShowToolbar`** — Determines whether a toolbar will appear in the application.
- **`SetView`** — Determines how the page will fit in the current view.

- **`SetViewRect`** — Sets the view rectangle according to the specified coordinates.
- **`SetViewScroll`** — Determines how the page will fit in the current view, and scrolls the page either to the right or down by the specified amount.
- **`SetZoom`** — Sets the magnification according to the specified value, expressed as a percent.
- **`SetZoomScroll`** — Sets the magnification according to the specified value, and scrolls the page either to the right or down by the specified amount.

4 DDE Support

This chapter describes DDE support in Acrobat under Microsoft Windows.

IMPORTANT: *You should use OLE automation instead of DDE whenever possible, since DDE is not a COM technology.*

This chapter lists all DDE messages. For complete descriptions of the parameters associated with DDE messages, see the DDE sections of the *Acrobat Interapplication Communication Reference*.

Differences Among the Acrobat Applications

Acrobat 4.0 and later supports 32-bit applications.

Acrobat supports all of the DDE messages listed in Acrobat Application DDE Messages.

Adobe Reader supports only the following DDE messages: **`AppExit`**, **`CloseAllDocs`**, **`DocClose`**, **`DocGoTo`**, **`DocGoToNameDest`**, **`DocOpen`**, **`FileOpen`**, **`FilePrint`**, **`FilePrintEx`**, **`FilePrintSilent`**, and **`FilePrintTo`**.

General Information

For all DDE messages listed in this chapter, the service name is **`acroview`**, the transaction type is **`XTYPE_EXECUTE`**, and the topic name is **`control`**. The data is the command to be executed, enclosed within square brackets. The item argument in the **`DdeClientTransaction`** call is **`NULL`**.

Example 1 sets up a DDE message:

EXAMPLE 1 Set Up a DDE Message

```
DDE_SERVERNAME = "acroview";
DDE_TOPICNAME = "control";
DDE_ITEMNAME = "[AppHide()]";
```

NOTE: The square bracket characters in DDE messages are mandatory.

NOTE: DDE messages are case-sensitive and must be used exactly as described.

You must first open a document using the **`DocOpen`** DDE message in order to be able to use other DDE messages on it. You cannot use DDE messages to close a document that a user opened manually.

You can use **`NULL`** for pathnames, in which case the DDE message operates on the front document.

If more than one command is sent at once, they are executed sequentially, and the results appear to the user as a single action. For instance, you can utilize this feature to open a document to a certain page and zoom level.

Page numbers are zero-based: the first page in a document is page 0.

Quotation marks are needed only if a parameter contains white space.

The document manipulation methods, such as those for deleting pages or scrolling, only work on documents that are already open.

Acrobat Application DDE Messages

This section lists all DDE messages. For complete descriptions of the parameters associated with DDE messages, see the DDE sections of the *Acrobat Interapplication Communication Reference*.

Application Configuration

- **`AppExit`** — Exits Acrobat.
- **`AppHide`** — Iconifies or hides Acrobat.
- **`AppShow`** — Shows Acrobat.
- **`CloseAllDocs`** — Closes all open documents.
- **`HideToolbar`** — Hides the toolbar.
- **`MenuitemExecute`** — Invokes a menu item, given its language-independent name.
- **`ShowToolbar`** — Shows the toolbar.

Document Manipulation

- **`DocClose`** — Closes the file without saving it and without prompting the user to save the document if it has been modified.
- **`DocDeletePages`** — Deletes a specified range of pages in a document. It cannot delete all pages in a document.
- **`DocInsertPages`** — Inserts specified pages from one file into another.
- **`DocOpen`** — Opens a document and adds it to the list of documents known to DDE, allowing it to be manipulated by other DDE messages (for example, **`FileOpen`**).
- **`DocReplacePages`** — Replaces specified pages using pages from another file.
- **`DocSave`** — Saves the specified file.
- **`DocSaveAs`** — Saves an open file into a new file, without warning the user if there is a problem saving.
- **`DocSetViewMode`** — Controls whether bookmarks or thumbnail images are shown in addition to the document content.
- **`FileOpen`** — Opens and displays a file, making it the current document and bringing it to the front if it is already open.
- **`FileOpenEx`** — Opens and displays a file, making it the current document and bringing it to the front if it is already open. The file is opened during an idle loop to allow DDE messages to continue flowing during the opening of large documents.

Document Printing

- **`DocPrint`** — Prints a specified range of pages from a document, without displaying a modal Print dialog box to the user.
- **`FilePrint`** — Prints all pages in a document, displaying a modal Print dialog box to the user.
- **`FilePrintEx`** — Prints all pages in a document, displaying a modal Print dialog box to the user. Only PostScript Level 1 operators are used for PostScript printing. Printing is performed during an idle loop to allow DDE messages to continue flowing during the printing of large documents.
- **`FilePrintSilent`** — Prints all pages in a document, displaying no print dialog box to the user.
- **`FilePrintSilentEx`** — Prints all pages in a document, displaying no print dialog box to the user. Only PostScript Level 1 operators are used for PostScript printing. Printing is performed during an idle loop to allow DDE messages to continue flowing during the printing of large documents.
- **`FilePrintTo`** — Prints all pages in a document to a specified printer, using a specified driver and port, displaying a modal Print dialog box to the user.
- **`FilePrintToEx`** — Prints all pages in a document to a specified printer, using a specified driver and port, displaying a modal Print dialog box to the user. Only PostScript Level 1 operators are used for PostScript printing. Printing is performed during an idle loop to allow DDE messages to continue flowing during the printing of large documents.

View Manipulation

- **`DocGoTo`** — Goes to the specified page.
- **`DocGoToNameDest`** — Goes to the specified name destination within the document.
- **`DocPageDown`** — Scrolls forward through the document by one screen area.
- **`DocPageLeft`** — Scrolls to the left by a small amount.
- **`DocPageRight`** — Scrolls to the right by a small amount.
- **`DocPageUp`** — Scrolls backward through the document by one screen area.
- **`DocScrollTo`** — Scrolls the view of the current page to a specified location.
- **`DocZoomTo`** — Sets the zoom for a specified document.

Search-related

- **`DocFind`** — Finds a string in a specified file.

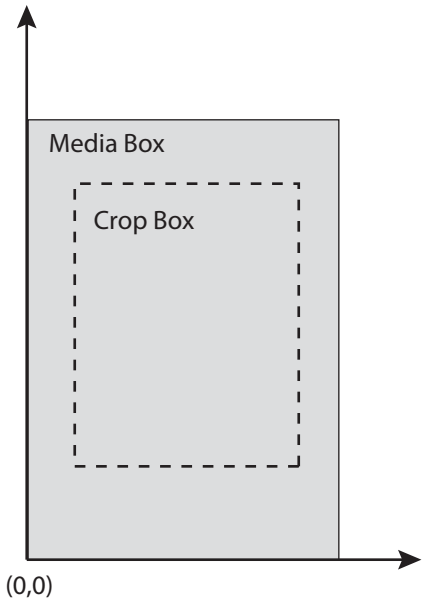
A IAC Coordinate Systems

The Acrobat application's IAC support uses two coordinate systems: user space and device space. This appendix describes these coordinate systems.

User Space

User space is the coordinate system used within PDF files. In the IAC interface, it is used for most PD layer objects (that is, objects such as `PDBookmark` whose names begin with "PD"). Figure A.1 shows the user space coordinate system. The orientation, origin, and scale of the user space coordinate system can be changed by operators in the page description in a PDF file.

FIGURE A.1 User Space Coordinate System

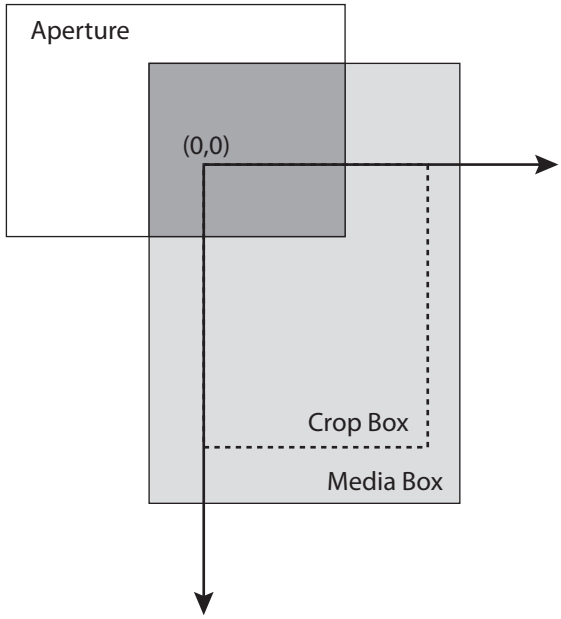


Default user space is the user space coordinate system in effect immediately before each page begins drawing. The origin of this coordinate system is the lower left corner of a page's media box. The *x*-coordinate increases to the right, and the *y*-coordinate increases upward. One unit in default user space is 1/72 of an inch.

Device Space

Device space specifies coordinates in screen pixels, as shown in Figure A.2. It is used in the AVModel portion of the IAC interface (that is, objects such as **AVDoc** whose names begin with "AV").

FIGURE A.2 Device Space Coordinate System



The origin of the device space coordinate system is at the upper left corner of the visible page on the screen (that is, the upper left corner of the white part of the page). The *x*-coordinate increases to the right, and the *y*-coordinate increases downward.

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

B Visual Studio .NET Migration

Introduction

In general, Acrobat 6 and earlier plug-ins require no code modification to make them compatible with Acrobat 7 running on any Windows platform.

However, the Acrobat 7 SDK is only supported within Visual Studio .NET 2003, so you must upgrade your plug-ins from earlier versions of Visual Studio.

Upgrading Plug-ins to Visual Studio .NET 2003

Use Visual Studio .NET 2003 to automatically convert your project from previous versions of Visual Studio. Once this is accomplished, you must also make the following changes to your plug-in's project:

- Update the relative path of the header files in the file `AcroDSPOptions.rsp`. Note that the **Headers** folder now contains three sub-folders: **ADM**, **API**, and **SDK**.
- Update the paths of all files that are located in the header files folders and are source files for your project. Additionally, for each plug-in you must also update the path for `PIMain.c`. For plug-ins that use **ADM**, update the paths for `ADMAcroSDK.cpp` and `ADMAcroSDK.h`. To update these files, delete the source file from your project and then add it back in using the correct path.
- For each plug-in, turn on the **GS** switch. This is accomplished through the usage of **Project->Properties->C/C++/->Code Generation**: set the **Buffer Security Check** flag to **Yes (/GS)**.
- Enable incremental linking for each plug-in. This is accomplished through the usage of **Project->Properties->Linker->General**: set the **Enable Incremental Linking** flag to **Yes**.

Once you have upgraded your project, recompile your plug-in using the headers provided in the Acrobat 7 SDK.

Be sure that the **AcroSDKPIDir** environment variable is set correctly. This is accomplished through the usage of **My Computer->Control Panel->System->Advanced**: set the environment variable to the desired location for your plug-ins.