Adobe Developers Association

ADA news

In This Issue

Volume 3, Number 9

The Acrobat Exchange Plug-In API

Version 2.0 of the Adobe™ Acrobat™ products are here! Versions for Microsoft® Windows™ and Apple® Macintosh® computers recently shipped. The new versions contain a number of enhancements, including a variety of ways for developers to control and enhance the Acrobat products. Acrobat Exchange, an application that provides a way to view, navigate, and print PDF files, now has extremely rich application programming interfaces (APIs). Developers can create plug-ins and can also use the extensive interapplication communication (IAC) support provided, as shown in Figure 1.

Acrobat Exchange plug-ins are very similar to those used with Adobe Photoshop™ or Adobe Illustrator.™ They are pieces of standalone code that are loaded by Acrobat Exchange each time it launches, and are implemented as code resources on the Macintosh and as DLLs under Windows. Users need only drag them into a specific directory in order for Acrobat Exchange to find and load them.

In this article, we'll describe the Acrobat Exchange plug-in API and give you some ideas of its power. Information on other ways to integrate with the Acrobat products will appear in future issues of this newsletter.

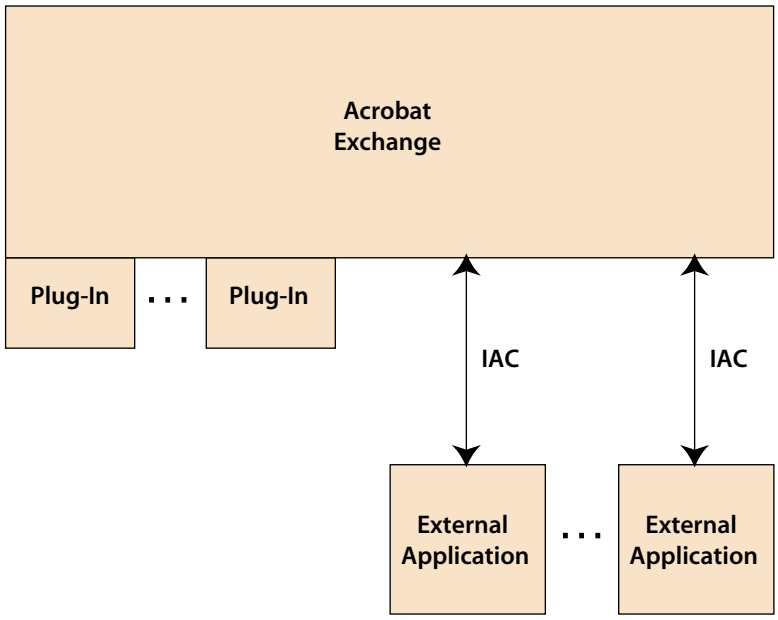

Figure 1. Acrobat Exchange integration

Adobe

How To Reach Us

DEVELOPERS ASSOCIATION HOTLINE:

U.S. and Canada:
(415) 961-4111
M–F, 8 a.m.–5 p.m., PDT.
If all engineers are unavailable, please leave a detailed message with your developer number, name, and telephone number, and we will get back to you within 24 hours.

Europe:
+31-20-6511-355

FAX:

U.S. and Canada:
(415) 967-9231
Attention:
Adobe Developers Association

Europe:
+31-20-6511-313
Attention:
Adobe Developers Association

EMAIL:

U.S.
devsup-person@mv.us.adobe.com

Europe:
eurosupport@adobe.com

MAIL:

U.S. and Canada:
Adobe Developers Association
Adobe Systems Incorporated
1585 Charleston Road
P.O. Box 7900
Mt. View, CA 94039-7900

Europe:
Adobe Developers Association
Adobe Systems Europe B.V.
Europlaza
Hoogoorddreef 54a
1101 BE Amsterdam Z.O.
The Netherlands

Send all inquiries, letters and address changes to the appropriate address above.

The Acrobat Exchange Plug-In API

Plug-in examples

Here are just a few of the many things you can do using the plug-in API. You can:

• Request Acrobat Exchange to render PDF files into a window you provide, allowing you to add PDF viewing to your application.

• Extract text and other information from a PDF file, allowing you to index the text in a PDF file or to convert a PDF file into another format.

• Highlight arbitrary sequences of text on a page. This can be used in conjunction with text extraction to create a cross-document search system for PDF files. The cross-document search capability that ships with Acrobat Exchange is implemented as a plug-in.

• Add menus, menu items, and toolbar buttons that execute your code. This might be something as simple as adding a toolbar button that always opens a pre-determined file to a particular page, or something as complex as a toolbar button that extracts a document from a document management system.

• Add custom annotation and action types to Acrobat Exchange. *Annotations* contain information that is associated with a page, but is not part of the page content itself. Acrobat Exchange ships with two types of annotations: notes and links. You can add a new type of annotation; for example, one that allows you to make arbitrary editing marks on a page. *Actions* are what happens when you click on a link or a bookmark. Acrobat Exchange ships with two built-in types of actions: "goto view" actions that jump to another location in the same document, and "open file" opens another file. A custom action could, for example, play a sound whenever a user clicks on a particular link.

• Read, modify, and add arbitrary data to a PDF file. You can use this to add your own private data into a PDF file, programmatically manipulate objects such as annotations, or determine the objects present in a PDF file.

• Replace the low-level file access mechanisms Exchange uses. You can use this to enhance Acrobat Exchange so that it can retrieve individual pages of a PDF file via a modem or other remote connection instead of from a local hard disk. You could also add the ability to retrieve PDF files from a database.

Plug-in API organization

The API is organized into objects, although it is implemented using a plain C interface. The API consists of approximately 50 object types organized into four groups. Object types represent items in Acrobat Exchange such as menus and toolbar buttons, items in PDF files such as fonts and annotations, and filesystem items such as files. Each object type has one or more methods that allow you to create, destroy, examine, and modify objects of that type. The API contains more than 500 methods.

Don't Forget to register!

If you have purchased a Software Development Kit (SDK), be sure to return the registration card. We will be incorporating new features into the SDK's in the future, and want to be sure to inform you about new additions.

The Acrobat Exchange Plug-In API

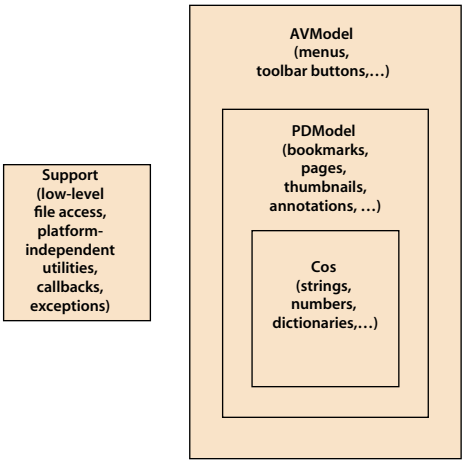

Figure 2. *API organization*

Figure 2 shows the overall organization of the API. The following paragraphs briefly describe each of the API's four method groups.

The AVModel group is the outermost group. It contains objects that represent parts of the Acrobat Exchange application, including menus, menu items, document windows, toolbar buttons, and the application itself. Some of the methods in this group allow you to open a window containing a PDF document, add or remove menus and menu items, add or remove toolbar buttons, and modify settings in the application's preferences file.

The PDModel group contains objects that represent components in a PDF file, including pages, fonts, annotations, actions, and bookmarks. The structure of a PDF file is described in the *Portable Document Format Reference Manual.* You use PDModel methods to extract text from a PDF file, to add annotations or actions, determine the fonts a document uses, etc. The PDModel methods, unlike the Cos methods described in the following paragraph, ensure that any changes you make to a file result in a valid PDF file.

The Cos group contains objects that represent the low-level data in a PDF file (Cos is a recursive acronym for Cos object system). Cos contains the seven data types described in Chapter 4 of the *Portable Document Format Reference Manual:* booleans, numbers, strings, names, arrays, dictionaries, and streams. You use Cos methods to add private data to a PDF file, or to read or modify parts of a PDF file for which PDModel methods are not provided. Because Cos is more general than PDF, it is possible to create an invalid PDF file using the Cos methods; use Cos methods only when the appropriate methods do not exist in the other API groups.

The Acrobat Support method group provides a collection of utility and file access methods. Included are utilities (fixed point math, memory allocation), platform-independent pathname methods, and methods that allow a plug-in to export its own callable functions to other plug-ins. Using methods in this group, you can convert pathname specifications between platform-independent and native formats, or replace the filesystem to allow files to be read via modem or from a database.

continued on page 4

The Acrobat Exchange Plug-In API

Plug-in operation

The operation of plug-ins has three phases: handshaking and initialization, operation, and unloading.

As mentioned previously, Acrobat Exchange looks in a special folder for plug-ins when it launches. It handshakes with each plug-in it locates, obtaining the plug-in's name and the addresses of its initialization and unload procedures. Initialization is carried out in two steps. The first occurs before all other plug-ins are loaded; as a result, the plug-in cannot make any decisions based on knowing what other plug-ins are present. The second is carried out after all plug-ins are loaded. Plug-ins generally use initialization to add user interface items, define custom actions or custom annotations, or replace portions of the filesystem. For each of these, the plug-in provides pointers to one or more functions that are automatically called by Acrobat Exchange. For example, when you add a menu item you provide a function that is called when the menu item is selected by the user. Annotations are more complicated; you must provide procedures that create, destroy, draw, highlight and un-highlight your annotation type, as well as procedures that control the cursor shape and handle mouse clicks in your annotation type.

After initialization, a plug-in generally waits for the procedures it registered during initialization to be called. For example, a procedure you specify is automatically called by Acrobat Exchange when a user clicks within a custom annotation your plug-in handles, and a different procedure you specify is automatically called when a user clicks on a menu item you've added. Acrobat Exchange also lets you register procedures that are automatically called for a variety of *notifications*, which occur when certain user actions such as zooming, changing the page, and opening or closing a file, occur.

Unloading currently occurs only when Acrobat Exchange quits. Your unload procedure can clean up, close any files it might have opened, and free any memory it allocated.

Conclusion

The APIs in version 2.0 of the Acrobat products provide innumerable opportunities to enhance and control the Acrobat products. The Acrobat Exchange plug-in API is a rich environment for you to develop exciting products that enhance Acrobat Exchange or help your product work more closely with it. Detailed technical information on the plug-in API and the other ways to integrate with version 2.0 of the Acrobat products is available in an Acrobat Software Development Kit. Contact the ADA for further information on SDK pricing and availability. Also, watch future issues of this newsletter for articles describing some of the other ways you can integrate with the Acrobat products. §

Questions Answers

Q I would like to compress my PostScript™ language program prior to sending it to a level 2 printer, in order to save transmission time. I've LZW compressed the program using the sample code in the PostScript Language SDK. Now what should I do with the compressed code?

A LZW (Lempel-Ziv-Welsh) compression was a good choice for what you are doing. Of all the compression methods that we support in Level 2, LZW usually yields the highest amount of compression for generic PostScript language code. Sending the compressed code down to the printer is straightforward. Just put the following lines around your compressed code:

```
%!PS-Adobe 3.0
%%Title: (Example using lzw decompression)
%%EndComments
%%Page: 1 1
currentfile /LZWDecode filter cvx
%%BeginData: 100 Binary Bytes
exec ...total 95 bytes of lzw compressed code here...
%%EndData
%%EOF
```

Note that LZW compression produces binary data, and whenever you put binary data into a job stream it is important to separate the preceding operator name from the data with exactly one byte of white space. It is simplest to use a space rather than a carriage return or linefeed character. So, in the example above, there should be exactly one space character between the **exec** operator and the binary data. Also, binary data is only appropriate to send over binary-clean channels, such as LocalTalk®. If you do not have a binary-clean channel, for example, if you're using Adobe Standard Protocol over a serial or parallel channel, then you should also ASCII85 encode your LZW compressed data. In that case, your compressed code will decompress correctly in the following context:

```
%!PS-Adobe 3.0
%%Title: (Example using ASCII85 and LZW filters)
%%EndComments
%%Page: 1 1
currentfile /ASCII85Decode filter /LZWDecode filter cvx
%%BeginData: 100 ASCII Lines
exec  ... ascii85 encoded, lzw compressed code here ...
...additional 99 lines of ascii85 lzw data ...
%%EndData
%%EOF
```

The LZW compression method is said to be the subject of United States patent number 4,558,302 and corresponding foreign patents owned by the Unisys Corporation. Adobe Systems has licensed this patent for use in its products. Adobe's license does not cover Independent Software Vendors (ISVs) that generate PostScript language programs containing LZW-compressed segments. If an ISV wishes to license the patent for that purpose, Unisys has agreed that ISVs may obtain such a license for a modest one-time fee. Further information can be obtained from: Welch Licensing Department, Law Department, M/SC2SW1, Unisys Corporation, Blue Bell, Pennsylvania, 19424.

For more information on Level 2 filters, see section 3.13 of the *PostScript Language Reference Manual, Second Edition*. Also see Technical Note #5115, "Supporting Data Compression in PostScript Level 2 and the filter Operator."

continued on page 6

Questions & Answers

Q I'm getting the error message below from my PostScript language code. Why would I receive this error when get is a standard operator that should be defined on all printers?

%%[Error: undefined; OffendingCommand: get]%%
%%[Flushing: rest of job (to end-of-file) will be ignored]%%

A This error message does not indicate that **get** is undefined; rather, it indicates that the interpreter encountered an undefined name while it was executing **get**. An **undefined** error can occur in a couple of ways. A common occurrence of the **undefined** error results when an incorrectly typed operator is encountered; for example you enter "noveto" instead of "moveto". The interpreter attempts to find noveto in its dictionary stack and returns an **undefined** error when it fails. You may also encounter this error when trying to execute Level 2 PostScript language code on a Level 1 interpreter. In that scenario, typically you receive an **undefined** error with a Level 2 operator as the offending command.

Another way you may receive an **undefined** error is less obvious, and has been the source of many debugging hours for PostScript language programmers. Several PostScript language operators use literal names as their arguments. During the course of its execution, an operator will attempt to look up the value of the name passed to it. If the operator cannot find that name defined in any of the dictionaries on the dictionary stack, an **undefined** error will occur. In this case, the operator name may be returned as the offending command in the error message. Here are some examples of possibly misleading **undefined** errors.

Code Sample:

```
%!
% Example when the get operator produces an undefined
error
userdict /Hello get
%%EOF
```

Error Produced:

%%[Error: undefined; OffendingCommand: get]%%

Reason for Error:
The name, Hello, is not defined in userdict. This error is not encountered until **get** is executed.

Code Sample:

```
%!
% Example of filter operator producing undefined error
currentfile /bogusname filter def
%%EOF
```

Error Produced:

%%[Error: undefined; OffendingCommand: filter]%%

Reason for Error:
The error occurs because bogusname is not one of the standard filter names supported by the **filter** operator.

Code Sample:

```
%!
% Example of load operator producing undefined error
/s /someproc load def
%%EOF
```

Error Produced:

%%[Error: undefined; OffendingCommand: load]%%

Reason for Error:
someproc is not found in any dictionary on the dictionary stack. This error is encountered during execution of **load** operator.

Code Sample:

```
%!
% Example when currentdevparams produces an undefined error
(%disk99%) currentdevparams {= =} forall
%%EOF
```

Error Produced:

%%[Error: undefined; OffendingCommand: currentdevparams]%%

Questions & Answers

Reason for Error:
There is no device named %disk99% associated with the PostScript output device that interpreted this code. Since the **undefined** error occurs during execution of **currentdevparams**, the offending command is **currentdevparams**.

The above examples illustrate that the offending command associated with an error message may be misleading. However, just knowing this fact may not be sufficient to help you debug your program. You still don't know which name caused the **undefined** error. In most cases, the name that was undefined is on the top of the stack after the error occurs. So, you may be able to use an error handler to obtain the offending name, which should be captured in the ostack array in the $error dictionary. The May 1993 issue of the *ADA News* has an article on error handling in PostScript language programs. You may also want to see the November 1992 issue of the *ADA News* which has a list of commercial PostScript language error handlers that you may choose to use.

Q I have a printer with an attached hard drive that I normally use for font storage. I want to write an image out to that drive, and then refer to that image data file in a PostScript language job. How do I do this?

A Here is some sample code for writing hexadecimal image data out to a file. The file will be created on the first available writable device connected to the printer. In the sample code, the file on the drive is called "image1", but you can modify the code to call it whatever you want. Also, the hexadecimal data below is just some sample 1-bit data, which you can replace with your real image data.

```
%!
%%Title: (Example that writes hex image data to disk)
%%EndComments
%%BeginSetup
/fileout (image1) (w) file def % open image1 for writing
/buff 128 string def
{
    currentfile buff readhexstring {
         fileout exch writestring
    }{
         dup length 0 gt {
             fileout exch writestring
         }{
             pop
         } ifelse
         fileout closefile
         exit
    } ifelse
} bind
%%BeginData: 7 Hex Lines
loop
003B00 002700 002480 0E4940
114920 14B220 3CB650 75FE88
17FF8C 175F14 1C07E2 3803C4
703182 F8EDFC B2BBC2 BB6F84
31BFC2 18EA3C 0E3E00 07FC00
03F800 1E1800 1FF800
```

After sending the above code down to the printer, you can refer to the file on the hard disk in the same way as you would normally refer to **currentfile** in your PostScript language code. Note that in the above example the code reads in hexadecimal data with the **readhexstring** operator but

continued on page 8

Colophon

This newsletter was produced entirely with Adobe PostScript software on Macintosh and IBM® PC compatible computers. Typefaces used are from the Minion™ and Myriad™ families, all of which are from the Adobe Type Library.

Managing Editors:
Nicole Frees, Debi Hamrick

Technical Editor:
Jim DeLaHunt

Art Director:
Karla Wong

Designer:
Lorsen Koo

Contributors:
Tim Bienz, Nicole Frees

Part Number ADA0050 9/94

writes out binary data with the **writestring** operator. There is no reason to store hexadecimal data on a printer's storage device since such files are twice as large as binary files and the communication channel between the interpreter and the disk is binary clean.

Below is some sample code that prints the image stored by previous code sample. Since the data is in binary form in the file, we use **readstring** in our data acquisition procedure for the **image** operator.

```
%!PS-Adobe-3.0
%%Title: (Example that images data stored on disk)
%%EndComments
%%BeginSetup
/PaintTarget (image1) (r) file def
/Inbuf 3 string def % one scan line
%%EndSetup

%%Page: 1 1
54 112 translate
120 120 scale
24 23 true [24 0 0 -23 0 23]
{PaintTarget Inbuf readstring pop}
imagemask
PaintTarget closefile
showpage
%%EOF
```

The above code is useful for images that are used in several documents, such as your company logo. If you do set up a system that relies on certain files being present on the hard drive, we strongly recommend that you document the above procedure and archive the original data for future use. §