Adobe Developers Association

ADA news

In This Issue

Volume 4, Number 2

p . 1
Adobe Premiere Plug-In API

p . 2
How to Reach Us

p . 7
Developing with
Adobe Fetch

p . 9
Developing with
Adobe PageMaker

p . 11
Acrobat Column

p . 13
Developing with Illustrator

p . 14
PostScript Language
Technologies

p . 16
Questions and Answers

The Adobe Premiere Plug-In API

The news services seem to constantly report on the latest in the digital video industry. Much of what is discussed remains hype, but there are examples of digital video successes, particularly in the role of video editing. Adobe Premiere™ is one of those successes. The plug-in architecture of the program makes it possible for you to be a part of that success.

The Adobe Premiere program allows the user to capture, edit, view, and output digital movies on a personal computer. Media "clips", including movies, graphics, and sounds, are collected in a "project" and then compiled into digital movies. Effects such as combining two clips or transitioning between two clips can be applied. Filters can be applied to a movie to produce distortions or other effects in the same vein as Adobe Photoshop™ filters.

The latest versions of the Adobe Premiere application for Macintosh® and Windows® use a plug-in architecture to produce these effects and extend the functionality of the program. Adobe Premiere plug-ins are stand-alone pieces of code loaded as needed by the main program. Macintosh plug-ins are code resources and Windows plug-ins are DLLs. Most Adobe Photoshop version 2.5 filters will work with Adobe Premiere; however, creating a Premiere-specific version of the filter will allow it to be more memory efficient and flexible.

Types of Plug-Ins

Adobe Premiere provides several different types of filters. The basic plug-in types are discussed here. Video and audio filters and export modules operate on some defined data and process it to a new form. Adobe Premiere handles the overhead involved with retrieving and storing the frames and sound data, allowing you to concentrate on the effect. Hardware control modules control video hardware in or attached to your computer.

Adobe Premiere also handles part of the user interface of the filter. At start-up, each plug-in in the Plug-Ins directory will be added to the appropriate menu or dialog. In some cases, Adobe Premiere provides some additional interface functionality. In the case of transitions, for instance, Adobe Premiere software displays a preview of each transition effect in the transitions dialog. This interface is handled automatically by Adobe Premiere. Your plug-ins can handle additional interface issues as needed, usually by using a dialog.

Transitions and Video Filters

A transition in Adobe Premiere is like a transition in film, where two separate segments have some appropriate effect to smoothly change scenes. A transition plug-in is given a series of frames from two digital video sources and processes them into one final frame. Examples of transitions include a fade to black and then to the new video clip, and a page turning effect.

continued on page 2

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

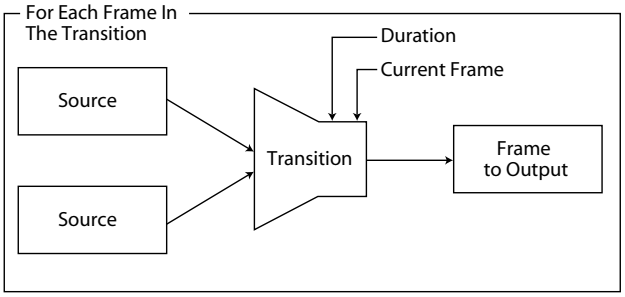
MAIL:

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

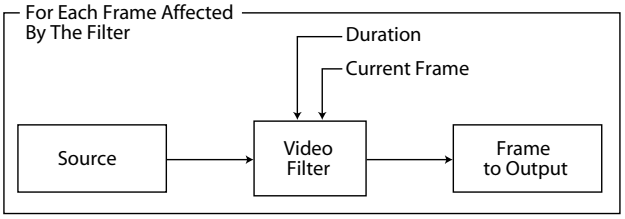
Europe:
Adobe Developers Association
Europlaza
Hoogoorddreef 54a
1101 BE Amsterdam Z.O.
The Netherlands

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

Premiere Plug-In API



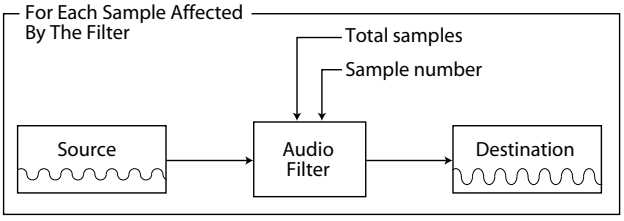
A video filter is similar to a filter in Adobe Photoshop. Rather than merging two frames into one, video filters process each frame of an affected clip into a final form.



Transitions are almost always time-variant; video filters can use a time factor if needed. Both types of plug-ins are given the total duration in frames of the effect and the current frame number. Both transitions and filters may present a dialog to obtain and store additional information about how they should process the affected frames.

Audio Filters

An audio filter in Adobe Premiere takes an audio source and processes it into a final form. An example of an audio filter included in the software package is Echo, which creates an echo effect where the user controls the delay and other variables.



As with video effects, audio filters may be time-variant. The filter is given the total duration in samples of the filter and the sample number of the first sample in the source buffer. Audio filters may present a dialog to collect information to use in processing the audio.

Adobe Japan Has Moved!

Please note that Adobe's Tokyo office has moved. Their new location is as follows:

Adobe Systems Co., Ltd.
Yebisu Garden place Tower
4-20-3 Ebisu, Shibuya-ku
Tokyo 150 Japan

Tel: 81-3-5423-8169
Fax: 81-3-5423-8204

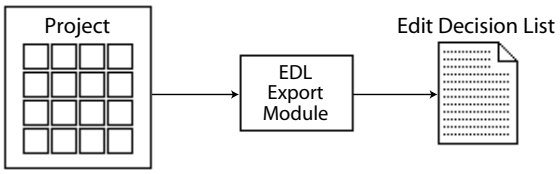
Premiere Plug-In API

Export Modules

Export modules are used to output all or part of the video being processed. There are two types of export modules: data and 'edit decision lists' (EDLs). Both types of plug-ins are called when the user chooses an item from the Export command of the File menu.

Export data modules work with the clip information and output it to some other format. The plug-ins can request to work with audio, video, or both. Export modules normally put up dialogs to ask the user for appropriate export parameters and a destination file.

The EDL export plug-in's job is to export the current project into a text EDL format. Usually these EDL files are used to drive more traditional video processing devices.

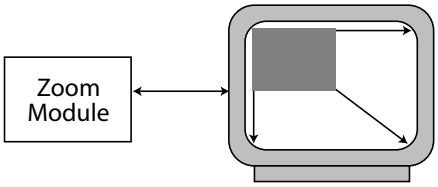


The EDL export module is provided with information about the project in the Adobe Premiere application's Construction window, which is normally used to generate a text file.

Export modules do not necessarily have to output information to files. Adobe Premiere software's "Print To Video" export module is a good example of a module where the output is to the screen rather than to a file. An EDL plug-in might directly control several hardware devices to assemble a project from source tapes.

Hardware Control Modules

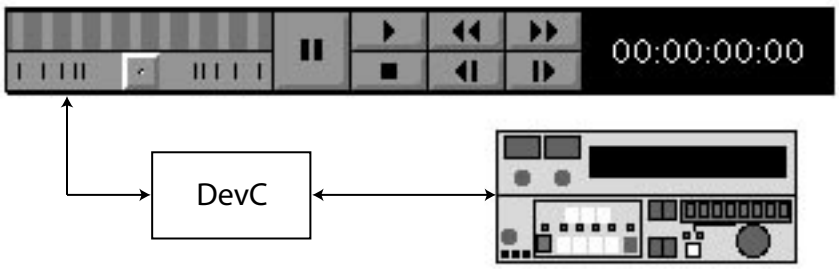
The Adobe Premiere API allows plug-ins to directly control your computer and video hardware. A zoom module in Adobe Premiere handles hardware-specific details of zooming and video card mode-switching.



continued on page 4

Premiere Plug-In API

A device control module allows Adobe Premiere to control hardware devices such as tape decks or laser disc players.



Device control modules are called by parts of Adobe Premiere that take video input, like the Movie Capture window. A device control module's most important functions are to set hardware operating modes, tell Adobe Premiere what mode the hardware is in, and provide Adobe Premiere software with timecode from the hardware.

The Plug-In Mechanism

The plug-ins described above all use a similar calling mechanism. To execute one, the Adobe Premiere program first loads the plug-in code and then jumps to a predefined entry point, which will be defined similar to:

```
short VideoFilter( short selector, VideoHandle theData);
```

Adobe Premiere uses a message passing scheme to control the plug-ins; the message is passed in the selector variable. A message which all plug-ins receive is an execute message. This is where the plug-in would apply its effect to the data Adobe Premiere supplies. Another common message is a setup message, where interaction is done with the user. Most often, the plug-in uses platform specific dialog manager calls to handle this interaction. For the above transition, these messages are defined as:

```
#define  fsExecute  0
#define  fsSetup   1
```

Other messages which might be passed are more plug-in specific. A hardware control module might receive the message cmdZoomIn, for instance.

Premiere Plug-In API

The entry point of your code receives the message and related data. Your `main()` function would look something like:

```
//====================================================
// Perform the filter
//====================================================
pascal short main(short selector, VideoHandle theData)
{
    short result = 0;

    // Save the current state
    ...

    // Act according to the selector
    switch (selector) {
        case fsExecute:
            // do the transition
            ...
            break;
        case fsSetup:
            // get any information
            ...
            break;
    }

    // Restore the entry state
    ...

    return(result);
}
```

The data passed to the entry function contains the information necessary for a plug-in to run. A video filter would receive a record with the following information:

```
typedef struct {
    Handle                  specsHandle;  // User data for effect
    GWorldPtr               source;       // The source frame
    GWorldPtr               destination;  // The output frame
    long                    part;         // part/total = %complete
    long                    total;
    char                    preview;      // No longer used
    Handle                  privateData;  // Data and function to
    VFilterCallBackProcPtr  callBack;     //     get other frames
    BottleRec               *bottleNecks; // Beyond article scope
    short                   version;      // Zero
    short                   sizeFlags;    // Info on output
    long                    flags;        // Unused
    short                   fps;          // The frames/second
} VideoRecord, **VideoHandle;
```

continued on page 6

Since each type of plug-in requires different data, the record passed will depend on the plug-in. For instance, notice the `source` pointer; if this were a transition plug-in, the record passed would have two sources since a transition works on two frames at a time.

Given the message and data, the rest of the plug-in code is up to you. It might be massaging the bits of the source frames or outputting the project information in some special format.

Platform Differences

With version 4.0 of Adobe Premiere for Macintosh and Windows the platform differences have been minimized, though there is not a one-to-one feature correspondence. This is also true of the platform-specific implementations of the plug-in APIs. The Macintosh version provides an extensive library of routines that are useful to the plug-in developer. This is not provided in the Windows version of the API. This is not an issue since they are not critical in the development of filters.

Other differences in the APIs are primarily with basic platform differences. For instance, Macintosh developers have the concept of a GWorld for graphics, for which there is no corresponding structure in the Microsoft Windows API. Adobe Premiere for Windows defines a similar mechanism, a PWorld, to address this. The API differences that exist are not a factor in creating filters for both Macintosh and Windows.

Conclusion

The world of digital video will surely expand as the idea of "multimedia" begins to mature. Adobe Premiere for Macintosh and Windows will be a major player in this future; and by writing plug-ins for the program, you can easily take part in this success. The plug-ins can be general effects such as transitions and filters to process video, or more specific modules to integrate hardware with Adobe Premiere. The API offered by Adobe is an exciting opportunity for developers interested in or committed to digital video.

For more information about developing Adobe Premiere plug-ins, your best course is to join our Graphic Applications Plug-in Developers Program. This program gives you copies of the latest Adobe Premiere Software Development Kit (SDK), plus support in your development efforts. The SDK is also available electronically from Adobe's FTP site, WWW (World Wide Web), and bulletin board service. §

Developing With Adobe Fetch

Adobe™ Fetch™ is a multimedia database product for the Macintosh® that allows customers to catalog, browse, search, retrieve, and reuse graphic, movie, and sound files.

The pnot Resource

A Standard from Apple Computer

You may be wondering how Adobe Fetch can catalog so many different file types and how support for additional file types can be added without revisions to the Fetch product. There are three ways Adobe Fetch can obtain thumbnails from files. One way is through the program's own built-in filters, which support a limited number of file formats. A second way is through Apple's Macintosh Easy Open technology. In this article, we discuss the third way for Fetch to obtain thumbnails, which is for graphic applications to include pnot resources in their files.

Some pnot History

With QuickTime® 1.0, Apple provided a standard way for applications to add visual previews to document files. These previews can be viewed from the standard file selection dialog with any application that uses the QuickTime software's StandardGetFilePreview function.

The release of QuickTime 1.5 expanded the preview interface. Previews may now contain movies and sounds, in addition to pictures. Furthermore, the preview types may be expanded by developers.

QuickTime, versions 1.5 and greater, also allows developers to store other types of information about a file, in addition to the visual preview. This information can include a brief description of the file, a list of keywords, or any other appropriate kind of data. Each piece of file information is stored with its own language code to allow a single file to be used in various countries. This should make life easier for content providers. Each piece of file information is also tagged with a modification date to make it easy for applications to determine when the data has been changed.

pnot Data Structures

QuickTime uses a resource of type 'pnot' with id 0 to store the visual preview information. Its structure with QuickTime 1.0 is shown below:

```
typedef struct pnotResource {
     unsigned long     modDate;    // last preview modification date
     short             version;    // version of this pnot Resource
     OSType            resType;    // resource type contains the preview
     short             resID;      // resource id of preview
} pnotResource;
```

With QuickTime 1.5 this structure was expanded to:

```
typedef struct pnotResource {
     unsigned long   modDate;      // last preview modification date
     short           version;      // version of this pnot Resource
     OSType          resType;      // resource type contains the preview
     short           resID;        // resource id of preview
     short           numResItems;  // number of additional file
                                      descriptions
     pnotResItem     resItem[ ];   // array of file descriptions
} pnotResource;
```

This extended structure allows for an unlimited number of additional pieces of file information. Each piece contains a reference to its data using the following structure:

```
typedef struct pnotResItem {
     unsigned long   modDate;     // last modification date of this item
     OSType          useType;     // what type of data this is
     OSType          resType;     // resource type containing this item
     short           resID;       // resource id containing this item
     short           rgnCode;     // Macintosh Region code for language
     long            reserved;    // set to zero
} pnotResItem; *pnotResItemPtr;
```

The useType field indicates the purpose of the data pointed to by this item. There are currently two different values defined for this field. KeyW indicates that it points to a list of keywords, typically stored in an STR# resource. Desc indicates that the item points to a brief text description of the file, typically stored in a TEXT resource. Apple encourages developers to expand the list of types to include additional relevant kinds of information.

Fetch recognizes Prev useTypes, as well as KeyW and Desc useTypes. A Prev useType indicates that it points to a full-size visual preview stored in a PICT resource.

Special Considerations

QuickTime 1.5 was modified so that it will not destroy any data stored in the additional pnotResItem fields when creating, updating, or removing file previews. Any applications which work with file previews, and do not use the QuickTime software's routines for managing previews, should do the same.

If QuickTime encounters a file which contains a pnot resource but the resType field of the pnot Resource data structure is set to 0, it will treat it as if the file has no preview. This allows developers to add a list of keywords, or a text description, without being forced to create a visual file preview.

The modification date of a preview is an inexact number. The problem is that the preview's modification time is stored in the file, and in order to store that time in the file it is necessary to modify the file. This causes the file's modification time to be changed. Applications should allow for up to a one minute difference between the file and preview modification times before considering the times to be different.

How Fetch Uses pnot Information

When cataloging a file, the Fetch application will first look in the pnot resource for preview information. If the pnot resource references a PICT resource, Fetch will extract the PICT data from the resource, generate a 112 x 112 pixel thumbnail from that data, and store it in the database. If the extended pnot resource references TEXT or STR# resources, Fetch will extract the keywords from the STR# resource and the description from the TEXT resource and store them in the database. These keywords and descriptions will appear in the Fetch Item Info dialog box.

When a user previews an image in Fetch, the Fetch application will again look to the pnot resource to determine if a preview is referenced there. Fetch will extract the preview from the PICT resource that is referenced by the Prev useType in the extended pnot structure. If there is no preview referenced in the extended pnot structure and Fetch does not have a built-in filter for the format, Fetch software will not provide the user with a preview.

Application Support of pnot

Applications typically store a small thumbnail-size PICT in a PICT resource referenced by the pnot resource structure and a full-sized 32-bit QuickTime compressed preview in a separate PICT resource referenced by the Prev useType field in the extended pnot structure.

In the following example, PICT 128 contains a 112 x 112 pixel, 32-bit, QuickTime compressed PICT. PICT 129 contains full-sized, 32-bit, QuickTime compressed PICT. STR# 140 contains a list of keywords in American English. TEXT 150 contains a text description in American English.

```
struct pnotResource pnotSample =  { date, 1, 'PICT', 128, 3,
                                    { date, 'Prev', 'PICT', 129, 0, 0}
                                    { date, 'KeyW', 'STR#', 140, 0, 0}
                                    { date, 'Desc', 'TEXT', 150, 0, 0} };
```

With this simple enhancement to your graphic application, you will be allowing your customer to easily organize and search on their graphic files in Adobe Fetch. Even if Fetch has a built-in filter for the file formats that your application exports, storing a thumbnail and preview in the pnot resource is still a good idea. Cataloging and previewing files with a pnot resource is dramatically quicker if you do this.

More Information

For more information about the pnot resource and how to make your applications pnot aware, refer to the Fetch Compatibility Toolkit. This toolkit is located on the Spec Pack CD from the Adobe Developers Association or can be downloaded from the Adobe Bulletin Board and ftp server. When you are ready to implement pnot for your application, you should order the QuickTime Developer's Kit from Apple to ensure you have the latest information.

Developing With Adobe PageMaker

When a new developer first attempts to write a plug-in (formerly called an addition), one of the most confusing steps is that of getting the plug-in to show up in the appropriate menu. The process of registering a plug-in was not described clearly in the initial documentation, and so will be revisited in the next printing of the plug-ins documentation. In the meantime we'll address plug-in registration for the Macintosh in this month's column and registration for the PC in next month's column.

Registration Basics

When the Adobe PageMaker™ program is launched, it finds and registers all plug-ins that are in the Addition sub-folder within the Usenglsh sub-folder within the standard Aldus folder (Aldus:<language folder>:Addition). Adobe PageMaker reads the resource information for the plug-in, which is a code resource on the Macintosh and a DLL on the PC, and retrieves the name and ID of each plug-in. With this information PageMaker can build the Additions menu. When a plug-in is selected from the menu, PageMaker retrieves the id for that plug-in and loads the plug-in into memory.

On the Macintosh

As mentioned previously, when the PageMaker application is launched, it finds and registers all plug-in libraries that are in the Addition sub-folder. The plug-in library must be built as a code resource; and, in order for it to be registered by PageMaker properly, it must contain an ADNI resource in its resource fork. For each plug-in in the library, the ADNI resource should contain the following required fields: Menu name index (that corresponds with an entry in a STR# resource), Appear in menu, plug-in version, and Function ID. In the course of registering the plug-ins, Adobe PageMaker creates a menu item in the Utilities–Additions sub-menu for each name listed in the ADNI resource.

To help you create this ADNI resource Adobe has provided a ResEdit™ template called *adni.tmpl*. This template can be found in the RESOURCE folder of the PageMaker Plug-ins SDK along with a README file containing directions on how to install the template into ResEdit.

ResEdit

The ResEdit template, *adni.tmpl*, simplifies the creation of the ADNI resource. It is recommended that you use ResEdit 2.1 or later. To use this template you simply copy it to the ResEdit preference file. This is very straight-forward for those that have used templates; for those who have not, simply follow these steps:

1) Open the *adni.tmpl* file with ResEdit.

2) Copy the TMPL record into the clipboard.

3) Open the ResEdit file in the Preferences sub-folder in the System folder.

4) Paste the TMPL record into the ResEdit preference file.

Now you're ready to use it.

To create an ADNI resource in your plug-in's resource fork simply open the plug-in's resource file in ResEdit and create a new resource. If you installed the ADNI template correctly there will be a ADNI resource type in the "Select New Type" dialog. Double click on this type and you will get a window that looks similar figure 1 that will allow you to define a ADNI resource.

In the first block you should enter the version of your plug-in interface. In the example in figure 1 we are using $0100 to represent version 1.00. You should enter $0100 in this field.

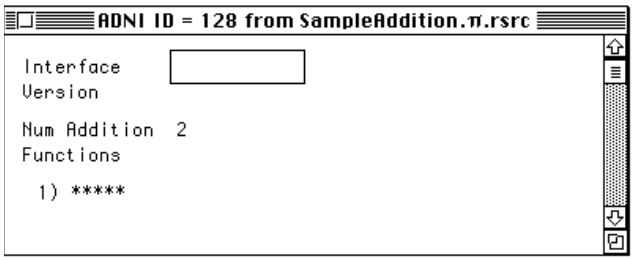


Figure 1

Next select the "1) *****" in the template and then choose "Insert New Field (s)" from the Resource menu. This will cause the template to "fill out" and you can finish defining the ADNI resource as shown in figure 2.

The Menu Name Index field refers to the ID of a STR# resource that should contain the name of your plug-in as you want it to appear in the Additions sub-menu.

The "Reserved Bit" radio buttons should all be '0'.

The "Appear in Menu" radio button needs to be set to '1' in order for the plug-in to appear in the Additions sub-menu.

The "Addition Version" field is for the version of your plug-in. Again, in the example in figure 2, we use "$0100" for version 1.00.

The "Function ID" field is for a number that can uniquely identify each plug-in in the library. Since it's possible to have more than one plug-in per plug-in library the Menu ID's and Function ID's must be unique for each. When the plug-in library is invoked, the particular plug-in can be determined by retrieving the function id via the macro PBGetID( ).

To create a second ADNI resource for a second plug-in in the library simply select the "2) *****" and then choose "Insert New Field (s)" from the Resource menu. This will cause a second template to "fill out" and you can define it as before, remembering to use a different menu and function id.

After you've finished, save the resource and you're ready to go. Next month we'll discuss plug-in registration on the PC.

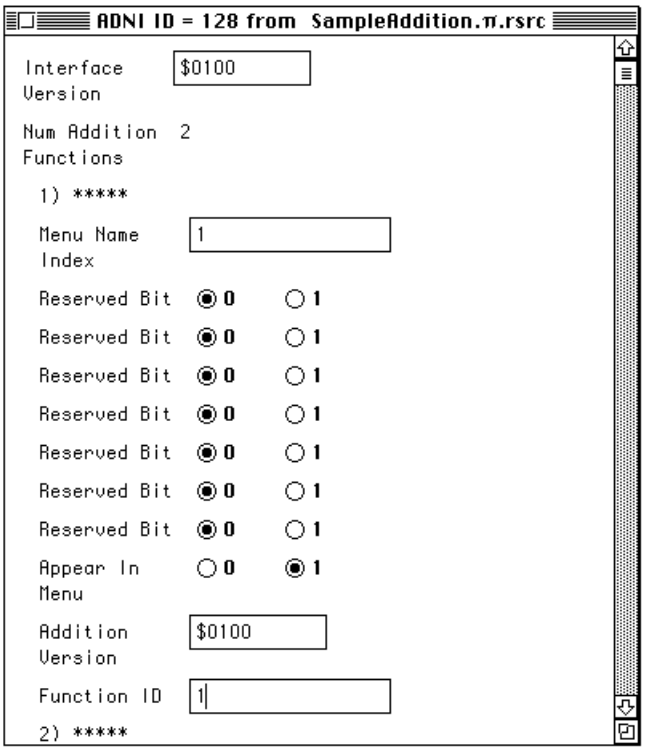
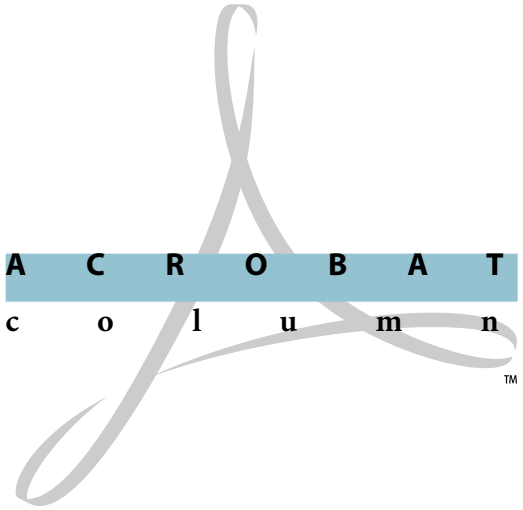


Figure 2



The development system available for the Adobe Acrobat™ 2.0 family of products allows developers to integrate Acrobat into their applications in a variety of ways. Here are our suggestions for integrating Acrobat software with document management systems.

Create PDF Automatically

Since PDF can represent any application file from any platform, it is ideal for use as a universal file format. Document management systems can provide an option to automatically generate a PDF file whenever an application file is checked into the system. This can be done by driving the PDF Writer and/or the Acrobat Distiller™.

We recommend that you create a PDF file whenever a file is checked in or modified, and then associate this PDF file with the original application file.

Use Acrobat for Document Viewing and Annotating

Users frequently do not need, or are not permitted, to edit a file, but still wish to view and possibly add comments to it. By using Acrobat software as the viewing and annotating portion of your document management system you allow all users on the system to view any file, regardless of the application or platform where the file was created. The first step to using Acrobat for viewing is to create PDF files automatically (see above). After that, the document management system should show the PDF version of the file whenever a user without edit permission requests the file, or when a user with edit permission requests only to view the file.

Document management systems can either present the PDF file by invoking an Acrobat viewer or by rendering the PDF file into their own window using either IAC or the Acrobat Plug-in API.

Manage Annotations

Acrobat applications can be used for annotation management. Annotations in Acrobat can be kept as separate files from the PDF file they annotate and can be brought up in conjunction with another PDF file. This allows all annotations for a specific PDF file to be viewed simultaneously, and for security to be used in conjunction with annotations.

File System Management

Document management systems generally have their own file system, and do not use the underlying disk file system. For this reason they should override the save, open and quit functions in the Acrobat viewers to allow users to open PDF files directly from the document management system.

Cross-document links in PDF files must contain the file id instead of the pathname in order to access files in the document management system. To do this, write a plug-in that replaces the Acrobat Exchange cross-document link functionality.

Extract Thumbnails

Where the document management system presents an icon or miniature representation of the file to the user, the system can extract a thumbnail image directly from a page of the PDF file using the plug-in API.

Read and Write Document Info Fields in PDF File

PDF files created with Acrobat 2.0 contain document information such as title, subject and keywords. Document management systems should populate the document information fields when creating PDF files. These fields can then be used in a variety of ways within the system, including as search qualifiers.

Index and Search PDF Files

Many document management systems allow users to search collections of documents and present the search hits highlighted to the user. For document management systems that incorporate Acrobat viewing technology, indexing and searching can be done through the Acrobat system. Adobe is working directly with full text search vendors to include support for PDF files in existing full text search systems. If your document management

system is licensing full text search technology from an outside vendor, you should discuss Acrobat integration with that vendor.

For indexing and searching PDF files directly, we provide support through IAC and API calls. For indexing PDF files, we provide text extraction APIs. Text extraction also supplies position information that can be used to highlight search hits in the original PDF file. The text extraction tools are provided as calls in the plug-in API on the Acrobat Exchange 2.0 viewer platforms (currently, Macintosh and Windows); highlighting is provided as API and IAC calls. A standalone text extraction toolkit is currently available for SunOS™ 4.1.x and Solaris® 2.3, and will be available soon for HP/UX. These text extraction tools can be the basis of a PDF-only indexing product or can add support for PDF files to a current indexing product.

Provide Markup

As well as using the notes feature in Acrobat for commenting on documents, users of document management systems may wish to use a free-form markup tool, highlighting or other custom annotation. Document management systems can provide these tools to their users as plug-ins for Acrobat Exchange.

Associate the PDF file with its Original Authoring Document

One convenient way to keep documents and their PDF representations together in the document management system is to store the document inside the PDF file. Through the use of 'private data' in a PDF file, a document management system can embed the entire authoring document as part of the PDF file that represents it. This way, not only is the resulting electronic document viewable by anyone with Acrobat, it is also editable by users who have the authoring application. A plug-in for Acrobat would need to be written to allow embedding and extracting of the authoring document. This plug-in would simply add the authoring document as a private data stream when embedding and, when extracting, save the stream to a temporary file and invoke the authoring application. Embedding can be done at the time users check in the application files to the system and the PDF file is automatically created. Acrobat viewers ignore this private data. Embedding authoring documents in PDF files will greatly increase the size of the PDF file and should not be done in all cases.

An association between the PDF file and the authoring document can also be maintained through the use of link in the PDF file. In Acrobat 2.0, links can be created that will invoke files and their associated applications. If a document management system places such a link in the PDF file, then the users can invoke the original authoring document by executing the link.

And More

These are just a few suggestions for how to integrate Acrobat technology into document management systems. The Acrobat Software Development Kits contain all the information you need to implement these suggestions and more.

Developing With

Adobe Illustrator

Writing PowerPC™ Native Plug-Ins for Adobe Illustrator™

Despite the fact that the plug-ins installed with the PowerPC version of Adobe Illustrator are the same as their 68K brethren, the API does provide a mechanism for creating native PowerPC plug-ins and fat plug-ins. This allows developers of speed-critical filters to take advantage of the hardware. The mechanism is simple enough, though, that all plug-in developers should consider going native. The PowerPC compiler used in this discussion is a part of the Metrowerks CodeWarrior environment.

There are several steps you need to follow to create a native PPC filter. The first is to make certain that any Macintosh toolbox callback routines are declared Universal Procedure Pointers. UniversalProcPtrs (UPPs) are an interface to allow the Power Macintosh® to operate in a mixed mode environment, where it switches between 68K and PowerPC code. The most likely use of a toolbox ProcPtr within an Adobe Illustrator plug-in is a filterProc passed to the dialog manager to allow customized behavior. The Macintosh Dialog Manager is still 68K code, but the filterProc of your native plug-in is PowerPC code. To enable the different processor code to intermingle, we need UPPs. It is a simple process to coerce the dialog filter procedure callback, as the following example shows:

```
ModalDialog( (ModalFilterUPP)params->functions->ModalDialogProc, &hit );
```

That's all there is to it! You will need to identify the procedures where the coercion is necessary and some may be a little more involved than others, but in simple cases the ModalFilterUPP is the only change needed.

The second step is to notify Adobe Illustrator that native PPC code is available. PowerPC code is kept in the data fork of a file rather than the resource fork. A PPC native application uses a cfrg resource to indicate that PowerPC code is available. Adobe Illustrator uses a similar mechanism, denoting a native PPC filter's existence with an ARTI resource. This resource contains two long words (8 bytes) of flags. In the current version of Adobe Illustrator, the only bit defined is the lowest bit of the second long word. It should be set to 1 to indicate that PowerPC code is in the data fork.

```
ARTI (8 bytes) = 0x0000 0000 0000 0001
```

All of the other bits are reserved by Adobe for future use and should be set to 0.

The last step is to build your modified project. Here you must adhere to several conventions. There is one entry point which must be declared to the linker, your main( ) procedure. You don't need an initialization or termination entry point. The Export Symbols option of the PEF preferences must be set to None. Lastly, the project type must be a shared library, *not a code resource.* The file creator and type are the same as 68K plug-ins, ART5 and ARTF respectively.

The rest of the filter is the same as a 68K filter. You still need to include all the normal resources, of course. If you are making a fat plug-in, your final file would include a ARTF resource in addition to the ARTI resource. Both code types coexist and the appropriate code will be used by the program.

When considering whether you want to develop a fat plug-in, there are a few things to keep in mind. Processor intensive tasks will definitely benefit from a native version. The biggest cost of a PPC native plug-in is that PowerPC code will not be as compact as its 68K equivalent. This means that a fat plug-in could be quite large, a potential concern from the standpoint of disk space. If a collection of fat plug-ins were to exceed the capacity of a floppy disk, you might want to evaluate whether they all truly benefit from their PPC components.

You now know everything you need to get started. For more information, a sample plug-in with some expanded documentation on creating PowerPC native plug-ins is available in the Adobe Illustrator 5.5 Plug-In SDK. Contact the Adobe Developer's Association for a copy of the SDK. Go native!

PostScript Language Technologies

In this month's PostScript™ Language Technologies column, we address a question about global memory, which we recently received from one of our ADA members.

Q I would like to define some global variables in my PostScript language program, so that their values will not be affected by save and restore . I thought I could accomplish this by defining the variables in global vm, using setglobal. The *PostScript Language Reference Manual, Second Edition,* states that the "creation and modification of global objects are unaffected by the save-restore operators". However, when I ran the following code, the results were not what I would expect.

```
%!
% global memory test

true setglobal
/myStr (first string) def
/num 3 def
false setglobal

save
/myStr (new string) def
/num num 1 add def
restore
%%EOF
```

After running this short segment of code, I would expect that num would have the value '4', and myStr would now equal "new string", but these variables still have their original values of '3' and "first string". Why doesn't setglobal work as I would expect?

A It is true that a global object's contents are not affected by the **restore** operator. However, your code does *not* modify the contents of a global object, rather it modifies an object in local VM, namely userdict. After the call to **save**, you call the **def** operator twice. When called with an already-defined key, **def** changes the value associated with that key in the current dictionary. Figures 1 and 2 show how memory is affected by these two calls to **def**. The values in userdict are changed, but the string that was allocated in global VM is not changed. The changes to userdict are subject to **save/restore** since userdict is located in local VM.

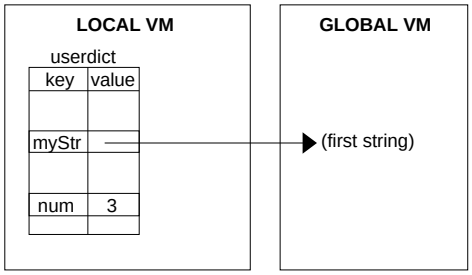


Figure 1. Memory snap-shot before call to save an operator.

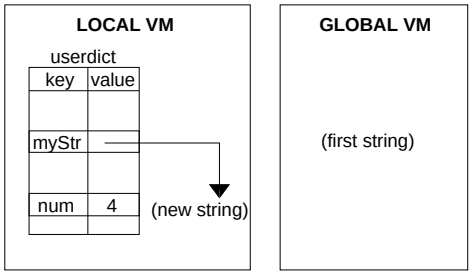


Figure 2. Memory snap-shot before call to restore an operator.

In order to preserve the values of variables across calls to **save/restore**, you may allocate a dictionary in global memory, and define your variables in that dictionary. With a global

dictionary as the current dictionary, calls to **def** will not be affected by **save/restore**. With this in mind, to get the results that you expected originally, you could rewrite your code as follows:

```
%!
% new global memory test

true setglobal
/myGlobalDict 3 dict def
myGlobalDict begin
   /myStr (first string) def
   /num 3 def
end
false setglobal

save
   true setglobal
   myGlobalDict begin
      /myStr (new string) def
      /num num 1 add def
   end
restore
%%EOF
```

The reason that the line "true **setglobal**" is needed after the call to **save** is to ensure that (new string) is allocated in global VM. Otherwise, an invalidaccess error would occur upon the attempt to store a local string object into a global dictionary. The value of **setglobal** is irrelevant to the behavior of the **def** operator; it only affects how new composite objects, such as (new string), are allocated. Also note that you could also use globaldict instead of the explicitly created global dictionary, myGlobalDict.

Colophon

All proofs and final output for this newsletter were produced using Adobe PostScript software. The document review process was accomplished via electronic distribution using Adobe Acrobat software. Typefaces used are from the Minion™ and Myriad™ families from the Adobe Type Library.

Managing Editors:
Nicole Frees, Debi Hamrick

Technical Editor:
Jim DeLaHunt

Art Director:
Karla Wong

Designer:
Lorsen Koo

Contributors:
Matt Foster, Nicole Frees, Mike Mitchell, Carrie Requist, Michelle Sellars

Part Number ADA0055 2/95

Adobe PostScript

Questions Answers

Q With the Adobe Illustrator API, how can I found out if a text object is point text, text in-a-path or text on-a-path ?

A The GetArtType( ) call only indicates that an object is of general type text, kTextArt. You need to use two API calls on the text objects path to obtain the desired information. The GetTextPathPath( ) function works on in-path or on-path text objects. The handle returned by it will be nil if point text objects are passed to it, since point text needs only a position point and not an effective path. GetTextPathOffset( ) only works with text paths of on-path text objects. Other text types do not have an offset position and will cause an error code to be returned. These calls and conditions are used in the C routine below to determine the text type. The routine works with the Macintosh 5.x headers files.

```
enum {
          kPointText = 20,
          kOnPathText,
          kInPathText
};

short GetTextType(AIFilterPB *pb, AIArtHandle textArt)
{
          AIFunctions *k = pb->functions;
          FXErr error;
          AIArtHandle textPath,realPath;
          Fixed offset;

          k->GetFirstTextPath(textArt,&textPath);
          k->GetTextPathPath(textPath,&realPath);

          // If the child is null, then we have point text
          if (!realPath)
                    return kPointText;

          // The text has a child, check to see
          // if it is on-path or in-path text
          if (error = k->GetTextPathOffset(textPath,&offset))
                    return kInPathText;

          return kOnPathText;
}
```

Q With the Adobe Illustrator API, is it normal that the matrix I use for transforming paths cannot be used directly to transform text objects? I have to invert the rotation angle before it works.

A Yes, there is a difference in the rotation matrices used for various art objects. A regular rotation matrix can be used for path art and placed art. The rotation matrix for text must be inverted as shown in order for it to work.

```
FXErr    error;
short    type;
FixedMatrix       matrix;

//set up expected matrix
...
k->GetArtType(art, &type);
if (type == kTextArt) {
          matrix.b = -matrix.b;
          matrix.c = -matrix.c;
          }
...
```