ADA Technical Journal | Volume 1, Number 1

ada technical journal
VOLUME 1, NO. 1

In This Issue

(Click on an article link to read it,
click on the ADA News header to return to this page.)

Adobe

How to Reach Us

Developer Information on the World Wide Web:
www.adobe.com

See the Support and Services section and point to Developer Relations.

Developers Association Hotline:

U.S. and Canada:
(408) 536-9000
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 one work day.

Europe:
+44-131-458-6800

Fax:
U.S. and Canada:
(408) 536-6883
Attention:
Adobe Developers Association

Europe:
+44-131-458-6801
Attention:
Adobe Developers Association

EMAIL:
U.S.
ada@adobe.com

Europe:
euroADA@adobe.com

Mail:
U.S. and Canada:
Adobe Developers Association
Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110-2704

Europe:
Adobe Developers Association
P.O. Box 12356
Edinburgh EH1146J
United Kindgom

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

Feature Article

Adobe Acrobat Viewer Layer of Plug-in API Revealed

Developers no longer need to sign non-disclosure agreements to buy the Adobe Acrobat Plug-ins Software Development Kit (SDK), so the deeper capabilities of the Acrobat Plug-in API can now be revealed.

This article introduces the object-oriented plug-in API and shows how to tailor the Acrobat viewer's user interface to your needs. Finally, it walks you through code samples that add a menu item and toolbar button to the viewer user interface.

Objects and Methods

Plug-ins are software modules that directly communicate with an Acrobat viewer (Acrobat Exchange® or Acrobat Reader) to add capability or change viewer operation. Unlike Interapplication Communication (IAC), the API for plug-ins is essentially platform-independent: its function set is nearly identical on Macintosh, Windows,® and UNIX® systems.

The plug-in API contains over 600 functions, which are structured around a set of objects. Most of these objects correspond to elements of either the viewer's user interface or PDF files.

The Acrobat viewer user interface is modeled as a set of *AV layer* objects (where AV stands for Acrobat Viewer). For instance, an **AVMenu** object represents a particular menu in the menubar; an **AVMenuItem** is a menu item in a menu. An **AVDoc** is the representation of a PDF document in a viewer window.

Adobe Acrobat Viewer Layer of Plug-in API Revealed

PDF file objects are classified as either *PD layer* or *Cos layer* objects. The PD model of objects views a PDF file as a set of high-level objects, such as pages and annotations. A **PDTextAnnot** object is a text annotation and a **PDBookmark** is a bookmark, for example. A **PDDoc** represents an open PDF file. The **Cos** (standing for the self-referential *Cos object system*) layer considers PDF files to be composed of low-level atomic objects, such as text strings and dictionaries. PDF's heritage is apparent at this level: **Cos** objects are all PostScript language objects.

From a developer's viewpoint, objects are opaque encapsulations of data in the Acrobat viewer API, implemented as structs in the C language. The internal structure of these objects is hidden from API users. Each object typically has a set of functions or *methods* associated with it.

The naming convention for classes and methods makes it easy to tell what they are and what they do. Objects' class names have a layer prefix: **AV–**, **PD–**, or **Cos–**. (An **AS–** prefix is used for the Acrobat Support layer, which contains utility objects and functions.) An object type follows the layer name and completes the class name. **AVMenubar** and **AVToolButton** are **AV** layer objects; **PDLinkAnnot** and **PDWord** are **PD** layer objects, **CosDoc** is a **Cos** layer object. Thus an object name identifies its layer and the kind of object it is.

Method names associated with a class start with the class name. This is followed by a verb to describe the action of the method and the verb's object, if it has one. For example, the **AVDocClose( )** method closes an **AVDoc**; that is, it closes the window displaying a PDF document. **AVDocPrintPages( )** is a method that

Adobe Acrobat Viewer Layer of Plug-in API Revealed

prints the pages in an **AVDoc**. **AVMenuGetName( )** gets the name associated with a menu. Knowing this pattern, you can generally tell what a method does or guess the method name for some operation on an object.

For most methods, the first argument is the object on which the method is operating: the **AVDocClose( )** method's first argument is the **AVDoc** to be closed, for instance. This follows the object-oriented paradigm of sending a message to an object telling it to perform one of its methods.

The rest of this article focuses on how a plug-in can change the viewer's user interface using AV layer methods.

AV Layer Objects

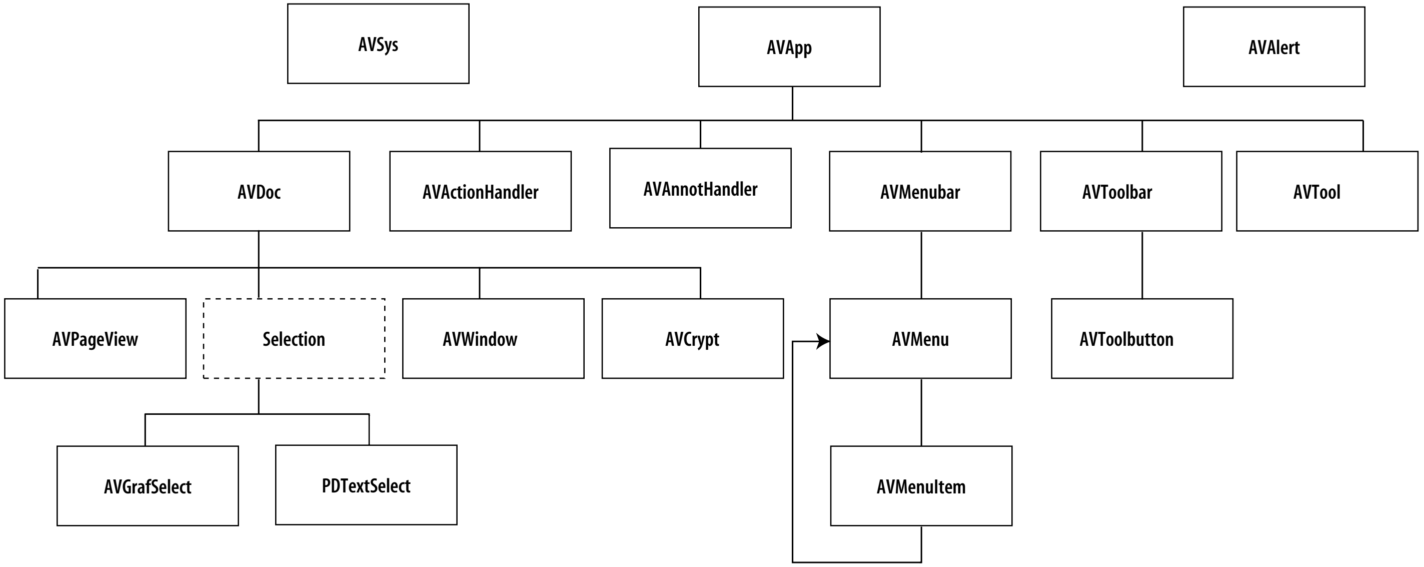
You can add or remove most of the viewer's controls—menu items, toolbar buttons, and so forth—to change the viewer's operation or to control capabilities you add to the viewer.

The **AV layer** contains 16 classes corresponding to parts of the user interface. Some of them, such as **AVMenu** and **AVToolButton**, correspond directly to things you can see in the user interface. Others are more abstract: an **AVPageView** represents a particular view of a document, considering the page displayed, scroll position, and zoom factor.

Figure 1 illustrates the hierarchy of AV layer objects.

Adobe Acrobat Viewer Layer of Plug-in API Revealed

Figure 1 AV layer objects



Adobe Acrobat Viewer Layer of Plug-in API Revealed

Here's a description of key AV layer objects:

- **AVApp**—the Acrobat viewer itself. Through **AVApp**, you perform global viewer operations, such as accessing the currently active tool or the frontmost document.
- **AVDoc**—the view of a document in a window. There is one **AVDoc** for every displayed document.
- **AVPageView**—the area displaying a particular view of a document, considering the page displayed, scroll position, and zoom factor.
- **AVMenubar, AVMenu, AVMenuItem**—the menubar, the menus within it, and the menu items in each menu. There is only one **AVMenubar**.
- **AVToolBar, AVToolButton**—the tool bar and the toolbar buttons on it. There is only one **AVToolBar**. Note the similarity of **AVMenubar** and **AVToolBar**: each holds a set of objects a user can click to perform commands.
- **AVTool**—a set of callbacks (wrappers for function pointers the viewer uses to call back the plug-in) that implement a tool, such as the hand tool, to handle key presses and mouse clicks in an **AVPageView**. A tool may be activated by a toolbar button, but you don't have to have a button for a tool—these objects are not necessarily coupled.
- **AVActionHandler**—a set of callbacks to carry out an action, such as traversing a bookmark.
- **AVAnnotHandler**—a set of callbacks that perform operations on an annotation: creating, displaying, selecting, and deleting a particular type of annotation.
- **AVSys**—an object to access system functions, such as setting the cursor shape and beeping.

Adobe Acrobat Viewer Layer of Plug-in API Revealed

- **AVAlert**—platform-independent support for displaying a simple dialog box.

Now let's survey some of the commonly used methods associated with these objects. This provides a concrete illustration of ways you can manipulate the viewer's interface.

AV Layer Methods

This section lists a few methods you'll find especially useful when writing plug-ins. For a detailed description of each method, see "AV Layer" in the methods section of technical note #5168, "Acrobat Viewer Plug-In API On-line Reference," in the Acrobat Plug-ins SDK.

AVApp

AVAppGetActiveDoc( )	Get the frontmost document.
AVAppGet... –Menubar( ), –ToolBar( )	Get the menubar or toolbar.
AVAppRegister... –ActionHandler( ), –AnnotHandler( ), –Tool( )	Register an action handler, annotation handler, or tool.
AVAppRegisterNotification( )	Register to be notified when a certain event occurs.

AVDoc

AVDocPrintPages( )	Print pages from a document.
AVDocPerformAction( )	Perform an action, such as traversing a link.
AVDocClose( ), AVDocDoSave( )	Close or save a document.

Adobe Acrobat Viewer Layer of Plug-in API Revealed

AVPageView

AVPageView... –GoTo( ), –ScrollTo( ), –ZoomTo( )	Go to a page, scroll to a location, or change the zoom factor.
AVPageViewGetAVDoc( )	Get the AVDoc for an AVPageView.

AVMenubar

AVMenuBarAddMenu( )	Add a menu to the menubar.

AVMenu

AVMenuNew( )	Create a new menu.
AVMenuAddMenuItem( )	Add a menu item to a menu.

AVMenuItem

AVMenuItemNew( )	Create a new menu item.

AVMenuItem *(continued)*

AVMenuItemSetExecuteProc( )	Set the procedure to be called when a menu item is selected.
AVMenuItemExecute( )	Execute a menu item, as if it were selected.

AVToolBar

AVToolBarAddButton( )	Add a button to the toolbar.

AVToolButton

AVToolButtonNew( )	Create a new toolbar button.
AVToolButtonSetExecuteProc( )	Set the procedure called when a button is clicked.
AVToolButtonExecute( )	Execute a button, as if it were clicked.

Adobe Acrobat Viewer Layer of Plug-in API Revealed

AVSys

AVSysBeep( )	Sound the system beep.
AVSysGetCursor( ), AVSysSetCursor( )	Get or set the cursor shape.

AVAlert

AVAlertNote( )	Display a dialog box with a message and OK button.

Bridge methods

Objects are associated with each other. For instance, every **AVDoc** has an **AVPageView**, and vice versa. It's frequently useful to bridge from one object to a related object through *bridge methods*. For instance, you will often want to get the current active **AVDoc**, that is, the document displayed in the front-most window. You can get this from the **AVApp object** using the method **AVAppGetActiveDoc( )**. You can also get the **AVDoc** associated with an **AVPageView** with **AVPageViewGetAVDoc( )**—or get the **AVPageView** for an **AVDoc** via **AVDocGetPageView( )**.

SDK sample

The SDK contains numerous samples of code showing how to use the API objects and methods. The *PLUGINS* directory contains *SAMPLES* directories for Macintosh, UNIX, and Windows platforms. The *SAMPLES* directory has a directory for each plug-in, containing all files needed to create that plug-in. The *SAMPLES.PDF* file in the SDK technical documentation describes each plug-in sample.

Template Sample 1: Adding a menu item

The SDK's Template sample illustrates several simple ways to modify the viewer's interface. Let's examine the sample code to add a menu item to a menu. This code resides in the function **MyInit()** of the file *Template.c.*

You first need to get the menubar, since it contains the menus:

```
AVMenubar menubar = AVAppGetMenubar();
```

Note that you get the menubar from the **AVApp** object—the viewer application itself.

Next, acquire the Preferences menu from the menubar, asking for it by name:

```
AVMenu prefsMenu;
AVMenuItem menuItem;


prefsMenu = AVMenubarAcquireMenuByName(menubar,
  "Prefs");
```

Each menu has a language-independent name to identify it, "Prefs" in this case. Note that **AVMenubarAcquireMenuByName()** can find the menu even though it is a submenu under the File menu.

If **prefsMenu** is not **NULL**, create a menu item and add it to this menu:

```
if (prefsMenu) {
  menuItem = AVMenuItemNew("Template…", "ADBE:Prefs",
    NULL, false, NO_SHORTCUT, 0, NULL, gExtensionID);
AVMenuItemSetExecuteProc(menuItem,
    ASCallbackCreateProto(AVExecuteProc,
    &TemplatePrefs), NULL);
AVMenuAddMenuItem(prefsMenu, menuItem,
    APPEND_MENUITEM);
AVMenuRelease(prefsMenu);
  }
```

Adobe Acrobat Viewer Layer of Plug-in API Revealed

The **AVMenuItemNew( )** method's arguments specify the (possibly localized) name that appears on the menu item, an additional name that's language-independent, whether it's a submenu item, and so on. Notice that the language-independent name, "ADBE:Prefs" has the prefix "ADBE:". Plug-in writers must begin their language-independent menu item names with a developer ID to avoid name collisions when more than one plug-in is present. Contact the Adobe Developers Association to obtain an ID.

AVMenuItemSetExecuteProc( ) tells the Acrobat viewer which function to call back when the menu item is selected. The Acrobat viewer uses *callbacks* to specify such functions.

Much of the viewer's API is *event-driven*: in response to some event, the viewer calls a function the plug-in supplies for that event. In general, the Acrobat viewer requires a plug-in to convert such function pointers to callbacks, or **ASCallback** objects, which the macro **ASCallbackCreateProto( )** creates. Using callbacks is necessary for some platforms so that global pointers are set up correctly. Using the **ASCallbackCreateProto( )** macro also allows compilers to perform type checking on the function's arguments.

In this case, a callback is created for the function **TemplatePrefs( )**, which the plug-in defines. The viewer calls back **TemplatePrefs( )** each time the menu item is selected.

AVMenuAddMenuItem( ) adds the menu item to the Preferences menu—designated by the **AVMenu** object **prefsMenu**.

Adobe Acrobat Viewer Layer of Plug-in API Revealed

Finally, the **AVMenuRelease( )** method releases the **AVMenu** object that was acquired earlier after you're done with it, decrementing this object's use count.

The **TemplatePrefs( )** function has this definition:

```
static ACCB1 void ACCB2 TemplatePrefs(void *data)
{
  AVAlertNote( "Template preferences" );
}
```

The **ACCB1** and **ACCB2** macros are required in every function for which a callback is created. They are automatically defined to be appropriate values for each platform and help provide platform-independent code. The only action **TemplatePrefs( )** takes is to call **AVAlertNote( )**, which simply displays a dialog box containing the text "Template preferences" and an OK button.

Template Sample 2: Adding a toolbar button

The Template plug-in also shows how to add a button to the toolbar, which is similar to adding a menu item to a menu. Adding a button is simpler since you can add it directly to the toolbar—you don't have to acquire an intermediate object like an **AVMenu**.

First, get the toolbar from the viewer:

```
AVToolBar toolBar = AVAppGetToolBar();
```

Next, find where to add the button on the toolbar and add it:

```
AVToolButton toolButton, toolsSeparator;

toolsSeparator =
  AVToolBarGetButtonByName(toolBar,
  ASAtomFromString("endToolsGroup"));
```

Adobe Acrobat Viewer Layer of Plug-in API Revealed

```
toolButton =
AVToolButtonNew(ASAtomFromString("ADBE:PrefsButton"),
GetToolIcon(), true, false);
AVToolBarAddButton(toolBar, toolButton, true,
 toolsSeparator);
AVToolButtonSetExecuteProc(toolButton,
ASCallbackCreateProto(AVExecuteProc,
 &TemplateCommand), NULL);
```

This code positions the new button at the end of the group of buttons that correspond to tools, such as the Hand tool and the Zoom tools. The small space at the end of the Tools group on the toolbar is actually a toolbar button itself, with the name "endToolsGroup". The method **AVToolBarGetButtonByName( )** gets the tools separator button—so you can position the new button relative to it.

AVToolButtonNew( ) creates a button with the desired properties, such as name and icon (which is platform-dependent). **ASAtomFromString( )** creates an **ASAtom** (an Acrobat Support layer object), a hashed token used in place of strings to improve performance. **ASAtoms** appear as arguments in many methods. The function **GetToolIcon( )** should return an icon appropriate to the platform.

AVToolBarAddButton( ) adds the button to the toolbar, positioning it just before the tools separator button, represented by the **AVToolButton** object **toolsSeparator**.

AVToolButtonSetExecuteProc( ) tells the viewer which function to execute when the button is clicked. As in the menu item example, **ASCallbackCreateProto( )** creates an **ASCallback** object from a function pointer.

Adobe Acrobat Viewer Layer of Plug-in API Revealed

Let's make this button execute the menu item defined in the first example. You can define **TemplateCommand( )**—the function the viewer calls when the button gets clicked—this way:

```
static ACCB1 void ACCB2 TemplateCommand(void
 *data)
{
  AVMenuItem templateMenuItem;
  AVMenubar menubar = AVAppGetMenubar( );

 templateMenuItem =
    AVMenubarAcquireMenuItemByName (menubar,
    "ADBE:Prefs");
  AVMenuItemExecute (templateMenuItem);
  AVMenuItemRelease (templateMenuItem);
}
```

Given the menu item name "ADBE:Prefs", the method **AVMenubarAcquireMenuItemByName( )** returns this menu item, which **AVMenuItemExecute( )** executes.

These examples show simple ways to modify the user interface. Much more is possible! You can add new tools, new types of annotations, and the interfaces to use them. The *SAMPLES.PDF* file in the SDK documentation describes each plug-in that's included with the SDK. The file *ROADMAP.PDF* offers general ideas of things you can do with the plug-in API. You can write plug-ins to extend the Acrobat viewer in many more ways than described in this article.

References

The Acrobat Plug-ins SDK contains several documents invaluable to plug-in authors.

Adobe Acrobat Viewer Layer of Plug-in API Revealed

Technical note #5166, "Acrobat Viewer Plug-In API Overview," provides an excellent background for developing plug-ins. It covers all the API layers and objects, summarizing their associated methods. This note also gives insight into various techniques and mechanisms used throughout the API, such as callbacks, notifications, and handlers.

Technical note #5167, "Acrobat Viewer Plug-In API Development," tells how to develop plug-ins on Macintosh, UNIX, and Windows platforms.

Technical note #5168, "Acrobat Viewer Plug-In API On-line Reference," tells you everything you need to know to use each API method: a functional description, arguments, return value, and other associated information, including code samples using the method.

Technical note #5169, "Acrobat Viewer Plug-In API Tutorial," illustrates coding in greater depth, showing how to do various operations with plug-ins.

The SDK's *SAMPLES* directories, located in the *PLUGINS* directory, contain the files for all plug-ins in the SDK. The *UNSUPPTD* directory contains files for additional plug-ins. Each plug-in's directory provides a Metrowerks® CodeWarrior® or Microsoft® Visual C++™ project to build that plug-in. §

DEVELOPING WITH

Adobe Photoshop

Making a plug-in scripting-aware for Adobe Photoshop 4.0

The Adobe Photoshop 4.0 application programming interface (API) introduces a new feature for automation: *actions*. Controlled by the user via the *actions palette*, plug-ins can execute pre-defined commands and batches to allow the user to automate routine and difficult tasks from a single button-click. This article details the process used to update two Adobe Photoshop 3.0.5 plug-ins, *Dissolve* and *GradientImport* (which was previously named *DummyScan*), to make them scripting-aware and controllable via the actions palette.

Welcome to Adobe Photoshop 4.0 Actions

The Adobe Photoshop 4.0 API extends the 3.0.5 specification to include a number of new items. One that affects all the plug-in types and specifications is the new automation system. The main user interface for the automation system is the *actions palette*. The actions palette allows the user to specify commands and plug-ins that are scripting-aware and record multiple events into actions that can be executed with a single mouse-click.

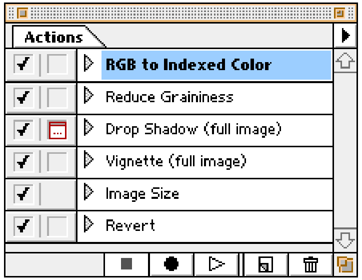


Figure 1 Actions palette

A folder or group of files can also be controlled so that actions can be applied in a batch. This is called *batch-processing* and is part of the Adobe Photoshop 4.0 actions palette.

All plug-ins can be controlled by the scripting system as execute-only commands. This means, whether the plug-in is scripting-aware or not, the action system can execute the plug-in as if the user had invoked it from its menu.

However, if your plug-in is scripting-aware, it goes further and allows the action system to control your plug-in's parameters automactically. This means that, unless there is an error or a parameter that your plug-in needs that it didn't get, your plug-in can operate silently, not needing to show its user

interface and interact with the user. This is extremely valuable for batch-processing and generating special effects that require numerous commands and parameters.

Converting 3.0.5 to 4.0

My task was to take the existing plug-ins that shipped with the 3.0.5 software development kit (SDK) and convert them all to the 4.0 API spec. This proved to be fairly straightforward for some plug-in types, such as simple filters, and more involved for others, such as Import modules, especially with ones that do multiple imports.

This article will detail how I converted two plug-ins, the Filter plug-in module *Dissolve* and the Import plug-in module *GradientImport* to be scripting-aware.

The filter plug-in was vastly simpler, so I'll start with that, and then detail the process for *GradientImport*, which required additional code to handle the multiple import routines.

Scope of this article

More detail is in the SDK

Intimate details on all the scripting parameters and callback suites are available in the Adobe

Photoshop 4.0 SDK, which is available at Adobe’s web site:

http://partners.adobe.com/asn/developer/gapsdk/PhotoshopSDK.html

This article will only address the callbacks and structures that were pertinent to updating the two plug-in example modules. There is much more to the scripting system than is covered in this document. I recommend that you read the SDK for more detail.

Macintosh or Windows?

Scripting implementation, recording, and playback are all part of the Adobe Photoshop API. This means that, except in a few rare exceptions, the callbacks, data structures, and parameters are all exactly the same on both Macintosh and Windows.This article shows Macintosh user interface examples, but the discussion and examples are comparable, if not exactly the same, on Windows.

Starting out

Basic scripting approach

The approach to creating a scripting-aware plug-in is detailed in the scripting chapter of the Photoshop SDK programmer’s guide:

1. Look at your user interfaces and describe the parameters as human-readable text.
2. Create a terminology resource for your plug-in and your PiPL HasTerminology property.
3. Update your plug-in code to record scripting events and objects.
4. Update your plug-in code to be automated by (playback) scripting events and objects.

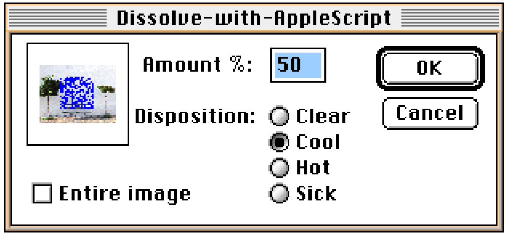

Dissolve-with-AppleScript
Amount %: 50
Disposition: Clear [✓ Cool] Hot Sick
Entire image
OK
Cancel

Figure 2 Dissolve filter user interface

With this in mind, I looked at the user interface for the Dissolve filter. This was the same both on Macintosh and Windows. The Macintosh interface is shown in Figure 2.

After looking at my interface, I was able to describe it as these elements:

1. A button, “OK”, which I don’t need to be recordable.
2. A button, “Cancel”, which I don’t need to be recordable.
3. An amount, expressed as an integer from 1 to 100 representing a percentage.
4. A disposition, expressed as a textual enumeration of a mutually exclusive list of options, either “Clear”, “Cool”, “Hot”, or “Sick”.
5. A flag for “entire image”, expressed as a boolean value of either yes or no.

This should look familiar. It is reminiscent of the resource text used to describe Macintosh dialog items.

When describing these items, it’s important to keep in mind how they will look when represented in the actions palette. Since the actions palette does get loaded with text, it makes sense to use single labels whenever possible and where it will be more readable to the user. I could have used four booleans for “Clear”, “Cool”, “Hot”, and “Sick”, but since “Disposition” should always only be one thing, it makes more sense to have the actions palette display:

```
Dissolve
Amount: 20%
Disposition: Cool
```

Than something like:

```
Dissolve
Amount: 20%
without Clear
with Cool
without Hot
without Sick
```

And speaking of booleans, it’s usually much better form to leave the default value of a boolean as implied instead of explicitly showing it in the actions palette. Again, because the palette can get pretty large, it’s better to only store boolean values that are different than your default. For instance, in the example above, “Entire Image” isn’t listed in the palette because it was in its default (off) state. If it is checked, then I would store it in the action descriptor and it would get displayed as:

```
Dissolve
Amount: 20%
Disposition: Cool
with Entire Image
```

Creating a terminology resource

AppleScript/AppleEvents

AppleScript and AppleEvents are the Macintosh's automation system. The Photoshop 4.0 scripting system is based heavily on the programming architecture defined by Apple. Most users think of AppleScript and AppleEvents from the user perspective: the Macintosh script editor, firing off events to different applications to automate procedures. What I'll be describing here is the internal workings necessary to define events to an external system. In this case, the plug-ins, such as Dissolve, must take on extra descriptors that make their parameters available to the host, in this case, Adobe Photoshop 4.0. The terminology resource is the first internal description system that bridges the gap between the plug-ins programming parameters and the external automation system.

Note that the Photoshop 4.0 automation system, while designed around the AppleScript/AppleEvent model, has been created to integrate fully with OLE Automation on Windows. More information on that is available in the appendix of the Photoshop SDK Guide.

Start with the examples

The terminology resource is a standard AppleScript/AppleEvent `'aete'` resource. The terminology resource is a bit cumbersome, so I always recommend starting with the example code. In this case, I had to make it from scratch. First, I chose to define some common parameters that would change from plug-in to plug-in:

```
#define vendorName     "AdobeSDK"             // Unique vendor name
#define ourSuiteID     'sdK1'                 // Must follow id guidelines
#define ourClassID     ourSuiteID             // Must be unique, but can be suite id
#define ourEventID     'disS'                 // Must follow id guidelines
#define ResourceID     16000                  // Typical id for plug-ins
#define uniqueString   ""                     // Empty
```

Then, I created the terminology resource:

```
resource 'aete' (ResourceID, purgeable)
{// Aete version and language specifiers:
  1, 0, english, roman,
 {// Vendor suite name:
   vendorName,                                  // "AdobeSDK"'
   "Adobe example plug-ins",                    // Optional description
   ourSuiteID,                                  // Suite id 'sdK1'
   /* This is extremely important. All IDs, keys, and names
      must be unique. The SDK describes a naming convention that
      must be followed explicitly. Your scripting keys and IDs
      (unsigned32) must always follow these rules:
      1. They must start with a lowercase letter.
      2. They must contain at least one uppercase letter.
      3. They cannot be all lowercase.
```

```
    4. They cannot be all uppercase.
    More below when we get to keys. */
1,                                          // Suite code, must be 1
1,                                          // Suite level, must be 1
{// Structure for filters. Unique filter name:
 vendorName " dissolve",                    // "AdobeSDK Dissolve"
 "dissolve noise filter",                   // Optional description
 ourClassID,
 // Class id must be unique or suite id. Suite id 'sdK1'.
 ourEventID,                                // Unique event id 'disS'

 NO_REPLY,                                  // Never a reply
 IMAGE_DIRECT_PARAMETER,
 // Direct parameter. See PIActions.h for other macros.
  { // Parameters:
   "amount",                                // Parameter name
```

```
/* Must be predefined parameter name and key from
   PIActions.h or unique name and key id. See
   'disposition' for example. */
keyAmount,                                // Parameter key
/* Must be predefined parameter key from PIActions.h or unique key id. */
typeFloat,                                // Parameter type
// TypeInteger, typeBoolean, typeText, etc., all defined in PIActions.h
"dissolve amount",                        // Optional description
flagsSingleParameter,                     // Parameter flags
// Other parameters in PIActions.h

// Second parameter:
vendorName " disposition",
// Unique name "AdobeSDK disposition"
keyDisposition,                           // Unique key 'disP'
typeMood,                                 // Unique type 'mooD'
```

```
    "dissolve disposition",                // Optional description
    flagsEnumeratedParameter               // Parameter flags for enum

    vendorName " entire image",
    // Unique name "AdobeSDK entire image"
    keyEntireImage,                        // Unique key 'entI'
    typeBoolean,
    flagsSingleParameter
   } // Close parameters
  }, // Close filter structure
  { }, /* Plug-in classes for all other plug-ins here
     (we'll use this later) */
  { }, // Comparison ops (not supported)
  { // Any enumerations. We have one, typeMood:
   typeMood,                               // Unique type 'mooD'
   {
```

```
vendorName " clear",
// Unique name "AdobeSDK clear"
dispositionClear,                       // Unique key 'moD0'
"clear headed",                         // Optional description

vendorName " cool",
// Unique name "AdobeSDK cool"
dispositionCool,                        // Unique key 'moD1'
"got the blues",                        // Optional description

vendorName " hot",
// Unique name "AdobeSDK hot"
dispositionHot,                         // Unique key 'moD2'
"red-faced",                            // Optional description
```

```
         vendorName " sick",
         // Unique name "AdobeSDK sick"
         dispositionSick,                         // Unique key 'moD3'
         "green with envy"                        // Optional description
        } // Close typeMood
       } // Close enumerations
      } // Close vendor suite
    }; // Close 'aete'
```

The terminology resource is parsed on the Macintosh side by a standard template included with most compilers. On the Windows side, it is precompiled along with the **'PiPL'** resource and then parsed by the Photoshop resource file converter, `CNVTPIPL.EXE`. Either way, the `Dissolve.r` file is converted into a working resource that is used at runtime by the host.

Add the HasTerminology resource to your PiPL

Once I had a complete terminology resource, I had to tell Photoshop where to find it, since a single plug-in file can have multiple modules in it. To do this, I added a new **PiPL** type, *HasTerminology*.

Its syntax is:

```
HasTerminology { ourClassID, ourEventID, ResourceID, uniqueString }
```

Just to review, I defined the following parameters:

```
#define vendorName     "AdobeSDK"              // Unique vendor name
#define ourSuiteID     'sdK1'                  // Must follow id guidelines
#define ourClassID     ourSuiteID              // Must be unique, but can be suite id
#define ourEventID     'disS'                  // Must follow id guidelines
#define ResourceID     16000                   // Typical id for plug-ins
#define uniqueString   ""                      // Empty
```

The AppleScript and AppleEvent architecture makes all key and name dictionaries global, which is why unique key/name pairs are required. A predefined dictionary of common terms is defined in `PIActions.h`. You can use those keys and their obvious names (`keyColor`, name “Color”) instead of having to create unique key and name pairs. I recommend using the standard keys whenever you possibly can.

If you define a **uniqueString**, then your plug-in will stay scoped only to Photoshop and you will not have to worry about having globally unique names. But you still have to worry about conflicting with your other suites using that same **uniqueString**. For example, I didn't have to use key names such as "AdobeSDK disposition"—I could have just used "disposition." But I chose to keep everything scoped globally for future AppleScript/AppleEvent compatibility.

Creating a scripting recording function

The next step for Dissolve was to record my parameters. There are a number of utility routines defined in `PIUtilities.h` and `PIUtilities.c` to make reading and writing from descriptors easier than having to access the procedures directly through the callback structure. You cannot check a scripting playback function, nor whether a terminology resource is correct, until some parameters are handed to Photoshop.

To use globals or not to use globals, that is the question!

For versions of Photoshop prior to 4.0, the only way to track global variables was for you to allocate the memory yourself and store the global values in a parameter handle that was handed back to the plug-in on subsequent iterations.

The Photoshop 4.0 scripting system will always pass your plug-in a *descriptor* at every selector call. A descriptor is a set of keys and values, very much like a set of predefined global values. Theoretically, I could use the scripting system to track my global values, instead of passing my entire global struct to my different routines and storing it in the parameter handle.

To make that change, I'd have to take out all my global variables and change to reading and storing my parameters in the scripting descriptor on every selector call. That's a lot of work, and I didn't feel I would gain anything from that.

Instead, I decided to stay with my global variables, and use the scripting system to write out my final values and read in values to override my initial global values. This made much more sense, and allows the plug-ins to operate in a non-scripting environment, such as older versions of Photoshop.

WriteScriptParams routine

I created a routine, `WriteScriptParams`, that took the global values and created a descriptor to hand back to the host.

I created a new source file, `DissolveWithScripting.c`, to hold the playback and recording script functions.

```
OSErr WriteScriptParams (GPtr globals)
{
  double            percent = gPercent;
  /* I'm using a double because I want to use scripting type UnitFloat with unitPercent,
  which is a double value. By using UnitFloat, my value will display in the actions palette
  with a percent sign after it. Cool! */
  PIWriteDescriptortoken = nil;
  OSErr             gotErr = noErr;

  if (DescriptorAvailable( ))
  {
   /* DescriptorAvailable( ) is a macro from PIUtilities that checks to see if the
     gStuff->descriptorParameters callback parameter block is available. */

     token = OpenWriter( );
   // OpenWriter( ) is a macro from PIUtilities that creates a new write descriptor.
```

```
if (token)
 { // We got a valid token to work with. Write our keys:
  PIPutUnitFloat(token, keyAmount, unitPercent, &percent);
  /* This is a macro from PIUtilities. It requires the token to write to, the key, the
   unit (unitPercent, unitDistance, unitPixels, etc., defined in PIActions.h), and then
   a pointer to the double. */

  PIPutEnum(token, keyDisposition, typeMood, gDisposition);
   /* Another macro from PIUtilities. This writes an enumeration. It takes the token,
   the key, the list of enumerations (the type) and the actual enumeration. gDisposition
   is an unsigned32 that is either dispositionClear, dispositionCool, dispositionHot, or
   dispositionSick. Note that if these weren't defined in the terminology resource, it
   would display nothing, or garbage. The enum stored must match the keys in the enumera-
   tion list in the 'aete'.*/
```

```
if (gIgnoreSelection)
  PIPutBool(token, keyEntireImage, gIgnoreSelection);
  /* Like I suggested, when you are writing boolean values, it makes the actions
  palette look cleaner if you only write them when they are in their non-default value.
  In this case, when gIgnoreSelection is true (the default is to use the selection)
  then the macro from PIUtilities writes the key and boolean value to the descriptor
  in token. */

 gotErr = CloseWriter(&token);
  /* This is a very useful routine defined in PIUtilities. When you close a token,
  it returns with a handle to a descriptor. This descriptor is then what you pass to
  the host for it to display in the actions palette (and subsequently return to you
  on playback.) CloseWriter closes the token and stores the descriptor in the
  gStuff->descriptorParameters callback parameter block, which is how a plug-in
  hands back a descriptor. It then deallocates token and sets it to NULL. Lastly, it
  sets the recordInfo parameter to dialogOptional, which is the standard return value
```

```
        to tell the host "Only pop my dialog when the user wants it." For a description of
        recordInfo, see the Scripting chapter of the SDK and PIUtilities.*/
        } // Close token
    } // Close DescriptorAvailable
    return gotErr;
} // End WriteScriptParams
```

Calling WriteScriptParams

I call `WriteScriptParams` in `DoFinish`, as that's the last routine the plug-in executes before it completely returns to the host.

Running the plug-in and errors in scripting

Once I completed my `WriteScriptParams` routine, it was time to try it out to see if the terminology resource, `HasTerminology` PiPL property, and `WriteScriptParams` routine worked. I did this by placing an alias to the plug-in in the Photoshop plug-ins directory, deleting my preferences file (to start fresh) and running Photoshop.

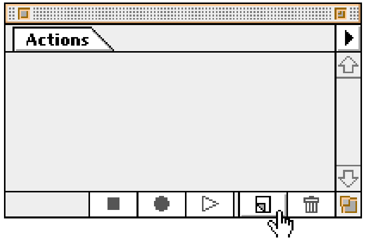


Figure 3 Creating a new action in the actions palette

I then opened a document and clicked the "document" icon in the actions palette, which is the "New Action" button. I named it, and then went to my plug-in and executed it with some basic parameters. Finally, I clicked the "stop" button in the actions palette, and checked to see if my plug-in had been recorded.

Here is a list of issues and answers I found in debugging from this step:

My plug-in wasn't in the filters menu.
This happened because I didn't put the plug-in in the right directory, or because Adobe Photoshop was loading plug-ins from the preferences file (and not scanning the directory to look for new plug-ins), or becasue my PiPL resource wasn't valid.

My plug-in didn't get recorded.
This was usually because I wasn't handing back a proper descriptor. I was either handing back `NULL`, accidentally, or I was storing garbage data in the descriptor, which was messing everything up.

The actions palette says my plug-in's name, but none of its parameters (such as "Using: Dissolve" but nothing else)
This means the scripting system did not find a valid `'aete'` dictionary resource, and/or it did not find a valid reference to the resource in the `HasTerminology` property. It's usually either a bad reference number in the `HasTerminology` property, a bad construction of the `HasTerminology` property, or a badly formed dictionary resource. On the Macintosh side, the resource compiler will complain if the dictionary resource of `Dissolve.r` is not formed properly. On the Windows side, `CNVTPIPL.EXE` will complain. Unfortunately, neither will complain if the keys and data you hand back in your descriptor do not match the keys in your dictionary resource. It just won't display.

The actions palette displays labels with no data after them, such as "Amount: %"
This was due to a messed up descriptor. I was either handing back invalid (or improper) data (such as mixing up my keys and data types) or I was handing back no descriptor (accidentally handing back `NULL`, for instance.)

The actions palette displays labels with scrambled data

This happened when I had different keys in my dictionary than I was storing in my descriptor, if I had a `typeInteger` for `keyAmount` but then stored using `typeFloat`, or if I was storing `typeText` and passed binary instead of alphanumeric information in the handle.

Actions palette with Dissolve action

Figure 4 shows the actions palette once I got the proper descriptor recorded, along with good dictionary and `HasTerminology` resources.

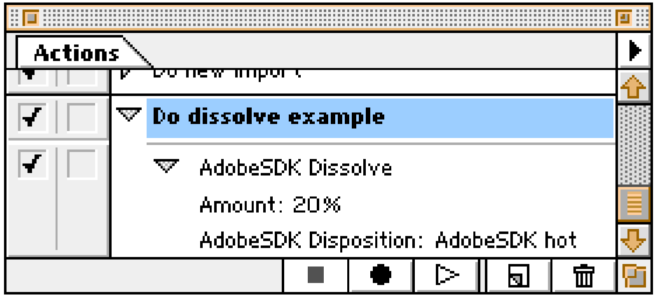


Figure 4 Dissolve filter actions palette display

Automating the plug-in for playback

Now that the plug-in was correctly recording and displaying a descriptor, it was time to prepare it to read that descriptor when it was handed to me, and honor those parameters.

Taking the same approach to globals as the `WriteScriptParams` routine, I created a `ReadScriptParams` routine, with the purpose of opening, pulling keys and values out of a descriptor, and overriding the global variables.

```
Boolean ReadScriptParams (GPtr globals)
{
 double               x = 0;
 const double         minValue = kPercentMin, maxValue = kPercentMax;
 // Used to pass minimum and maximum values for PinUnitFloat
 unsigned long        percentUnitPass = unitPercent;
 // Used to pass unitPercent to PinUnitFloat
```

```
PIReadDescriptor      token = NULL;
DescriptorKeyID       key = NULLID;
DescriptorTypeID      type = NULLID;
int32                 flags = 0;
DescriptorKeyIDArray                array = { keyAmount, keyDisposition, NULLID };
/* This array will be checked off as each key is read. It should return { keyNULL,
keyNULL, NULL }. If it doesn't, then we've missed a key somewhere.
See errMissingParameter, below. */

OSErr                 stickyError = noErr;
Boolean               returnValue = true;
// ReadScriptParams returns with whether to pop the dialog or not (true = show dialog)

if (DescriptorAvailable( ))
{ // If descriptorParameters callback suite is available, do this:
```

```
 token = OpenReader(array);
 /* Routine from PIUtilities. Opens the descriptor pointed to in
gStuff->descriptorParameters, starts tracking keys in array, and returns a read token
to work with. */
 if (token)
 { // Got a valid read token. Now start grabbing keys until we get NULL:
  while (PIGetKey(token, &key, &type, &flags))
  { // We got a valid (non-NULL) key. See which value it is:
   switch (key)
   { // We can receive these keys in any order, so check to see which one:
    case keyAmount:
     PIGetPinUnitFloat(token, &minValue, &maxValue, &percentUnitPass, &x);
     /* This is a routine from PIUtilities. It gets a unit-delimited value (such as
    unitPixels, unitPercent) and automatically pins it between minValue and maxValue.
    The value is returned in the last parameter, which is the address of a double
    (in this case, "x"). If the value had to be coerced (pinned to the low or high
```

```
  number) then this routine will return the coercedParam error, but "x" will still
  be a valid number. */
   gPercent = x; // Assign to our global
   break;

 case keyDisposition:
   PIGetEnum(token, &gDisposition);
  /* This is another routine from PIUtilities. It reads an enumerated value. Since
  our global is an unsigned32, we can have PIGetEnum store the value directly to the
  global. */
  break;

 case keyEntireImage:
   PIGetBool(token, &gIgnoreSelection);
   /* From PIUtilities, returns a boolean value. Since our global is a boolean,
  we pass its address and have it set directly. */
   break;
```

```
      // Ignore all other cases and classes
     }
   }

   stickyError = CloseReader(&token);
  /* CloseReader, from PIUtilities, automatically closes the read token, deallocates it,
   and stores NULL in token. It returns an error code, indicating if any errors were
   encountered during the getKey routine.

  if (stickyError)
  {
    if (stickyError == errMissingParameter)
     { } /* errMissingParameter = -1715, which means one of the keys in
         descriptorKeyIDArray was not found. Walk the array, and whatever is not
         "typeNull" is the value not found in the descriptor. For this example, I can go
         with the default values if I missed a key. If you cannot, or cannot coerce a
```

```
          value from the keys you did receive, then you might want to show your dialog.
          Whether or not you can show your dialog depends on PlayDialog( ). See below. */
      else
        gResult = stickyError; // We got a real error. Report it.
      } // Close stickyError
    } // Close token
    gQueryForParameters = returnValue = PlayDialog( );
    /* PlayDialog( ) examines playInfo inside gStuff->descriptorParameters and returns true
     if it is plugInDialogDisplay, which means "please display your dialog." If it is
     plugInDialogSilent, you must never show your dialog, and if it is
     plugInDialogDontDisplay, then don't display your dialog unless you need to.
     (Such as if you missed a key you need and cannot coerce.) */
  } // Close descriptorAvailable
  return returnValue; /* The global variable gQueryForParameters determines whether I need
  to pop my dialog, but I'll return this value, as well. */
} // End ReadScriptParams
```

Calling ReadScriptParams and ValidateParameters

Calling `ReadScriptParams` is a little trickier. I want to call it after I've initialized my globals, but before I need them. Sometimes, however, my plug-in may be called and I may never get to the `DoParameters` routine, which initializes my globals. This happens in Adobe Premiere® which only executes the plug-in completely once, then passes its parameters in for every frame of a filmstrip. This also can occur when a plug-in has been recorded, then the user quits Photoshop, runs it again, and executes the action right from the palette. Literally, I may go to store values in my globals before I've allocated space for them. Because of this danger, I decided to pull some of the initialization routines out of `DoParameters` and create an additional routine, `ValidateParameters`, which checks to see if the parameters are valid, and if not, initializes them. That way I can call it right at the beginning of my `DoStart` routine, right before I dispatch to my user interface and code which depends on my globals.

Anywhere before `DoStart` that I might use my globals, I need to check them for validity first. That could be in `DoParameters`, `DoPrepare`, or `DoStart`:

```
void DoParameters (GPtr globals)
{ /* Called on selectorParameters. We may not always get here on our first iteration (for
  instance, if a user created an action calling this plug-in, quit Photoshop, then ran
  Photoshop again and immediately executed the action). */
```

```
 ValidateParameters (globals); // Check for valid parameters

 gQueryForParameters = TRUE;
 // If we're here, that means we're being called for the first time.
}
```

Now `ValidateParameters` does most of the work of `DoParameters`. This allows me to call it from multiple routines, to make sure my globals are valid and at least have default values before I use them:

```
void ValidateParameters (GPtr globals)
{ // Called whenever parameters need to be validated before used:
 if (!gStuff->parameters)
 { // Oops. Parameters haven't been allocated yet. Do that now.
   gStuff->parameters = NewHandle ((long) sizeof (TParameters));

   if (!gStuff->parameters)
   { // Couldn't do it. Must be out of memory.
    gResult = memFullErr;
```

```
      return;
    }
    // Assign default global values:
    gPercent = 50;
    gDisposition = dispositionCool;
    gIgnoreSelection = false;
    gUseAdvance = false;
    gRowSkip = 1;
  } // Close gStuff->parameters
}
```

My `DoPrepare` routine does access some global variables, so I had to include a call to `ValidateParameters` before I used `gRowSkip`:

```
void DoPrepare (GPtr globals)
{ // Called on selectorPrepare to allocate memory requirements
  short                    rowWidth = 0;
  short                    total = 0;
```

```
 long                   oneRow = 0;
 long                   inOutRow = 0;
 long                   inOutAndMask = 0;

 gStuff->bufferSpace = 0;

// Check maxSpace to determine if we can process more than a row at a time

ValidateParameters (globals);
// Check to make sure gRowSkip has been initialized BEFORE we use it!

 total = gStuff->filterRect.bottom - gStuff->filterRect.top;
 rowWidth = gStuff->filterRect.right - gStuff->filterRect.left;

 oneRow = rowWidth * (gStuff->planes);
// One row of data and its planes
```

```
  inOutRow = oneRow * 2; // inData, outData
  inOutAndMask = inOutRow + rowWidth;
 // MaskData is only one plane (alpha)

  while (((inOutAndMask * gRowSkip) < gStuff->maxSpace) &&
      (gRowSkip < total))
     gRowSkip++;

 gStuff->maxSpace = gRowSkip * inOutAndMask; // All we need
}
```

Finally, right at the top of `DoStart`, I make a call to `ValidateParameters` to make sure, before I use my globals, that they've been at least assigned default values. Then I call `ReadScriptParams` to read the keys from the descriptor, if there is one, and override the default global values with the script parameters.

```
void DoStart (GPtr globals)
{ // Called from selectorStart. Main routine.
  ValidateParameters (globals);
  /* If stuff hasn't been initialized that we need, do it, then go check if we've got
  scripting commands to override our settings */

  ReadScriptParams (globals);
  // Update our parameters with the scripting parameters, if available

  if (gQueryForParameters)
  { /* We got either plugInDialogDisplay or this is the first time the user has selected
    the plug-in (so I have to pop the dialog to get the initial values) */
     PromptUserForInput (globals);       // Show the UI
     gQueryForParameters = FALSE;
  }

 // Rest of DoStart here.
```

Playback and recording questions: How do I know when…?

The obvious questions I had were:

"How do I know when I'm being played back?"
"How do I know when I'm being recorded?"
"How do I know when the user has selected me from the menu?"
"How do I know when the user has selected me in the actions palette?"

The answer to all of these is "*You don't.*"

A plug-in has no way of knowing whether it's being recorded, played back, or directly interacted with by the user. This decision was made in the scripting implementation to make it as seamless with the original interface as possible. As long as you honor the `playInfo` flag, you will always know whether to pop your dialog or not. This includes if the user has clicked the *Dialog On* icon in the actions palette and is playing back your plug-in, or the user has selected your plug-in directly from the menu.

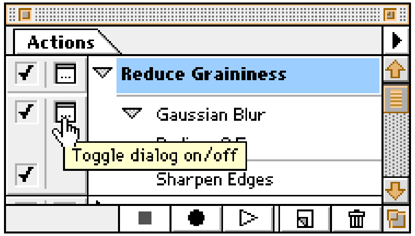


Figure 5 Toggle dialog option in actions palette

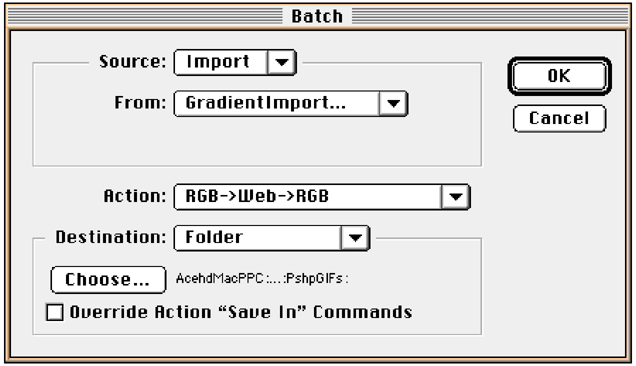
Whether the dialog has been requested or not, it makes sense to override any globals with any scripting keys provided before deciding to display the dialog — that way, the user can double-click to re-record an action and your plug-in will pop its dialog with the scripting parameters handed to it. Don't make the mistake (like I did, originally) of ignoring the scripting parameters just because `plugInDialogDisplay` has been requested. If it has been requested from within an action, like Figure 5, the user will expect to see the parameters from the actions palette in the plug-in's dialog.

Now that we're deep in the pool of scripting and you've gone through the simple example of the Dissolve filter plug-in, let's step up the complexity and look at an Import module. In my case, it was the *DummyScan* example from the 3.0.5 SDK, which I renamed *GradientImport*, which was more in sync with what it did.

GradientImport import plug-in module

So you thought the Dissolve example was torture enough? Oh no, things get much more fun when you try to apply scripting to a module that can be controlled in a *batch*. *Batch importing* is an additional method for processing numerous images at a time. This is in addition to the old *multiple acquire* mechanism that is part of the import module interface.

The batch command is available from the pull-down menu attached to the actions palette.



Source: Import
From: GradientImport…
Action: RGB->Web->RGB
Destination: Folder
AcehdMacPPC:…:PshpGIFs:
☐ Override Action "Save In" Commands

Figure 6 Batch dialog

With so many options, there are several approaches to updating an Import module:

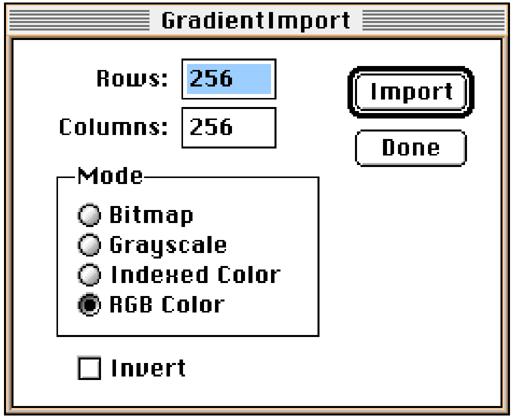
1. Leave it alone. The scripting system will automatically call the import module for each import in a batch. Even vanilla plug-ins can be called by the scripting system. Your dialog will be popped for every iteration, which may not be desirable.
2. If it is a single import module, meaning it only returns one image at a time, you can update it for scripting and record all the parameters necessary for that single import. The batch mechanism will pass your parameters to your plug-in automatically.
3. If it is a multiple acquire module, that means that all control for opening multiple images happens within your plug-in. You can: a) maintain detailed control over the iterative imports and use the scripting system to call your plug-in with some default parameters, such as preferences, and/or b) record every iterative import as another scripting event.

The GradientImport module uses the older multiple acquire mechanism. To showcase the most robust scripting setup, I chose the last option, 3b, and decided to make the plug-in record every event of its multiple acquire. That way a user can blast off a single action and have multiple images open. This makes the most sense for digital cameras that cache a set of images and let the user import and color correct multiple images.

Creating the GradientImport terminology resource

Assessing the user interface

The first thing I did was examine the user interface dialog to determine what parameters to represent in the terminology resource.



Rows: 256
Columns: 256
Mode
( ) Bitmap
( ) Grayscale
( ) Indexed Color
[✓ RGB Color]
☐ Invert

Figure 7 GradientImport user interface

The items were:

1. An “OK” button (“Import”) which does not need to be recordable.
2. A “Cancel” button (“Done”) which does not need to be recordable.
3. An integer from 1 to 30,000 representing Rows.
4. An integer from 1 to 30,000 representing Columns.
5. A mutually exclusive enumeration, “Mode”, representing “Bitmap”, “Grayscale”, “Indexed Color”, or “RGB Color”.
6. A boolean, “Invert”.

Below is the terminology resource I used for GradientImport.

GradientImport terminology resource

```
resource 'aete' (ResourceID, purgeable)
{ // Aete version and language specifiers:
  1, 0, english, roman,
```

```
{ // Vendor suite name:
   vendorName,                             // "AdobeSDK"'
   "Adobe example plug-ins",               // Optional description
   ourSuiteID,                             // Suite id 'sdK3'
   1,                                      // Suite code, must be 1
   1,                                      // Suite level, must be 1
   { }, // Structure for filters
   { // Structure for all other plug-in types:
     vendorName " GradientImport",         // "AdobeSDK GradientImport"
     "gradientImport multiple import",     // Optional description
     { // Properties:
        "<Inheritance>",
        /* All non-filters inherit from a base class of the same name as their plug-in
        type, such as classFormat, classExport, etc. See PIActions.h. Inheritance must
        be the first property entry, even if there are no others. */
```

```
        keyInherits,                    // Always
        classImport,                    // Either classExport, classFormat, etc.
        "parent class import",          // Optional description
        flagsSingleProperty,            // Parameter flags

        // Second property:
        "multi-import",                 // Property name
        keyMultiImportInfo,             // Unique key 'mulK'
        classMultiImportStruct,         // Unique class 'mulS'
        "multiple import info",         // Optional description
        flagsListProperty               // Flags for a list
    }, // Close properties
    { }, // Elements (not supported)
    /* Normally you won't need to create other classes, but since I'm going to be
    storing a list of "import information" (the values needed to create one image),
    I'm creating a class with the set of information, called "import info": */
```

```
"import info",                       // Unique class name
classMultiImportStruct,              // Unique class 'mulS'
"class import info",                 // Optional description
{ // Import info class properties:
   "rows",                           // Property name
   keyRows,                          // Standard key keyHorizontal
   typeFloat,                        // Property type
   "number of rows",                 // Optional description
   flagsSingleProperty,              // Flags for property

   "columns",                        // Property name
   keyColumns,                       // Standard key
   keyVertical
   typeFloat,                        // Property type
   "number of columns",              // Optional description
   flagsSingleProperty,              // Flags for property
```

```
      "mode",                             // Property name
      keyOurMode,                         // Standard key 'keyMode'
      typeGradientMode,                   // Unique type 'grmT'
      "color mode",                       // Optional description
      flagsEnumeratedProperty,            // Flags for property

      "invert",                           // Property name
      keyInvert,                          // Unique key 'invR'
      typeBoolean,                        // Property type
      "invert image",                     // Optional description
      flagsSingleProperty                 // Flags for property
    } // Close class import info
    { }, // Elements (not supported)
  } // Close non-filter classes
  { }, // Comparison operators (not supported)
  { // Any enumerations go here. We have one, typeGradientMode:
    typeGradientMode,                     // Unique type 'grmT'
```

```
  { // Enumeration listing:
     "bitmap",                            // Property name
     ourBitmapMode,                       // Unique key 'bitM'
     "bitmap mode",                       // Optional description

     "grayscale",                         // Property name
     ourGrayscaleMode,                    // Unique key 'gryS'
     "grayscale mode",                    // Optional description

     "indexed color",                     // Property name
     ourIndexedColorMode,                 // Unique key 'indX'
     "indexed color mode",                // Optional description

     "rgb color",                         // Property name
     ourRGBColorMode,                     // Unique key 'rgbC'
     "rgb colormode",                     // Optional description
  }, // Close typeGradientMode
} // Close enumerations
```

```
  } // Close vendor suite
}; // Close 'aete'
```

After the terminology resource was done, I added the `HasTerminology` to the **PiPL**.

GradientImport HasTerminology PiPL property

```
HasTerminology { ourClassID, ourEventID, ResourceID, uniqueString }
```

With:

```
#define vendorName     "AdobeSDK"          // Unique vendor name
#define ourSuiteID     'sdK3'              // Must follow id guidelines
#define ourClassID     'graD'              // Must be unique, but can be suite id
#define ourEventID     typeNull
/* Must be typeNull or the host will think it's a filter (event) instead of an import,
 export, format, or selection (class) */
#define ResourceID     16000               // Typical id for plug-ins
#define uniqueString   ""                  // Empty
```

Writing scripting parameters in GradientImport

The next step was to create the routine to pass the scripting parameters back out to Photoshop. Taking the same approach as with the Dissolve example, I used my globals to pass their values across my different functions, then, at the last minute, I passed the list of events back encapsulated in a descriptor.

Due to the nature of the multiple acquire mechanism, I needed a way to track the multiple imports that would occur and then hand them back to the scripting system. I decided to do this by creating an actual descriptor for each import, then storing all the descriptors inside an encapsulating descriptor to hand back to the host at the very end of execution. This took the form of:

1. In `DoFinish`, create a descriptor and store it in a static array with a maximum of `kMaxDescriptors` (in this case, 50) via `CreateDescriptor( )`.
2. In `DoFinish`, if multiple acquiring was not available, write the descriptor out to the host in final form via `CheckAndWriteScriptParams( )`.
3. In `DoFinalize`, write the descriptor out to the host in final form via `CheckAndWriteScriptParams( )`.

So, DoFinish looked like this:

```
void DoFinish (GPtr globals)
{
  gStuff->acquireAgain = gContinueImport;
  // gContinueImport tracks whether to continue importing

  // Now create a descriptor and store it in our static array for saving later:
  CreateDescriptor(globals); // Creates and stores descriptor in next open gArray

  // If we can't finalize, then we'll have to write our parameters now:
  if (!gStuff->canFinalize)
     CheckAndWriteScriptParams(globals); // Writes script params
}
```

And DoFinalize:

```
void CreateDescriptor (GPtr globals)
{
  PIType                mode = GetGradientMode(gLastMode);
  // Converts a global enumeration to the actual unsigned32 mode

  const double          rows = gLastRows, columns = gLastCols;
  // Converting globals to doubles for PutUnitFloat to use unitPixels value

  Boolean               invert = gLastInvert;
  PIWriteDescriptor     token = NULL;
  PIDescriptorHandle    h;
  OSErr                 stickyError = noErr;

  if (DescriptorAvailable( ))
  { // PIUtilities routine to check for descriptorParameters callbacks succeeded.
     token = OpenWriter( ); // Open new write descriptor
     if (token)
     { // Got the descriptor. Go ahead and write the keys into it:
```

```
PIPutUnitFloat(token, keyRows, unitPixels, &rows);
// Puts our rows as pixels

PIPutUnitFloat(token, keyColumns, unitPixels, &columns);
// Puts our columns as pixels

PIPutEnum(token, keyOurMode, typeGradientMode, mode);
// Puts the exact enumeration (must match terminology resource!)

if (invert) PIPutBool(token, keyInvert, invert);
// Again, only if non-default (true), writes "with invert"

stickyError = CloseWriteDesc(token, &h);
/* Have to call PIUtilities CloseWriteDesc, which closes a specific token, and
```

```
returns a descriptor handle in "h". If I called CloseWriter, it would close it and
automatically store it in gStuff->descriptorParameters, which I don't want, since
I'm trying to create a static array of descriptors before passing them to the host. */
token = NULL; // Just in case

if (!stickyError)
{ // As long as we didn't have an error writing:
   if (gLastImages >= kMaxDescriptors)
   { // Oops, went over our limit. Delete the last and replace it:
     gLastImages—; // Just keep replacing last one
     PIDisposeHandle(gArray[gLastImages]);
     // Dispose last handle
   }

   gArray [gLastImages++] = h; // Stick handle on array

   gArray [gLastImages] = h = NULL; // NULL out end, just in case
} // Close stickyError
```

```
        } // Close token
    } // Close descriptorAvailable
} // End createDescriptor
```

The `CheckAndWriteScriptParams` routine checks for any data then calls the `WriteScriptParams` routine:

```
OSErr CheckAndWriteScriptParams (GPtr globals)
{
  OSErr            gotErr = noErr;

  if (gLastImages) gotErr = WriteScriptParams(globals);
  // If we have done at least one import (gLastImages > 0), write our scripting parameters
  else gotErr = gResult = userCanceledErr;
  /* Else error out of entire loop (if we don't do this, we might end up with a single
  recorded parameter, "Import using: GradientImport" which looks ugly. */
  return gotErr;
}

OSErr WriteScriptParams (GPtr globals)
{
```

```
unsigned32          count = gLastImages;
PIWriteDescriptor   token = NULL;
OSErr               stickyError = noErr;

if (DescriptorAvailable( ))
{ // gStuff->descriptorParameters callbacks available.
   token = OpenWriter( ); // open write descriptor
   if (token)
   { // Got our token. Write our keys.
     PIPutCount(token, keyMultiImportCount, count);
     /* A list is always preceded by its count. Note the count, and the following keys,
     are stored a keyMultiImportCount for the entire list. */

     for (count = 0; count < gLastImages; count++)
     { // Iterate through local array:
        PIPutObj(token, keyMultiImportInfo,
        classMultiImportStruct, &gArray [count]);
```

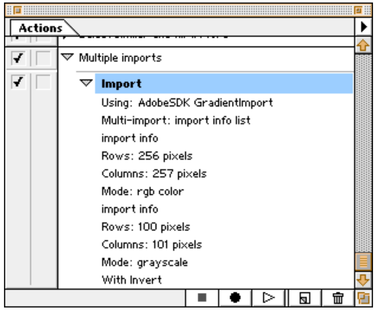
```
                /* PIPutObj, from PIUtilities, automatically disposes the handle and sets it to
                NULL. */
            }

            gLastImages = 0; // Reset
            stickyError = CloseWriter(&token);
            /* Closes descriptor, stores it in gStuff->descriptorParameters, sets
            plugInDialogOptional, and sets token to NULL. */
        } // Close token
    } // Close descriptorAvailable
    return stickyError;
} // End WriteScriptParams
```

Testing the multiple import routine

Now that the write routines were done, I was able to test the multiple import routines. I turned recording on in the actions palette and imported a couple of images, one after the other, then dismissed the GradientImport dialog. Figure 8 shows the resulting display in the actions palette.

Figure 8 GradientImport display in the actions palette



```
Multiple imports
  Import
    Using: AdobeSDK GradientImport
    Multi-import: import info list
    import info
    Rows: 256 pixels
    Columns: 257 pixels
    Mode: rgb color
    import info
    Rows: 100 pixels
    Columns: 101 pixels
    Mode: grayscale
    With Invert
```

Note how the multiple import list is presented: as its label, “Multi-import”, with its type label, “import info” and “list” after it. Then each individual item of the list is headed with the type label “import info”. The first image is a 256x257 RGB image; the second image is a 100x101 grayscale inverted image. Again, I only display a boolean when it’s in its non-default (“with invert” only, as opposed to “without invert” and “with invert”). Another nice feature is the display of the word “pixels” after the

"Rows" and "Columns" entries. This is thanks to `PutUnitFloat` and `unitPixels`.

Playback of scripting parameters for GradientImport

Now that I had GradientImport correctly recording parameters, it was time to modify it to read back parameters. This, too, is complicated, because it requires reading from a list and dispatch parameters through the multiple acquire loop, iterating through the list. I decided to break it out into this logic:

1. At `DoPrepare`, open any descriptor handed to me by the host and see if there was a list in there, via `OpenScriptParams`.
2. At `DoStart`, read the next descriptor object in the list via `ReadScriptParams` and assign all its keys to globals via `SwitchScriptInfo`.
3. In `DoStart`, as soon as the dialog is asked for, or if there is an error, we no longer need to iterate through the list. Close it via `CloseScriptParams` and continue to create our own array to pass back later.

```
void DoPrepare (GPtr globals)
{
```

```
    gStuff->maxData = 0;

    if (!WarnBufferProcsAvailable ( ))
       gResult = userCanceledErr; // Exit. Already displayed alert.

    // If finalization is available, we will want it:
    gStuff->wantFinalize = true;

    ValidateParameters (globals);
    /* This should look familiar. Same functionality, but instead, checks variables
    pertinent to GradientImport for default values and allocation, if needed. */

    // Now see if the scripting system has passed us anything:
    OpenScriptParams (globals);
  }

  void DoStart (GPtr globals)
  {
```

```
int16 j = 0; // Used later

// Insist on having the buffer procs:
if (!WarnBufferProcsAvailable ( ))
{
   gResult = userCanceledErr; // Should probably display err
   return;
}

// Assume we won't be coming back around for another pass unless explicitly set:
gStuff->acquireAgain = gContinueImport = false;

// Validate our globals then override them with scripting parameters, if available:
ValidateParameters (globals);
ReadScriptParams (globals);

if (gQueryForParameters)
{ // Open our dialog. If it's already up, this returns with no err:
```

```
if (!OpenOurDialog (globals))
{ // Couldn't open our dialog. Abort! Abort!
  gQueryForParameters = false;
  CloseScriptParams(globals); // Close up the open descriptor!
  gResult = memFullErr; // Return with memory full error
  return;
}

// So far so good. Now dispatch our dialog routines:
if (!RunOurDialog (globals))
{ // User canceled. Close everything up.
  gQueryForParameters = false;
  CloseOurDialog (globals); // Deallocates dialog
  CloseScriptParams(globals); // Closes open descriptor
  gResult = userCanceledErr; // Exit without err
  return;
```

```
        // Rest of DoStart here.
```

With `DoPrepare` and `DoStart` set up, there were four routines to be created. `OpenScriptParams`, to open the descriptor; `ReadScriptParams`, to read the next object in our list; `SwitchScriptInfo`, which reads keys from the object and overrides the global values; and `CloseScriptParams`, to close and tidy up the open descriptor handed to the plug-in from Photoshop.

`OpenScriptParams` was one of the easier ones, as all it had to do was watch for the count key and find it in the descriptor handed in by the host:

```
void OpenScriptParams (GPtr globals)
{
  DescriptorKeyID     key = 0;
  DescriptorTypeID    type = 0;
  int16               loop = 0;
  int32               flags = 0;
  Boolean             leaveEarly = false;
```

```
if (DescriptorAvailable( ))
{ // Descriptor procs available. Now open the descriptor:
   gToken = OpenReader(NULL);
   /* Normally would pass an array indicating the expected keys. Problem is I don't
   know how many items are in the list until I open it. Therefore, I'm passing NULL to
   indicate to the scripting system not to bother with a key array list. */
   if (gToken)
   { /* Since we'll be reading from this descriptor in numerous routines, I store the
     token in a global variable. */
     while (!leaveEarly)
     { // Until we find our key or run out of keys in the descriptor, we'll look for it:
        leaveEarly = PIGetKey(gToken, &key, &type, &flags);
        switch (key)
        { // Only interested in one case, keyMultiImportCount:
          case keyMultiImportCount:
                PIGetCount(gToken, &(gCount));
                leaveEarly = true;
                break;
              /* I'm ignoring all other keys. All I'm looking for is the list, which will be
              preceded by a count key. Once I find that, I drop out, eventually to be called
              by the read routine. */
            } // Close switch
          } // Close leaveEarly
        } // Close gToken
        gQueryForParameters = PlayDialog( );
        // If true, show the dialog

    } // Close descriptorAvailable
  } // End OpenScriptParams
```

The `ReadScriptParams` routine needs to take up where the `OpenScriptParams` routine left off: There is an open descriptor, `gToken`, and it is sitting on an object which is another descriptor. I need to take

that descriptor, open it, parse all its keys, and override my globals. That happens in `SwitchScriptInfo`.

```
void ReadScriptParams (GPtr globals)
{
  int16               loop = 0;
  int32               flags = 0;
  DescriptorTypeID    type = 0;
  DescriptorKeyID     key = 0;
  PIDescriptorHandle  subHandle = NULL;
  PIReadDescriptor    subToken = NULL;
  OSErr               stickyError = noErr;

  DescriptorTypeID    passType = classMultiImportStruct;
  // GetObj needs to know what class type to expect

  DescriptorKeyIDArray subKeyIDArray =
  { keyRows, keyColumns, keyOurMode, NULLID };
```

```
/* These are all expected. If keyInvert is there, it's handled, just not checked off the
list. If I put it in the list, then the list will generally always return with an error,
saying it didn't get keyInvert. I'd rather have it be a pleasant addition than always
expecting it and rarely getting it. */

if (DescriptorAvailable( ))
{ // Have descriptor procs.
   if (gToken)
   { // Global token is valid
     if (gCount > 0)
     { // Have another item waiting
        gLastInvert = false;
        /* Default is no invert. If we get the key, we'll override the default.
        Otherwise, we set it here, just in case we have an error below and don't get
        a chance to set it one way or the other. */

        PIGetObj(gToken, &passType, &subHandle);
```

```
/* From PIUtilities, reads an object from descriptor gToken into subHandle of
type passType */

subToken = OpenReadDesc(subHandle, subKeyIDArray);
/* Can't use OpenReader( ) because that automatically uses the descriptor passed
in gStuff->descriptorParameters. Instead, we use a subroutine, OpenReadDesc, which
opens handle subHandle and tracks array subKeyIDArray, returning its descriptor
token. */
if (subToken)
{ // Was able to open descriptor.
  SwitchScriptInfo (globals, subToken);
  // Reads the keys from descriptor subToken and overrides globals

  stickyError = CloseReadDesc(subToken); // Done
  subToken = NULL; // Just in case
  PIDisposeHandle(subHandle); // Dispose handle
  subHandle = NULL; // Just in case
```

```
            if (stickyError)
            { // Error occurred while reading keys
               if (stickyError == errMissingParameter)
                 {} /* -1715 missing parameter. Walk keyIDArray to find which one. */
               else
                 gResult = stickyError; // Real error occurred
            }
            gContinueImport = true; // We got something, so keep going!
          } // Close subToken
          gCount-; // One less in list
       } // Close count
       if (gCount < 1)
          CloseScriptParams(globals); // That was the last one! Close it up!
     } // Close readToken
  } // Close descriptorAvailable
} // End ReadScriptParams
```

The `SwitchScriptInfo` routine reads keys out of the descriptor, overriding their global values:

```
void SwitchScriptInfo (GPtr globals, PIReadDescriptor token)
{
  DescriptorKeyID     key = 0;
  DescriptorTypeID    type = 0;
  int16               loop = 0;
  int32               flags = 0;
  int32               count = 0;

  double              rows = kRowsMin, columns = kColumnsMin;
  // Default value for rows and columns
  PIType              mode = ourRGBColorMode;
  // Default value for mode is RGB
  Boolean             invert = false;
  // Default for invert is false
```

```
const double        minRows = kRowsMin, maxRows = kRowsMax,
                    minColumns = kColumnsMin,
                    maxColumns = kColumnsMax;
/* PinUnitFloat will pin a value between minimum and maximum bounds, but, since those
values are passed as addresses, I assign these locals to the constant values */

unsigned long       pixelsUnitPass = unitPixels;
// Have to pass address of unsigned long for unitPixels, so assign local to constant

while (PIGetKey(token, &key, &type, &flags))
{ // Continue while there are more keys
    switch (key)
    {
      case keyRows:
          PIGetPinUnitFloat(token, &minRows, &maxRows,
          &pixelsUnitPass, &rows);
          /* Pins the value between min and max, returning it in "rows". It will return
```

```
        coercedParam if it had to coerce the value to between min and max */
        gLastRows = rows; // Assign local double to global short
        break;
    case keyColumns:
        PIGetPinUnitFloat(token, &minColumns, &maxColumns,
        &pixelsUnitPass, &columns);
        // Pins columns between min and max
        gLastCols = columns; // Assign local double to global short
        break;
    case keyOurMode:
        PIGetEnum(token, &mode);
        // Returns an enum — must be the same as terminology enum list
        gLastMode = GetPlugInMode(mode);
        // Maps enum to ordinal
        break;
    case keyInvert:
```

```
                PIGetBool(token, &invert); // Returns boolean
                gLastInvert = invert; // Assigns boolean to global
                break;
            } // Close switch
        } // Close getkey
    } // End SwitchScriptInfo
```

`CloseScriptParams` is called from multiple places whenever there is an error or the list is finished and the descriptor passed to the plug-in by Photoshop should be closed. Note that the descriptor passed by the host is a handle, and is the plug-in's responsibility to deallocate. If I didn't call this routine, we'd have a memory leak, unless I passed the exact same descriptor back to the host. But I don't pass the same descriptor back, because, even while this open descriptor is being read and used to do multiple imports, the `CreateDescriptor` routines are creating descriptors to pass back to the host in `WriteScriptParams`. Ergo, since I'm putting my own descriptor in `gStuff->descriptorParameters`, I have to call `CloseScriptParams`, at least once, to make sure that the host descriptor is disposed.

```
void CloseScriptParams (GPtr globals)
```

```
{
  OSErr                   stickyError = noErr;

  if (DescriptorAvailable( ))
  { // Descriptor procs available
     if (gToken)
     { // Have our global token
       stickyError = CloseReader(&gToken);
       // Closes token, deallocates memory, and sets it to NULL

       if (stickyError)
       { // Oops, got an error
          if (stickyError == errMissingParameter)
            {} // -1715 missing parameter. Sort of late, by now.
          else
            gResult = stickyError; // Real error occurred
       }
     } // Close token
```

```
    } // Close descriptorAvailable
    gCount = 0; // Reset global list count
    gContinueImport = false; // Finish importing and exit
} // End CloseScriptParams
```

Playing back GradientImport

Now that the playback functions have been completed, the last task was to record some actions and play them back to make sure the parameters were honored. It's pretty cool to create a single action that contains multiple imports inside of it, and you can see how the actions palette can get pretty full.

Other issues and future implementation

Opaque data

You can see that the actions palette can fill up pretty fast with large multiple imports. *Opaque data* is the term for information that you don't want displayed in the actions palette. This is sometimes useful because the data: 1) is serial or registration information; 2) is complex; 3) cannot be represented to the user in the actions palette; 4) simply looks yucky.

In `PIActions.h` there is a key, "`keyDatum`" (I couldn't use `keyData`, it was taken) that displays in the

actions palette as:

Data: "..."

Which is an opaque display. `keyDatum` (and other opaque keys) must be stored as textual data. That means that if you want to store an array of hexadecimal values, for instance, you must convert them to their *textual* representation. To store:

```
$01 $02 $03 $04 $05
```

You must store it as:

```
"0102030405"
```

Or some such similar representation. The reason for this, and the reason there are no opaque keys that simply do not display at all in the actions palette, stems from the user interface issues of the AppleScript and AppleEvent automation architecture. Without getting into too much detail, it has to do with the fact that the user side of the architecture is made so that a user may pass any English-like

string into the automation system to be parsed, such as:

```
tell application "Photoshop" to do Gaussian Blur with Radius™ 2.0
```

Opaque data breaks this mold, but not completely, because opaque data, by its definition, has no English equivalent. (Otherwise, you would just display it in the actions palette like any other parameter.) Because strings and sentences can be passed as automation and event requests, even the opaque data must be able to be typed and passed as a simple sentence. So, by this example, the user could pass the event:

```
tell application "Adobe Photoshop 4.0" to do GradientImport with data "0102030405"
```

There is more detail on this in the AppleScript and AppleEvent *Inside Macintosh* books, and references to them in the Photoshop SDK Guide.

External scripting

Scripts can be controlled via OLE on Windows and AppleScript on Macintosh. Documentation on triggering scripts externally is in the *Photoshop SDK Guide* in the scripting chapter and in *Appendix B: OLE Automation.*

Saving filenames

What isn't covered in the scope of this article, but is an interesting scripting question, is what to do with filenames when saving them as scripting keys. I recommend looking at the example Format module in the SDK. The basic logic used by Photoshop for converting the filename dialog into a scripting parameter, and, therefore, the logic I recommend you use, is:

1. If the user types a new name, save that entire path.
2. If the user leaves the default name, save the path to the folder, but append the current filename to the path when saving.

More detail about this is can be found in the SDK guide and the Format example.

Future features

Photoshop 4.0 scripting is available to all plug-in module types, and, as stated, it can control non-scripting aware plug-ins by executing them as if a user had selected them.

We recommend that you update your plug-in to be Photoshop 4.0 scripting-aware. Because execute-only plug-ins pop their user interface every time they're called from an action, a user running a batch

on a folder of hundreds of files is going to have a much more positive experience, and therefore prefer, working with plug-ins that have been made scripting-aware.

I recommend playing with the batch control mechanism to get a good understanding of how it interacts with the user, and also to look at how Save and Open dialogs are handled, as far as scripting is concerned.

Next issue I'll take a look at some of the new plug-in types introduced in Photoshop 4.0 and all the new API features related to those, including color picker plug-ins and the new selection modules. §

DEVELOPING WITH

Adobe PageMaker

The PageMaker 6.5 API

PageMaker 4.2 was the first version of PageMaker to support plug-ins (then known as “Additions”). Plug-ins and external applications used commands and queries to communicate with PageMaker and were the underlying mechanism for the PageMaker scripting language. PageMaker 5.0 and 6.0 saw the addition of new commands and queries, as well as the removal of obsolete items, but the architecture of the API was unchanged.

With the release of PageMaker 6.5, Adobe has kept the commands and queries API, added a new API architecture beside it, and made the relationship between the application and plug-ins more dynamic in nature. The new APIs, collectively referred to as the component interfaces, open up entire new areas of the application to plug-in developers. In this article, we’ll discuss how these new APIs enable richer, more powerful plug-ins.

The pieces of the PageMaker API

The PageMaker API can be divided roughly into two pieces: the component interfaces, and the command and query interface. The component interfaces give you access to the basic areas of functionality within the PageMaker application (such as the print engine, and the window manager), and the command and query interface provides a programming interface to the scripting engine in PageMaker. While the command and query interface is still a vital part of the PageMaker plug-in architecture, it is the component interfaces that give you the additional services, beyond what is available through scripting. You will use the component interfaces together with the commands and queries, in almost all of the plug-ins that you might create.

The architecture of the PageMaker API

In earlier versions of PageMaker, plug-ins were limited to modal interfaces. In the modal scheme, a user invokes a plug-in from the plug-ins menu and the plug-in displays a dialog the user must dismiss before continuing.

In PageMaker 6.5, the modal interface restriction is gone. You can create floating windows that are completely integrated with PageMaker software's windows, or plug-ins that respond to PageMaker

events, rather than being invoked by the user. The windows can be tool palettes, informational displays, or new types of windows designed for specific tasks.

The PageMaker 6.5 API is the first Adobe SDK based entirely on C++. Starting in version 6.0, the PageMaker Class Library offered a C++ framework for commands and queries (for an overview of the PageMaker Class Library, see the *ADA news*, volume 5, number 2). This framework is now the preferred method for executing commands and queries. The new component interfaces, which open up new areas of PageMaker such as the window interface, are based on C++ objects.

The Component Interface

The component interface is the main new feature of the PageMaker 6.5 API. It provides access to events, windows (non-modal), the print stream, more efficient object and text access, converting and saving images, support for frames, and publishing custom components. The component interfaces are provided to a plug-in as sets of related functions packaged into C++ classes. For example the **CIWindow** interface contains functions for creating, showing, hiding, and destroying windows, and the **CIObjectAccess** interface provides a high performance interface for processing the objects in a

PageMaker publication. In some cases the same functionality is available through the new component interface API and through the old command and query API. Where the two APIs overlap, you will want to use the component interfaces for better performance.

The component interface itself is extensible. You can create new components and add their interfaces to the application through the **CIInterfaceManager** and use them as you would any other component interface. Creating a new component is outside the scope of this article, for more information, take a look at the **ExportInterface** sample project in the PageMaker SDK.

Using Component Interfaces

To use one of the interfaces you must first acquire the interface from PageMaker. PageMaker returns a pointer to an interface object (a C++ object). When you are finished with it, you must notify PageMaker to release the interface object.

Acquiring the interfaces is accomplished through the interface manager:

```
PMErr main(PMMessage *pMsg)
{
CIInterfaceManager *theInterfaceMgr = NULL;
```

```
CIObjectAccess *myObject = NULL;
PMErr        errorCode, objectError;
// Get the interface manager from the PMMessage struct,
// The PMMessage struct is found in PMTypes.h
theInterfaceMgr = pMsg->pInterfaceMgr;

if ( pMsg->opCode == kPMDoInvoke )
{
    // Acquire the CIObjectAccess interface…
    errorCode = gInterfaceMgr->AcquirePMInterface(PMIID_OBJACC, (void **)&myObject);
    if (errorCode == CQ_SUCCESS)
    {
        // The interface has been acquired and can be used.
        PMOBJ_REC pageItem;
        objectError = myObject->GetFirstObject(kGetSelectedObjectsOnly, &pageItem);
        if (objectError == CQ_SUCCESS)
        {
```

```
            //Normally, you would actually do something with the objects
            // But it isn't necessary for this example
            ...
            objectError = myObject->GetNextObject(&pageItem);
            ...
            // Since GetFirstObject was called, RestorePage MUST be called.
            objectError = myObject->RestorePage( );
        }
        else
        {
            //Error! Could be a simple CQ_OBJACC_NO_MORE_OBJECTS error
            // which means that there isn't an object to return.
        }
    gInterfaceMgr->ReleasePMInterface( myObject );
    }
}
```

```
// For this example we've returned errorCode, because
// the interface may not be available.
return errorCode;
}
```

The **AcquirePMInterface** method will set up the interface pointer and return the **CQ_SUCCESS** value, or it will return an error code. If an error is returned, you should not call **ReleasePMInterface**.

While the example above skips over the details of writing a plug-in, it is meant as an example of acquiring, using and releasing an interface.

Event Notification

There are two types of events that a plug-in can receive: PageMaker events and system events. While a plug-in will register for specific PageMaker events, the systems events are provided as a part of the support for creating new windows. Typical PageMaker events include, creating new objects in a PageMaker publication, selecting or deselecting an object, and changing the view size. In total, there are over 120 defined PageMaker events.

PageMaker plug-ins can register for one or more of the PageMaker events. Whenever an event occurs, PageMaker calls all plug-ins that have registered for that event. You can then use commands, queries, or component interfaces to respond to the event.

The **CIBasic** interface is used to register for events.

```
gInterfaceMgr->AcquirePMInterface( (unsigned long)PMIID_BASIC, (void **)&basic );
basic->RegisterPMEvent( (PMEventID) PMEVT_OPENPUB_BEFORE );
gInterfaceMgr->ReleasePMInterface( basic );
```

The example above registers for the **PMEVT_OPENPUB_BEFORE** event, which is sent after the user (or another plug-in) has selected a publication to open, but before it is actually opened. You will find that, like the **PMEVT_OPENPUB_BEFORE** event, many of the events end with **_BEFORE** or **_AFTER**. The suffix attached to the event name indicates whether the notification comes before the event occurs or after. The **_BEFORE** events can be used to interrupt PageMaker and replace its functionality. This is accomplished by registering for the desired **_BEFORE** event, and in response to the event, handling the event and returning the **wasHandled** flag as true. (There is a good example of doing just this in the **OpenCopy** sample plug-in that is a part of the SDK.)

A plug-in can remove itself from the notification list for a particular event using the **UnregisterPMEvent** function.

Commands and Queries

Introduced in PageMaker 6.0, the PageMaker Class Library is now the preferred way to perform commands and queries. The PageMaker Class Library has made writing PageMaker plug-ins far easier, allowing you to concentrate on the functionality of your plug-in rather than the details of communicating commands and queries with PageMaker.

The enhanced commands and queries API provides access to new PageMaker 6.5 features including layers, frames, and save image.

Commands and queries are also used to communicate with PageMaker through DDE or AppleEvents.

PageMaker Scripting

In earlier versions of PageMaker, scripting was limited to issuing one or more commands, providing basic automation features to set up or modify publications. PageMaker 6.5 contains an enhanced scripting language to create sophisticated, and even interactive scripts. The scripting language includes

variables, loops, and conditional statements designed to be easily written by end users or developers. The new scripting language is a great tool for testing plug-in ideas, or providing full solutions to PageMaker users. For more information about PageMaker scripting, check out the Adobe Press title: *Adobe PageMaker Scripting: a guide to Desktop Automation*, by Hans Hansen (ISBN: 1-56830-318-1). The Adobe Press Web pages are available at *http://www.adobe.com/adobepress/*.

Programming documentation

The documentation for PageMaker 6.5 has been significantly updated over the 6.0 version. All of the commands and queries are documented in their C++ form, (for 6.0 they were documented in their scripting form.)

The documentation is provided in HTML format for easy navigation and is optimized for onscreen use. There are hundreds of commands, queries, and components each of which is relatively self-contained, but which are related to other commands and queries. HTML provides unparalleled hyperlinking support; the documentation contains approximately 5000-7000 hyperlinks among over 500 HTML files.

The documentation has been designed to be used online, and a Web browser is an excellent tool for this. As a bonus, the HTML documentation for commands, queries, and components are hyperlinked to the actual header and source files, seamlessly merging the actual SDK source code with the documentation.

More to come

In the next issue we will cover the life span of a PageMaker plug-in, from the when the user starts the application to exit. §

PostScript Language Technologies

This month's column offers sample code for listing the writeable storage devices available on your destination printer. The device name, its search order, and the amount of available space is given for each listed device. The code supports both Level 1 and Level 2 devices.

The sample code provides a starting point for applications or drivers that need to write files, forms, fonts, or other resources to a printer's file system, and want to know more detail about that file system.

Example 1 Code To List Writeable Devices

```
%!PS-Adobe-3.0 Query
%%Title: (query for writeable storage devices, search order, space available)
%%?BeginQuery: WriteableDeviceInfo

(——List of writeable storage devices with search order and space——\n) print
/str 128 string def
```

```
% L2? determines whether a device has language level >= 2.

/L2? {
    /languagelevel where {
        /languagelevel get 2 ge
    }{
        false
    } ifelse
} bind def

% The following procedure is called in a Level 2 environment
% to determine if a given device is writeable.

/mayWrite {  %  devname  mayWrite  true/false
```

```
    /dstats exch currentdevparams def
    dstats /Writeable known {dstats /Writeable get}{false} ifelse
    dstats /Mounted known {dstats /Mounted get}{false} ifelse
    dstats /HasNames known {dstats /HasNames get}{false} ifelse
    and and
} def

% The following prints information about a given device to the screen.
% It is called in a Level 2 environment.

/showAvail {  %  devname showAvail  -
    dup print ( ) print
    dup currentdevparams /SearchOrder get str cvs print ( ) print
    currentdevparams /Free get str cvs print (\n) print
} def

% mayWrite1 is the equivalent of mayWrite to be called in a Level 1
% environment.
```

```
/mayWrite1 { % Devname mayWrite1 true/false
    devstatus % False if devname not found, list otherwise
    {
        pop             % Don't want size
        pop             % Won't check free now
        pop             % Or search order
        pop             % Or removable
        and             % Mounted together with hasNames
        and             % And writeable
        exch pop        % We're not checking searchable
    }
    {false}
    ifelse
} def

% showAvail1 performs the equivalent tasks as showAvail, but for a
% Level 1 environment.
```

```
/showAvail1 {  % devname showAvail1 -
    dup print ( ) print
    devstatus
    pop                         % Already checked can call devstatus
    pop                         % Size
    exch str cvs print ( ) print  % SearchOrder
    str cvs print (\n) print    % FreePages
    5 {pop} repeat
} def

L2?
{
    %  Use resourceforall
    (Using Level 2 resource machinery to list download target devices\n) print
    (*)
    {
        dup mayWrite {showAvail}{pop} ifelse
    } 128 string /IODevice resourceforall
```

```
}
{
    %  Try devforall
    systemdict /devforall known
    {
        (Using Level 1 systemdict operator devforall\n) print
        {dup mayWrite1 {showAvail1}{pop} ifelse} 128 string
        devforall
    }
    {
        (devforall not available, presumably no disk available\n) print
    }
    ifelse
}
ifelse
(End of query. \n) print flush
%%?EndQuery: Unknown
%%EOF
```

Output from Example 1

Below is an example of the output that was received by **stdout** when this code was sent to a Level 2 printer with two writeable drives. Notice that neither of the drives is named `%disk0%`. When parsing the backchannel output from this (or any) code, don't make any assumptions about how the information will be distributed among packets. For instance, even the use of **flush** will not guarantee that one line will appear in one packet.

```
(——List of writeable storage devices with order and space——)
Using Level 2 resource machinery to list download target devices
%disk3% 1 101614
%disk1% 0 529477
End of query.
```

DEVELOPING WITH

Adobe Illustrator

Taking Advantage of the Adobe Illustrator 6 API

I'm about to let you in on a heavily guarded secret regarding Adobe Illustrator 6.0. The improvements and feature upgrades were done via plug-ins only. The only changes to the main application were done to the API (application programming interface) itself, so that additional plug-ins and plug-in types would be supported.

Adobe Illustrator 6.0's API is one of the most advanced APIs for any software, allowing plug-in developers to add features by creating model dialog-based functions, floating palettes, and even tools anywhere within Adobe Illustrator software. The sad thing, in my opinion, is that up until this point, few software developers (with the exception of Alien Skin Stylist™ and Extensis VectorTools,™ see sidebar) have really taken advantage of all of these fantastic capabilities.

Ted Alspach is the author of several books, including the bestselling *Macworld Illustrator 6 Bible, KPT Studio Secrets, The Complete Idiot's Guide to Photoshop, Illustrator Filter Finesse, Photoshop Complete* and the just-released *Acrobat 3 Visual QuickStart Guide*. Check out his Mac-produced Web page at *www.bezier.com*, home to VectorVille (*www.bezier.com/vectorville*), a site for vector users and developers.

Maybe you're asking yourself, why bother with a plug-in when I can just create an application that does what I want? Why bother spending the time to learn the API? For starters, creating a plug-in within Adobe Illustrator allows you to take advantage of Adobe Illustrator software's file importing/ exporting options (Adobe Illustrator supports all of Photoshop software's pixel formats, as well as Illustrator native, EPS and PDF files), and printing. You don't need to worry about coding all of that boring stuff, but instead you get to dig into the meaty stuff that's really fun to create; and to use. The Adobe Illustrator API makes it easy to perform almost all of Adobe Illustrator software's functions and features through simple calls.

Even better, the SDK can be found on the Adobe Illustrator 6 CD-ROM, with plenty of sample plug-ins and source code.

Alien Skin Stylist & Extensis VectorTools

Alien Skin Stylist is a plug-in that not only provides complete text and object styles for Adobe Illustrator objects, but also a system for creating what Alien Skin calls "Complex Constructions," sets of styles that can be applied to paths. Most Complex Constructions contain several paths that are updated live when the Stylist plug-in is installed.

Extensis has announced and shown a brand new collection of plug-ins for Adobe Illustrator, VectorTools 2.0. With the exception of Extensis' trademark toolbars and tips, tricks and techniques dialog box, the other seven

plug-ins for Illustrator were created using the Illustrator 6 API. While VectorTools doesn't take full advantage of the API, it does show some of its powerful capabilities, including palettes and tools. Some of the more interesting plug-ins (from a developer's point of view):

VectorLibrary is a floating palette that stores Illustrator objects. Using the Macintosh drag manager, the API allows objects to be dragged in and out of the palette without negatively affecting the artwork in the document.

VectorFrame is another floating palette that provides an interactive slider that places frames on selected objects. The slider adjusts the frame (an Adobe Illustrator path) in real time.

VectorObjectStyles applies tagged styles to Adobe Illustrator paths.

VectorNavigator is a floating palette with two functions. First, it shows the entire existing illustration within the palette, scaled to the size of the palette, with a red rectangle showing what is currently displayed within the document window. Second, it allows the user to move around within the document by dragging the red rectangle around the palette.

VectorMagicWand is clearly the most impressive (both technologically and otherwise) of the set. The plug-in creates a tool that is added to the Plug-In tools palette. This tool is used for selecting paths that are similar to the path that is being clicked on with the tool. The amount of similarity is controlled by several sliders on a floating palette (which can be shown/hidden either via menu or by double-clicking on the tool). In addition, the palette contains a button that is used to select and deselect the MagicWand tool.

Plug-in types

There are three major plug-in types in Adobe Illustrator software: menu selectable modal dialogs, floating palettes, and tools.

Modal Dialogs are the standard “filter” type of plug-ins common to Photoshop software. The user selects a menu item, and a dialog box appears. MetaTools’ Vector Effects, CSI Socket Sets and BeInfinite’s InfiniteFX use modal dialog boxes for their plug-ins. Versions 5.0 and 5.5 of Adobe Illustrator only supported modal dialog box-based plug-ins, and they were only accessible via a submenu off the Filter menu. Version 6 supports putting menu items in any menu, not just the Filter menu.

Floating Palettes are fully supported by Adobe Illustrator 6. If you use the API to create a palette in Adobe Illustrator, the palette is treated as a standard Adobe Illustrator palette, following the behavior of other palettes in Adobe Illustrator, including snapping to the edges of other palettes, snapping to the document window, and snapping to the edges of the screen. Palettes you create via the API are also hidden and shown automatically when the user presses the Tab key.

Tools are plug-ins that add tools to the Plug-in tools palette. Tools can interact with Adobe Illustrator objects in various ways. The Spiral, Polygon, Star and Twirl tools were originally modal-based plug-ins (in version 5.0 and 5.5 of Adobe Illustrator), now transformed into tools with added functionality (double-clicking on the tools in the plug-in tools palette displays a dialog box that is eerily similar to the original filters modal dialog).

Suites

One of the most useful API innovations is that of suites. Adobe Illustrator 6 has several integrated suites that provide loads of additional functionality. The two suites that you might find especially helpful are the Path Construction Suite and the Shape Construction Suite. These two suites assist in creating and adjusting paths, and are especially helpful with distortion filters.

The most striking difference between the 5.0/5.5 distortion filter Twirl and its 6.0 counterpart (besides the fact that it is also a tool) is the way it works. Twirling a star in versions 5.0/5.5 resulted in a twirling of points only; the path shape was only affected in that the line segments followed the path. The Adobe Illustrator 6 Twirl filter (and tool) uses the Shape Construction Suite to adjust the entire path, not just the selected points, resulting in a smooth twirling effect.

The SDK Clock

There's one plug-in located in the SDK folder (on the Adobe Illustrator 6 CD-ROM) that manages to show some of the incredible power of Adobe Illustrator software's API.

To install the plug-in, drag it out of the SDK folder on the Adobe Illustrator 6 CD-ROM and place it in your Adobe Illustrator 6 application's plug-ins folder. Launch Adobe Illustrator. The Object menu will contain two new items: Create Clock and Pause Clock.

Choose Create Clock from the Object menu. A gray clock appears in the center of your document, with a ticking second hand. When I first saw this, I thought, "Cool. Now I can see the time in Adobe Illustrator." I didn't grasp what was happening; the clock was made of Adobe Illustrator paths. The plug-in actually creates animated Adobe Illustrator paths, in this case a set of paths that keep the time.

A few cool things you can do with the clock:

- Save the document and close it. Open it again a few hours, days, or months later, and you'll see that it has been keeping time correctly.
- Select the second hand and change the fill color...while the second hand ticks its way around the center of the clock.

- Pause the clock (using the command in the Object menu) and Option-copy it several times. Change the position of the hour hands by rotating them around the center of each clock slightly. When you Resume the clock running, you'll have a virtual (and accurate) set of clocks displaying the time in multiple time zones.
- Pause the clock and use any of Adobe Illustrator software's tools to distort the paths, then Resume the clock.
- Finally, use the clock as a time stamp by shrinking it down and placing it in the corner of each Adobe Illustrator document.

These are just the possibilities from a user's standpoint. From a developer's point of view, this opens up a whole new area of plug-in development.

Plug-Ins I'd like to See

There are all sorts of plug-ins that could be created for Adobe Illustrator, now that it has such a powerful API. Here are a few ideas of undeveloped plug-ins that Adobe Illustrator users have been clamoring for:

3D Transformation tool, *Find/Replace*, *Live Blends*, and an *Arc Tool.* Macromedia FreeHand™ has these features and many others that could be included in Adobe Illustrator using the Adobe Illustrator API.

Levels Color Controls. I use Levels in Photoshop as much (or more) than curves. Levels is perfect for quick "watermarking" of images.

Layers Management. Many Layer-based functions could be automated or enhanced, such as automatic layer creation, layer sorting and layer linking.

Enhanced Previewing. A plug-in could create a viewing mode that shows anti-aliasing, overprinting, and individual separations.

Spotlight/Lighting Effects Tool. A tool that could shine a spotlight on the artwork, creating both a reflected light surface and a drop shadow.

Random Movement and Distortion. A plug-in that scatters paths and points based on specific criteria.

Adobe Illustrator Document Viewer. A plug-in that cycles through custom views, multiple documents, and more using a simple VCR style palette; mouse clicks could be used to advance through images like a slide show.

Blur and Blur Tool. Blurring can be done with vector objects, it just isn't that easy. A plug-in that automatically blurred would be a tremendous boon.

3D Path Splines. VectorEffects extrudes; Adobe Dimensions® both extrudes and revolves. However, there is no tool that allows paths to be constructed in three dimensions, or to be displayed that way.

A Stippling Tool. A tool to create stippling effects in Adobe Illustrator software, with varying intensity and color amounts.

Path Generator. A plug-in that automatically generates random paths based on specific criteria. Perfect for backgrounds, random objects, and more.

Mosaic Creation. A plug-in to create mosaic tiles from vector artwork; the current Object Mosaic is quite limited.

Area Tool. Adobe Illustrator software's Measure tool is fine for measuring distance. An Area tool would measure the area within several clicked points or selected paths.

Many of these potential plug-ins can be done using little more than the tools provided with the Adobe Illustrator API. Other plug-ins would require a higher level of complexity. But most, if not all of them, are doable.

Extensis has feasibly taken a great step forward with its critically acclaimed set of plug-ins, due to be released in the near future, but there are many more avenues to be explored in the area of vector-based plug-ins. Using Adobe Illustrator software's API will make plug-in development much easier than most developers could imagine; by using it you'll have access to almost every function within Adobe Illustrator. §

DEVELOPING WITH

Adobe FrameMaker

Page Oriented Processing—Using the Frame® Developer's Kit

As a user edits a document, text reflows, page boundaries shift and the page location of objects changes. Perhaps because pages change so frequently, the Frame Developer's Kit (FDK) maintains no list of paragraphs or graphics per page. Nevertheless, FDK client programs can work with documents in a page oriented way.

This articles addresses two apects of the problem of page-oriented processing:

- Given a page identifier, how can an FDK client find the objects that make up that page?
- Given an object identifier, how can an FDK client determine its page location?

Debra Herman teaches Adobe FrameMaker+SGML and FDK classes. For more information, see her Website at *www.dtrain.com* or send email to *info@dtrain.com*.

Pages and Page Frames

The key to working in a page-oriented way is the page frame, an invisible unanchored frame with the exact dimensions of the page in question. Everything on a Frame document page is found within this unanchored frame.

Not a user concept, the page frame is accessible to the FDK programmer using the `FP_PageFrame` property of the page. Given a document and page identifier, you can obtain the identifier of the page frame using the page property `FP_PageFrame`.

```
pFrameId = F_ApiGetId(docId, pageId, FP_PageFrame);
```

Once you know a page's frame identifier, you can return to that page using the unanchored frame property `FP_PageFramePage`.

```
pageId = F_ApiGetId(docId, pFrameId, FP_PageFramePage);
```

Finding All Objects on a Page

Frame pages can contain both text and graphics. Thus, to learn what is on a page, you must locate the text and graphics that make up the page.

Start with the Page Frame

Start out with the identifier of the page that is of interest. Use that identifier to get the associated page frame identifier. With the page frame identifier you can learn what is on the page by examining the graphic objects that make up the unanchored frame that is the page frame.

Locate Text and Graphics

This task of locating text is simplified if you recognize that text is found in either text frames or text lines, both of which are graphic objects. Finding the objects that make up the page can be reduced to the task of finding the graphics that are in the page frame.

Obtain the list of graphics in the page frame as you would any other list of graphics in a frame. Get the head of the list using the `FP_FirstGraphicInFrame` property of the frame. Subsequent graphics are found using the `FP_NextGraphicInFrame` property of the graphic found.

Find Nested Graphics

If your client is interested in a limited subset of all graphics, use `F_ApiGetObjectType()` to determine the type of graphic found. Even if you are interested in any and all graphics, you need to learn the

type of each graphic found so that your client can correctly process those graphics that might themselves contain additional graphics.

The page frame, as any unanchored frame, can contain other unanchored frames or text frames, that can have within them additional objects of interest. Unanchored frames can contain the whole range of frame graphics. Text frames can have anchored frames which can themselves contain additional graphics. Your client must look inside any such objects if it is to find all objects on the page.

To determine if there are anchored frames within a text frame, call `F_ApiGextText()` specifying the document and text frame identifier. Request text items of type `FTI_FrameAnchor`.

```
tItems = F_ApiGetText(docId, textFrameId, FTI_FrameAnchor);
```

To look inside an unanchored frame, examine the list of graphics in that frame much as you might the list of graphics in the page frame.

In looking for nested graphics, it is not necessary to specially process grouped graphics. The list of graphics in a frame includes any grouped graphics and the group members.

Counting Graphics

The `countGraphicsInFrame()` function provides code to locate all graphics within a frame.The function looks at the list of graphics in the specified frame. If a graphic found is a text frame, `countGraphicsInFrame()` looks for all the anchored frames within that text frame. It then calls itself with the anchored frame identifier to recursively look at the graphics inside each anchored frame. If a graphic found is an unanchored frame, `countGraphicsInFrame()` makes a similar recursive call to examine the unanchored frame's graphics.

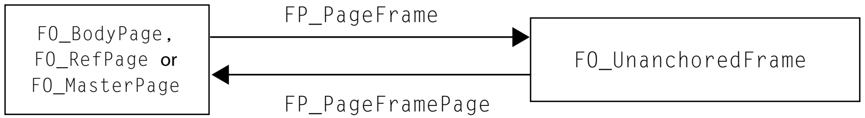


Figure 1 Counting Graphics in a Frame

```
IntT countGraphicsInFrame(docId, frameId)
     F_ObjHandleT docId;
     F_ObjHandleT frameId; /* FO_Unanchored or FO_AFrame */
{
```

```
F_ObjHandleT graphicId; /* ID of graphic object */
IntT gType; /* Graphic type */
IntT i; /* Counter for for loop */
IntT count=0; /* Graphic count */
F_TextItemsT tItems; /* Text items */

graphicId = F_ApiGetId(docId,frameId, FP_FirstGraphicInFrame);
while (graphicId) {
  count ++; /* Graphic just found */
  gType = F_ApiGetObjectType(docId, graphicId);
  switch (gType) {
  case FO_TextFrame:
    tItems = F_ApiGetText(docId, graphicId, FTI_FrameAnchor);
    if (tItems.len != 0) {
      count += tItems.len; /* Count all FO_Aframes */
      for (i=0; i<tItems.len; i++) /* Look for graphics inside FO_Aframe */
          count += countGraphicsInFrame(docId, tItems.val[i].u.idata);
    }
```

```
        F_ApiDeallocateTextItems(&tItems);
        break;
      case FO_UnanchoredFrame:
        /* Look inside unanchored frame for additional graphics */
        count += countGraphicsInFrame(docId, graphicId);
        break;
      }/* End switch */
      graphicId = F_ApiGetId(docId, graphicId, FP_NextGraphicInFrame);
    } /* End while */
    return(count);
  }
```

While `countGraphicsInFrame()` merely counts all graphics, you might make use of the graphic identifiers it locates to do more sophisticated processing. Use `F_ApiGetObjectType()` to selectively process graphics of a specific type.

Determining an Object's Page

Determining the page on which an object appears is, in some respects, the inverse of determining all the graphics on a page. In this case it is the identifier of a frame object that is known, and it is the identifier of the page that is sought. Much as was the case with finding all graphics on a page, it is the nesting of anchored frames within text frames and the full range of graphics within unanchored frames that complicates the task.

Rather than directly seeking a page identifier, it is convenient to find the page frame that contains the object. Once you know the page frame identifier, it is a simple matter to obtain the page identifier using the `FP_PageFramePage` property. The page frame is easily distinguished from other unanchored frames by the fact that it has no parent. That is, when you get the value of a frame's `FP_FrameParent` and get back a zero identifier, the frame just passed to `F_ApiGetId()` is the page Frame.

Once you know a page's frame identifier, you can return to that page using the unanchored frame property `FP_PageFramePage`.

```
pageId = F_ApiGetId(docId, pFrameId, FP_PageFramePage);
```

Locating Objects of Differing Types

In seeking to locate a frame object on the page frame, it is necessary to move up the tree of objects. In going from object identifier to the page frame identifier, you face two distinct situations. In the first case, you have a graphic object (that is not an anchored frame). Such objects have the `FP_FrameParent` property. Otherwise, you have an object that appears in a text frame.

Objects In Text Frames

In dealing with an object that is found at some location in text (that is, in a text frame), the best approach is to translate the object's text location into a paragraph identifier. Once you know the object's location as an `F_TextLoctT` or an `F_TextRangeT`, this is an easy matter of examining the data structure. For objects that might span pages you will need to decide whether to use, for example, the object start or end as the relevant location.

Objects such as markers have the `FP_InTextLoc` property that can be used to tie them to a text location. Cross References and variables have the `FP_TextRange` property. By taking the start location in the text range specified, it is possible to obtain an appropriate paragraph identifier.

Complex objects such as a tables can span pages. You need to narrow your investigation to a particular cell.

Once you have a paragraph or cell identifier, you can use the `FP_InTextFrame` property to determine the text frame in which these objects appear. Recall that text frames do not span pages.

Conveniently enough anchored frames have the `FP_InTextFrame` property.

```
tFrameId = F_ApiGetId(docId, aFrameId, FP_InTextFrame);
```

If the paragraph spans multiple pages, `FP_InTextFrame` will give you the identifier of the first page.

Subcolumns have the similar `FP_ParentTextFrame` property, which takes you to their associated text frame.

```
tFrameId = F_ApiGetId(docId, sColId, FP_ParentTextFrame);
```

If the paragraph spans multiple pages, `FP_InTextFrame` will give you the identifier of the first page.

Graphic Objects

Once you have a text frame, your problem is reduced to that of locating a graphic object. Text frames along with graphic objects such as unanchored frames, rectangles, arcs, polylines, and groups have the property `FP_FrameParent`. This property provides the identifier of the containing frame. There are three possibilities for the value that is obtained when calling `F_ApiGetId( )` with this graphic object property:

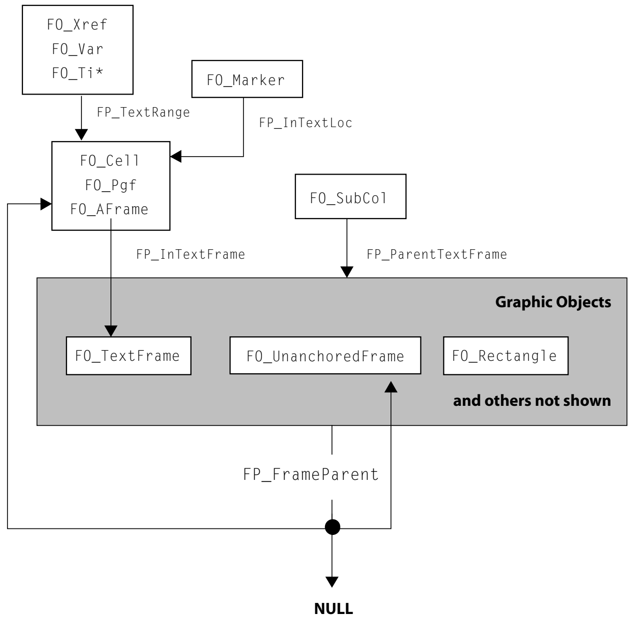
- The object obtained is in an unanchored frame.
- The object obtained is in an anchored frame.
- No object is obtained, as the graphic in question has no frame parent.

If the object located is an unanchored frame, you need simply ask once again for its frame parent. If the object is an anchored frame, you can as before use the anchored frame property `FP_InTextFrame` to locate the text frame in which it is found.

In the final case, you have by definition located the page frame—that frame whose frame parent is 0. The work of finding the object's page is nearly complete.

A schematic view of the process of relating an object identifier to its page frame is shown in Figure 2.

Figure 2 Locating an Object’s Page Frame

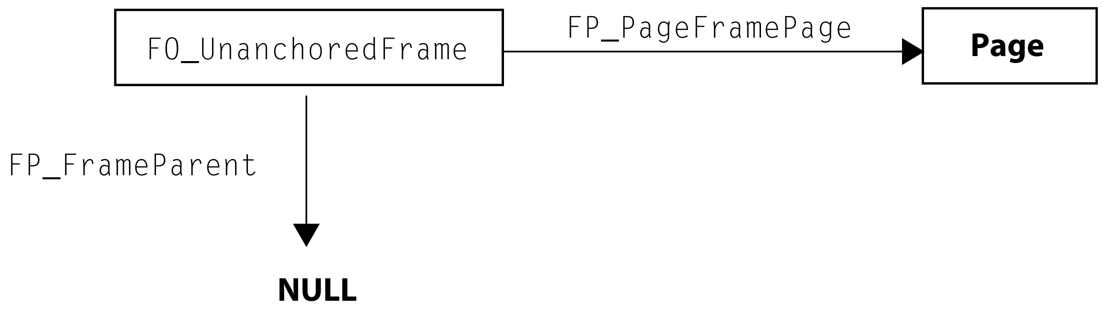


Linking Page Frame to Page

Use the `FP_PageFramePage` property to take you from the page frame to its associated page. If the page is a body page, use the property `FP_PageNumString` to get the page number that actually appears on the printed page or the `FP_PageNum` property to get its page number relative to the start of the document. (Counting starts with page zero.) If the page is a master page or reference page, the `FP_Name` property identifies the page in question.

The properties that link pages and page frames are shown in Figure 3.

Figure 3 Relating the Page Frame to the Page



The techniques described for relating objects to their pages shown can be helpful in writing clients that construct error logs or which do automatic checking for layout problems. But even if you do not need to track page numbers or examine page breaks, the relationship of objects on the page is central to gaining an understanding of the architecture of Frame documents. §

Colophon

All proofs and final output for this newsletter were produced using Adobe PostScript and Adobe Acrobat Distiller for final file output. The document review process was accomplished via electronic distribution using Adobe Acrobat software.

Managing Editor:

Jennifer Cohan, Ursula Kinney

Technical Editor:

Susan Tiner

Art Director:

Min Wang

Designer:

Lorsen Koo

Contributors:

Ted Alspach, Andrew Coven, Nicole Frees, Debra Herman, Gary Staas, Paul Norton

This newsletter was created using Adobe Acrobat, Adobe PageMaker, and Adobe Photoshop, and font software from the Adobe Type Library.