ada technical journal

VOLUME 1, NO. 2

In This Issue

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

- How to Reach Us
- Colophon
- Feature Article: An Introduction to Adobe® Dialog Manager (ADM)
- Developing with Adobe After Effects®
- Developing with Adobe Photoshop®
- Developing with Adobe FrameMaker®
- Developing with Adobe PageMaker®
- Developing with Adobe Acrobat® Capture®
- PostScript® Technology Column

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 Kingdom

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

Feature Article

An Introduction to Adobe® Dialog Manager (ADM)

Your plug-in's user interface is the face of your plug-in. A user's perception and experience of your plug-in greatly depends on having an easy-to-use, logical, well-designed interface. As you know, writing the underlying code for your user-interface is time consuming and mostly platform specific. Enter ADM—Adobe Dialog Manager. A cross-platform user-interface API. ADM makes implementing user interfaces convenient, simple, and even fun. This document provides a technical overview of ADM technology. For more information, refer to the technical specification entitled, "Adobe Dialog Manger Software Development Kit" (henceforth referred to as ADM API reference manual), which can be found on the Illustrator® version 7 SDK.

What is ADM?

The Adobe Dialog Manager (ADM) is a cross-platform API for implementing dialog interfaces. ADM allows plug-ins to create modal and non-modal dialogs as well as a variety of user-interface controls such as radio buttons, pop-up menus, and text-edit boxes. ADM modal and non-modal dialogs can take advantage of the Illustrator 7.0 user-interface conventions. This includes the new 3-d "puffy" design under Mac OS and color themes under Windows® 95. Non-modal dialogs created by ADM resemble tabbed docking dialogs as currently seen in Illustrator 7.0 and Photoshop 4.0. Tabbed docking dialogs built

An Introduction to ADM

using ADM can be docked and tabbed with any other dialog. To the user, your plug-in will appear to be an integrated part of the application. As a matter of fact, ADM was used to implement all of the dialogs in Illustrator 7.0 software.

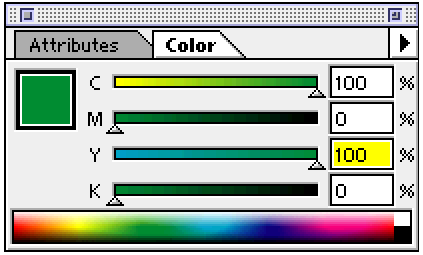


Figure 1a ADM Dialog, Macintosh

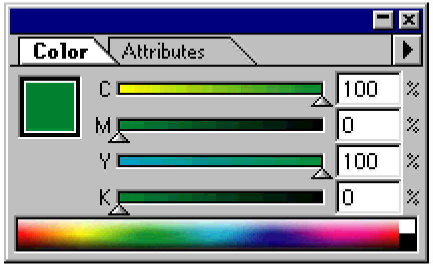


Figure 1b ADM Dialog, Windows

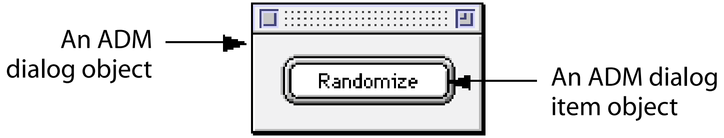


Figure 2 ADM objects

ADM dialogs are built using a variety of ADM user-interface objects. These include the dialog windows (dialog objects) and the dialog items (dialog item objects) within the windows.

The standard dialog types (modal and modeless) and items (buttons and other user-interface controls) are available. Standard platform resources are used to

An Introduction to ADM

define the layout of the UI objects. All objects have properties and events that determine their default behavior and allow them to be modified or extended.

ADM has an object-oriented design even though its interfaces are exported as procedural C functions. This is important since many of the properties, behaviors and callback functions of the various types of ADM UI objects (dialogs or dialog items) are the same. Understanding the fundamentals of one UI object means understanding much about others. For instance, all ADM objects have associated text. For ADM windows the associated text is the window title. For a button, the associated text is the button title. For a text edit item, it's the default text or text which was entered by the user. To access any ADM UI object's associated text, you can use the following functions:

```
SetText( item, text );
GetText( item, &text, maxLen );
```

Some ADM objects need additional support functions or properties. A window object, for example, has callback functions to perform tasks such as setting the minimum and maximum window size. ADM edit text items have additional functions beyond those listed above to support properties such as justification and numeric precision. Please refer to the ADM API reference manual for more details.

Who should use ADM?

Virtually all plug-ins can use ADM in some capacity. Spending a small amount of time writing cross-

An Introduction to ADM

platform ADM code will save much time when porting plug-ins to other platforms. All plug-ins must at least have an About box, which is a simple modal dialog. Plug-in filters can use ADM to create some type of modal dialog. Plug-in tools can create a non-modal floating ADM dialog to provide real-time feedback to the user.

Why should I use ADM?

ADM dialogs provide a look consistent with the user interfaces for Adobe software applications. If you take full advantage of ADM, all dialogs and controls will appear exactly like those found in the application.

ADM functions are platform independent. Your plug-in uses the same ADM function whether it's a Windows or MacOS plug-in. (Currently ADM is only implemented on Windows and MacOS platforms.)

ADM provides a convenient way to write portable plug-in code whether you are evaluating, creating, or modifying, artwork or creating a user-interface.

How can I use ADM? A Technical Overview

Overview

Now that we've covered who should be using ADM, and why, we will address how your plug-in can use ADM.

1. Create a platform resource

You begin by using a platform resource editor to create a dialog resource in your plug-in file. At an appropriate point in your plug-in code (likely responding to an application API event), your plug-in will create a new ADM dialog with either the ADM Dialog suite function `Modal( )` or `Create( )`.

An Introduction to ADM

2. Initialize the dialog

Pass the **Modal( )** or **Create( )** function an initialization procedure, which is called after ADM has loaded the resources and created the dialog. You use this opportunity to set initial values or otherwise customize the dialog's behavior. ADM provides several suites of functions for accessing ADM objects, and these are used to do the initialization. ADM will then display and maintain the dialog for you, processing events as needed. Illustrator will call your plug-in to handle certain events which you requested in your initialization function, such as notification that a button has been pressed.

3. Destroy the dialog

For modeless dialogs, in the initialization procedure you must specify a destroy procedure which your plug-in calls when it shuts down. This is your opportunity to release memory which you allocated and save size and state information about your dialog. ADM will destroy the dialog and free its resources.

For modeless dialogs, your plug-in will call the ADM Dialog suite function **Destroy( )** when the dialog is no longer needed. ADM will destroy the dialog and free its resources.

Event handling

There are five events received by all ADM user-interface objects. They are:

Event	When Received
Init	When object is created
Draw	When screen is invalidated or updated

An Introduction to ADM

Event	**When Received** *(continued)*
Track	When mouse is over the object
Notify	When the object is hit
Destroy	When the object is disposed

For most UI objects, you can rely on the default behavior for an event. For instance, when the cursor moves over a text item, ADM will change it to the insert text cursor. If the default behavior of an ADM user-interface object at a given event is not what is desired, it can be changed by assigning a new event handler to the user-interface object. When your plug-in receives this event, it can invoke a procedure. For example, your plug-in may designate a procedure which is called to respond to a user clicking a button.

Replacing a dialog or item event-handler function is done using definitions and functions in the ADM suite for the object type. The replacement handler function must follow the correct function prototype, which is defined to have enough information to handle the event. For example, to override the default drawing behavior of an ADM item object you would use:

```
// replacement draw proc.
ADMItemDrawProc( ADMItemRef item, ADMDrawerRef
     drawer );
// function used to set the replacement draw proc.
SetDrawProc( ADMItemRef item, ADMItemDrawProc
     drawProc );
// function used to call the default draw proc.
DefaultDraw( ADMItemRef item, ADMDrawerRef
     drawer );
```

An Introduction to ADM

Initialization procedures

An initialization procedure is the first procedure called after a dialog is created. This is your opportunity to do such tasks as setting up your dialog, allocating memory, and filling in initial values. ADM initialization functions for dialogs and items are passed as a parameter when the **Create( )** or **Modal( )** function is called. For example, the initialization procedure for a dialog is:

```
// dialog initialization procedure.
ADMDialogInitProc( ADMDialogRef dialog );
```

Similar to replacement handler procedures, initialization procedures must follow the function prototype as specified in the appropriate header file.

Destroy procedures

A destroy procedure is where your plug-in performs any necessary clean up for an object which is about to be deleted from memory. If an initialization function has allocated memory, it should be released in the destroy procedure. You should also take this opportunity to save size and state information of your dialog. Your destroy procedure is triggered by your plug-in calling the **Destroy( )** function. Similar to initialization procedure, destroy procedures must follow the function prototype as specified in the header file. For example, the destroy procedure for a dialog is:

```
// dialog destroy procedure.
ADMDialogDestroyProc( ADMDialogRef dialog );
```

An Introduction to ADM

UI Object properties

All ADM UI objects have a set of properties associated with them. The following chart describes the properties associated with all ADM UI objects:

Properties and Data	**Purpose**
Type	Defines the general function of the object
Style	Determines the appearance and/or behavior of the object
ID	Numeric reference to the object in its defining space (e.g. its resource number or item number)
Text	The name of the object (It is usually the title or text value of an item)
Visible	Whether or not the object is visible
Enabled	Whether or not the object is enabled
Active	Whether or not the object is the active item, meaning having keyboard focus (e.g. editable text, or on Windows activated by <Enter>)
Plug-in	A reference to the plug-in that created the object
UserData	A pointer to any special data assigned to the object when it was created
LocalRect	The size of the object, 0,0 based
BoundsRect	The rectangle of the object in its container's space (A dialog item is located within a dialog, which is located within the screen bounds)

Properties are obtained or set by using API functions. Because all ADM UI objects share the same set of properties, there are functions of the same name which are associated with different UI objects. For example the API contains three different sets of functions to get or set the **LocalRect** property of different UI objects:

From the ADM Dialog Suite:

```
sADMDialog->SetLocalRect( )
sADMDialog->GetLocalRect( )
```

An Introduction to ADM

From the ADM Entry Suite:

```
sADMEntry->GetLocalRect( )
```

From the ADM Item Suite:

```
sADMItem->SetLocalRect( )
sADMItem->GetLocalRect( )
```

Platform-specific resources

ADM was designed to allow plug-in code for dialogs to be mostly cross platform, virtually eliminating the need to support two or more code bases. Also, it was intended to support the specific look and feel of its runtime platform. To enable this, resources are created on their host platform. ADM will load and use the resources correctly.

On Macintosh, dialogs are made up of normal **'DLOG'** and **'DITL'** resources. Normal dialog item types can be used for standard controls such as buttons and text items. Item types unique to ADM are implemented as controls defined in the ADM API reference manual which describes specific item types. Items that use pictures of some sort can use **'PICT'** and icon family resources to define them. ADM will scan for them in that order and use the first resource it finds with the searched for ID.

On Windows, dialog items are window classes. Variations are controlled by class styles. The mapping of Windows window classes and styles to ADM item types and styles is specified in the ADM API reference manual. Items that take a picture of some sort can use *.bmp* and icon resources. ADM will scan for them in that order and use the first resource it finds with the searched for ID.

An Introduction to ADM

ADM Suites

The ADM services are grouped into 11 suites:

Suite	Functionality
Basic	Provides minimal dialog and resource functionality such as alerts, beeps, resource access, and string utilities
Dialog	ADM property access functions for dialog objects
Item	ADM property access functions for dialog item objects
List	Functions for ADM list item objects
Entry	Functions for working with ADM list entry objects
Notifier	Functions for implementing custom notifier callbacks
Tracker	Functions for implementing custom tracker callbacks
Drawer	Functions for implementing custom drawer callbacks

The ADM Basic suite is acquired as follows:

```
SPBasicSuite *sBasic =
    (SPInterfaceMessage*)message->d.basic;
ADMBasicSuite *sADMBasic;
error = sBasic->AcquireSuite( kADMBasicSuite,
    kADMBasicSuiteVersion, &sADMBasic);
if ( error ) goto error;
```

Once acquired, the functions within it can be accessed:

```
// use a suite function to make a sound.
sADMBasic->Beep( );
```

For detailed information about the ADM suite functions, refer to the ADM API reference manual found on the Illustrator version 7 SDK.

ADM Objects

ADM Dialog Objects

ADM dialog objects can be either modal or floating type. As mentioned earlier, the dialogs created by ADM exactly match those found in the application.

An Introduction to ADM

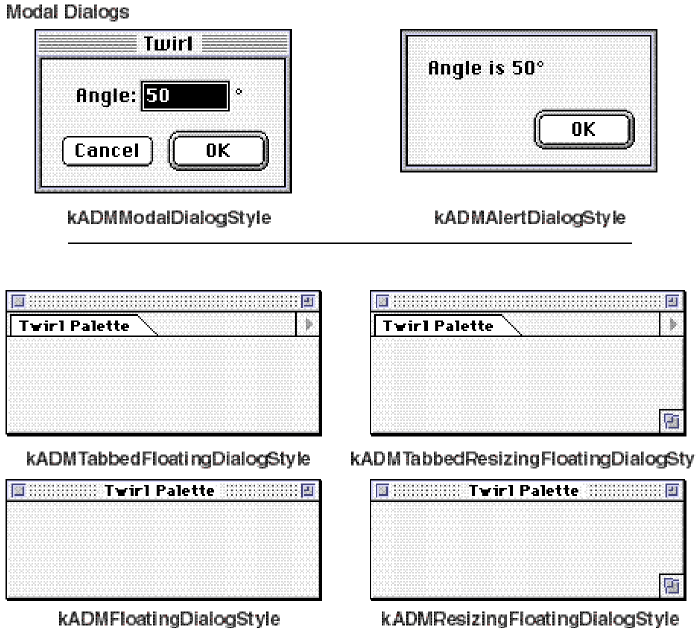


Figure 3 Dialog Objects

Modal dialogs must be dismissed before the user performs another task in the application. Non-modal dialogs will "float" over the main application window and allow the user to interact with the window and the main application simultaneously. Floating tabbed dialogs can be "docked" with others as shown in Figure 4.

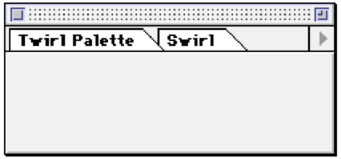


Figure 4 Combined Tabbed Palettes

Behaviors such as moving a window or combining several tabbed windows are handled automatically by ADM. ADM will handle basic window resizing, but the plug-in will likely need to respond to a resize notification by moving its items or changing their size.

The size of the window is initially set by the size of the window resource. It can be set via a function at any time. On Macintosh, ADM dialog windows are specified with **'DLOG'** resources using a custom window definition of ID 1991 (WDEF 124 and variation 7). On Windows, ADM dialog windows are standard **DIALOG** resources.

ADM Dialogs are created by your plug-in using three calls from the ADM Dialog suite. Creating non-modal dialogs requires two functions. To create a non-modal dialog, the plug-in calls the ADM Dialog suite **Create( )** function. When the floating dialog is no longer needed, the plug-in calls the ADM Dialog suite **Destroy( )** function (usually in the destroy procedure). To create a modal dialog, the only function required is the ADM Dialog suite **Modal( )** function. Modal dialogs are automatically destroyed when the user dismisses them.

Dialog Item Objects

There are many types of ADM dialog item objects. Combined with style variations and custom callbacks for drawing, tracking and notification, you can create just about any dialog appearance and behavior needed.

ADM items associated with a dialog are normally created automatically with the dialog. You can manually create and dispose of them in your plug-in. Resource types for all ADM items for each platform are specified in the ADM API reference manual. ADM items are defined in the header file *ADMItem.h.*

An Introduction to ADM

Individual ADM items are of type ADMItemRef. In addition to the standard ADM object properties, all ADM items have a parent dialog and a parent window reference.

In addition to the standard ADM items, **ADMUser** and **ADMCustom** items are used to extend ADM with completely new items. You can create a custom ADM item using an item of type **kADMUserType** as its foundation. Your plug-in would specify a **kADMCustomType** item to access it.

Figures 5 through 12 display the various dialog item objects and how they are represented to the user. For implementation details, refer to the ADM API reference manual.

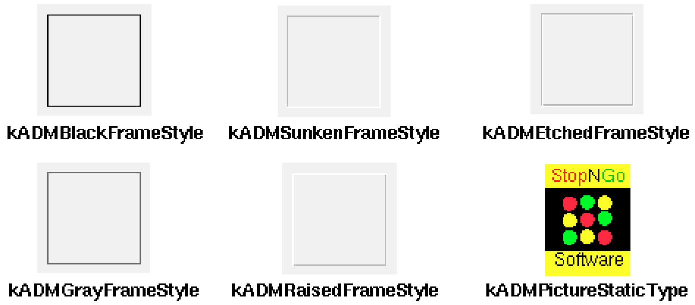


Figure 5 kADMFrameType, kADMPictureStaticType

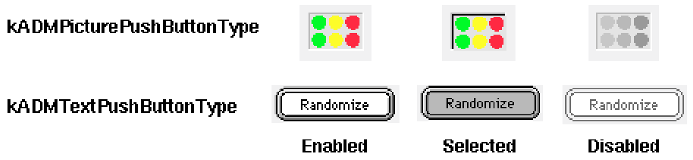


Figure 6 kADMPicturePushButtonType, kADMTextPushButtonType

An Introduction to ADM

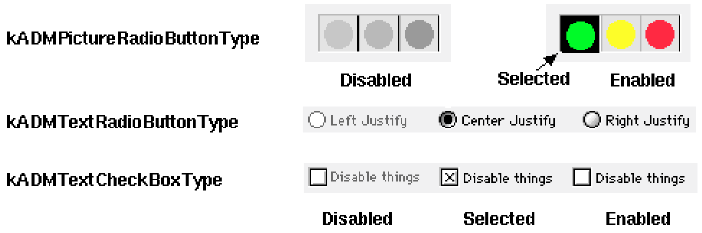


Figure 7 kADMTextRadioButtonType, kADMPictureRadioButtonType, kADMTextCheckBoxType

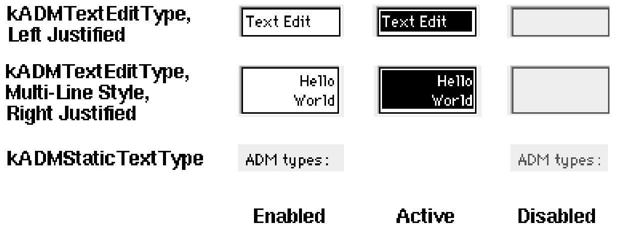


Figure 8 kADMTextEditType, kADMStaticTextType

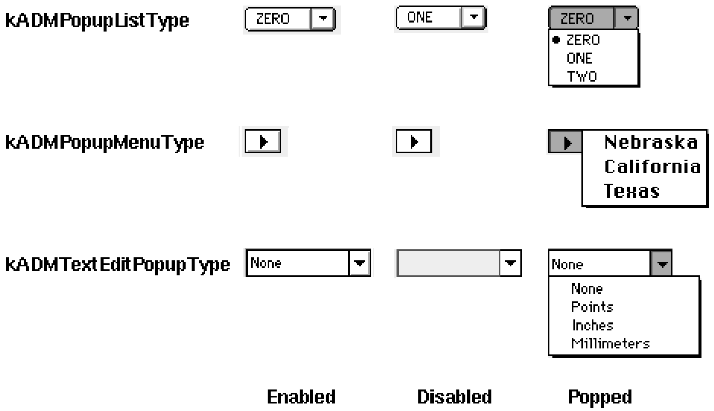


Figure 9 kADMPopupListType, kADMPopupMenuType, kADMTextEditPopupType

An Introduction to ADM

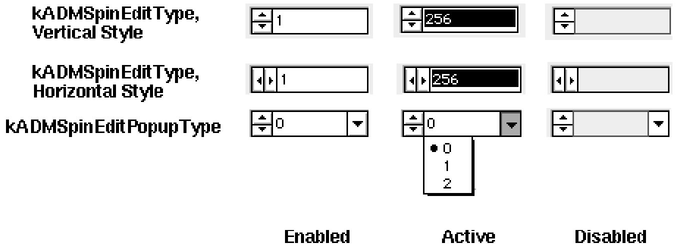

kADMSpinEditType, Vertical Style
kADMSpinEditType, Horizontal Style
kADMSpinEditPopupType
1
256
0
1
2
Enabled
Active
Disabled

Figure 10 kADMSpinEditType, kADMSpinEditPopupType

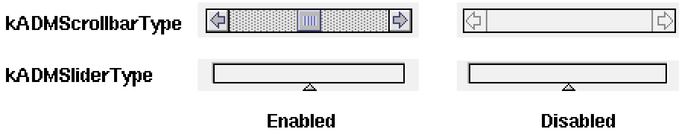


Figure 11 kADMScrollbarType, kADMSliderType

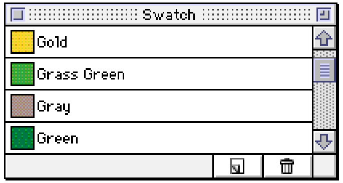


Figure 12 kADMListBoxType

Conclusion

One of the best features of Illustrator 7.0 is its cross-platform capability. This feature is extended to the plug-in developer through ADM. The Illustrator 7.0 API and ADM provide exciting new opportunities for cross-platform user-interface development. By using ADM, your Illustrator plug-in can access the same API that the application uses for its user-interface, providing your plug-in seamless integration into the user environment. §

DEVELOPING WITH Adobe Photoshop®

Color Services and the Photoshop 4.0 Color Picker plug-in

The Adobe Photoshop 4.0 application programming interface introduces a new plug-in type: Color-Picker plug-ins. Color Picker plug-ins allow you to develop and implement your own style of custom color-picking mechanism, with the goal to provide similar results and functions to the Photoshop or operating system's Color Pickers. While the new Color Picker plug-in architecture is very straightforward, the Color Services suite, a long-time complex and inconsistent architecture, has been a bit of a mystery. This article will detail the Color Services suite of functions and structures, and discuss the mechanics of the Color Picker plug-in module.

Scope of this article

Refer to the SDK

Intimate details on all the parameters and callback suites are available in the Adobe Photoshop 4.0.1 SDK, which is available at Adobe's web site:

```
http://www.adobe.com/supportservice/devrelations/sdks.html
```

This article will only address the callbacks and structures that are pertinent to the Color Services suite and the Color Picker plug-in module. There is much more than is covered in this document. Read the SDK for more detail.

Macintosh or Windows®?

Color Services, chosing a color, converting between color spaces, and Color Picker plug-in modules are all part of the Adobe Photoshop API. This means that, except in a few rare exceptions and in the specific cases of opening dialog windows and other platform resources, 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.

Color Services Suite

The Color Services suite is a set of functions that are specific to working with color data. The suite is used to perform any of four operations:

1. Return a sample point.
2. Return either the foreground or background color.

3. Choose a color using the user's preferred Color Picker (which, in Photoshop 4.0, can be a custom Color Picker via a plug-in).
4. Convert a color from one space to another.

The next sections of this article will discuss each operation in detail.

Returning a sample point

Returning the current sample point is one of the more straightforward of the Color Suite's functionalities. All basic Color Suite functions are accessed and triggered by handing a filled-out **ColorServicesInfo** struct to the Color Services callback. The **ColorServicesInfo** struct is defined as:

Figure 1 ColorServicesInfo struct

```
typedef struct ColorServicesInfo
{
    int32 infoSize;             // Size of ColorServicesInfo record.
    int16 selector;             // Operation.
    int16 sourceSpace;          // Color space of input color.
    int16 resultSpace;          // Color space of output color.
```

```
        Boolean resultGamutInfoValid;       // Has resultInGamut field been set?
        Boolean resultInGamut;              // Is returned color in gamut for printing?
        void *reservedSourceSpaceInfo;      // Must be NULL.
        void *reservedResultSpaceInfo;      // Must be NULL.
        int16 colorComponents[4];           // Actual components of input or output color.
        void *reserved;                     // Must be NULL.
        union
        {
            Str255 *pickerPrompt;           // Picker prompt string.
            Point *globalSamplePoint;       // Where to sample.
            int32 specialColorID;           // ID of which color to return.
        } selectorParameter;
    }
```

You can fill this struct out yourself, however there is a routine to initialize a ColorServicesInfo struct to typical values. The routine is in the PIUtilities library, which is the *PIUtilities.c* and *PIUtilities.h* files in the Photoshop SDK's Examples/Common folder:

Figure 2 CSInitInfo routine

```
OSErr CSInitInfo (ColorServicesInfo *ioCSinfo)
{
     OSErr err = noErr; // Assume no error, initially.

     if (ioCSinfo != NULL)
     {

          // Zero color components:
          CSSetColor(ioCSinfo->colorComponents, 0, 0, 0, 0);

          // Selector is one of these:
          // plugIncolorServicesChooseColor,
          // plugIncolorServicesConvertColor,
          // plugIncolorServicesSamplePoint,
          // plugIncolorServicesGetSpecialColor:
          ioCSinfo->selector = plugIncolorServicesConvertColor;
```

```
// sourceSpace and resultSpace can be:
// plugIncolorServicesRGBSpace,
// plugIncolorServicesHSBSpace,
// plugIncolorServicesCMYKSpace,
// plugIncolorServicesLabSpace,
// plugIncolorServicesGraySpace,
// plugIncolorServicesHSLSpace,
// plugIncolorServicesXYZSpace
// resultSpace could also be: plugInColorServicesChosenSpace, in
// that case, resultSpace will return with the users' chosen space:
ioCSinfo->sourceSpace = plugIncolorServicesRGBSpace;
ioCSinfo->resultSpace = plugIncolorServicesChosenSpace;

// Private data. Must be NULL or you'll receive an error:
ioCSinfo->reservedSourceSpaceInfo = NULL;   // Must be NULL.
ioCSinfo->reservedResultSpaceInfo = NULL;   // Must be NULL.
ioCSinfo->reserved = NULL;                  // Must be NULL.
```

```
        // Typical operation is choose a color or show the picker:
        ioCSinfo->selectorParameter.pickerPrompt = NULL;

        // Size parameter easily filled by taking size of structure:
        ioCSinfo->infoSize = sizeof(*ioCSinfo);
    }
    else
    {
        // You have to pass a ColorServicesInfo struct pointer. If you didn't,
        // this function returns a missing parameter error:
        err = errMissingParameter;
    }

    return err;

} // end CSInitInfo
```

CSInitInfo provides a function to reset and validate your initial **ColorServicesInfo** struct. Then you can set it up to tell the Color Services routine to grab a sample point. Figure 3 is a routine to return the first point in a window, in RGB space (regardless of what space the image is in).

Figure 3 GetSamplePoint routine

```
void GetSamplePoint ( globals,
                      *outSampleColorArray)
{
    OSErr err = noErr; // Assume no error, initially.

    // First make sure the ColorServices suite is available, and pop an alert if not:
    if (WarnColorServicesAvailable())
    { // was available. Go ahead and do this.

        // (1) Create the structure variable and the sample point location:
        ColorServicesInfo csInfo;
```

```
// (2) Initialize the structure and check for any errors:
err = CSInitInfo(&csInfo);

if (err == noErr)
{
	// No errors. Do this.

	// Lets go ahead and create a sample point to use. This can
	// be whatever you need, including examining the image's
	// size and using values from that:
	Point myPoint = { 1, 1 };

	// (3) Override the default selector with the sample point command:
	csInfo.selector = plugIncolorServicesSamplePoint;

	// (4) Our source space has been inited to RGBColor.
	// So now, just set our selectorParameter to sample:
	csInfo.selectorParameter.globalSamplePoint = &myPoint;
```

```
// (5) Call the proc using the macro from PIUtilities:
// #define ColorServices(info) (*(gStuff->colorServices)) (info)
err = ColorServices(&csInfo);

if (err == noErr)
{
     // Got the sample point. Return its data.

     // *outSampleColorArray is simply an array of four int16s.
     // Lets use the PIUtilities CSCopyColor routine to copy the
     // returned values into the outgoing array:
     CSCopyColor(colorComponents, outSampleColorArray);
     // outSampleColorArray[0] = red,
     // outSampleColorArray[1] = green,
     // outSampleColorArray[2] = blue,
     // outSampleColorArray[3] = undefined.
}
```

```
            } // err.

        } // ColorServices available

        return err;

    } // end GetSamplePoint.
```

CSCopyColor, called from **GetSamplePoint** is defined in **PIUtilities** and broken out in Figure 4.

Figure 4 CSCopyColor routine

```
OSErr CSCopyColor (int16 *outColor, const int16 *inColor)
{
    short loop;
    OSErr err = noErr;

    if (outColor != NULL && inColor != NULL)
    {
```

```
        for (loop = 0; loop < 4; loop++)
        {
            outColor[loop] = inColor[loop];
        }
    }
    else
    {
        // If the function was passed bad pointers, it will return
        // a missingParameter error:
        err = errMissingParameter;
    }

    return err;

} // end CSCopyColor
```

CSCopyColor, in other words, is a fairly simple copy routine from one four-element **int16** array to another.

Returning the foreground or background color

In many cases, returning the foreground or background color from the Color Services suite is harder than just grabbing it from the parameter block. For instance, the Filter parameter block has two parameters defined for the background and foreground colors, in the color space native to the image:

```
FilterColor   backColor;
FilterColor   foreColor;
```

See *PIFilter.h* for more details.

Some parameter blocks, however, such as the Export parameter block, do not provide values for the foreground and background colors. In that case you'd want to use the Color Services suite to provide that information. Here's a sample routine to return the foreground and background colors in two color arrays:

Figure 5 GetForeAndBackColors routine

```
void GetForeAndBackColors ( globals,
                             *outForeColorArray,
                             *outBackColorArray)
```

```
{
    OSErr err = noErr; // Assume no error, initially.

    // First make sure the ColorServices suite is available, and pop an alert if not:
    if (WarnColorServicesAvailable())
    { // Available. Go ahead and do this.

        // (1) Create the structure variable and the sample point location:
        ColorServicesInfo csInfo;

        // (2) Initialize the structure and check for any errors:
        err = CSInitInfo(&csInfo);

        if (err == noErr)
        { // No errors. Do this.

            // (3) Override the default selector with the get special color command:
            csInfo.selector = plugIncolorServicesGetSpecialColor;
```

```
// (4) Our source space is inited to RGBColor.
// So now, just set our selectorParameter to one of the colors:
csInfo.selectorParameter.specialColorID =
  plugIncolorServicesForegroundColor;

// (5) Call the proc using the macro from PIUtilities:
// #define ColorServices(info) (*(gStuff->colorServices)) (info)
err = ColorServices(&csInfo);

if (err == noErr)
{
     // Got the sample point. Return its data.

     // *outForeColorArray is simply an array of four int16s.
     // Let's use the PIUtilities CSCopyColor routine to copy the
     // returned values into the outgoing array:
     CSCopyColor(colorComponents, outForeColorArray);
```

```
                        // (6) Now let’s get the background color:
                        csInfo.selectorParameter.specialColorID =
                          plugIncolorServicesBackgroundColor;

                          err = ColorServices(&csInfo);

                          if (err == noErr)
                            CSCopyColor(colorComponents, outBackColorArray);

                    } // err @ ColorServices.

                } // err @ CSInitInfo.

            } // ColorServices available.

            return err;

        } // end GetForeAndBackColors.
```

The colors will be returned with the array, in the case of RGB, initialized to:

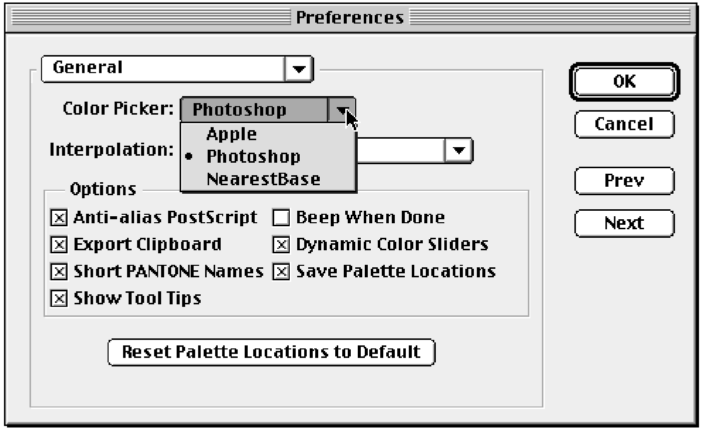
1. array[0] = red,
2. array[1] = green,
3. array[2] = blue, and
4. array[3] = undefined.

Each element is a number from 0 to 255, 255 being the most saturated and 0 indicating an absence of that color.

The parameters of the different color spaces will be discussed later, when converting from one color space to another. First, let's look at how a user would pick a new color.

Choosing a color

There are times when you'll want a user to pick a color so that your filter plug-in can use it for an effect. The Color Services suite provides a selector, **plugIncolorServicesChooseColor**, which pops the user's chosen color picker. The user chooses a color picker via the **File** » **Preferences** » **General** preference panel.



Color Picker: Photoshop

Interpolation:

Options

- [✓] Anti-alias PostScript
- [✓] Export Clipboard
- [✓] Short PANTONE Names
- [✓] Show Tool Tips
- [ ] Beep When Done
- [✓] Dynamic Color Sliders
- [✓] Save Palette Locations

Figure 6 Selecting a color picker from the Photoshop General Preferences Panel

In this panel the user can choose between the OS color picker (Windows or Mac OS), the Photoshop color picker, and any plug-in Color Picker modules. In the example in Figure 6, there is a plug-in color picker called “NearestBase” which is the additional selection.

The examples in this article show the default, the Photoshop color picker.

Now all you need is a routine to pop that picker and return the color. There's already one in **PIUtilities** with its cover macro, **CSPickColor**. **CSPickColor** pops the color picker and returns the user's chosen space and color. You could force **CSPickColor** to always return a specific space, but it's preferable to be flexible and convert the color to the target space after it's returned.

CSPickColor takes a color space and color as input, since the picker does have to default to something in its initial dialog settings. Another input value is the prompt string which is displayed at the top of the picker.

Figure 7 CSPickColor routine

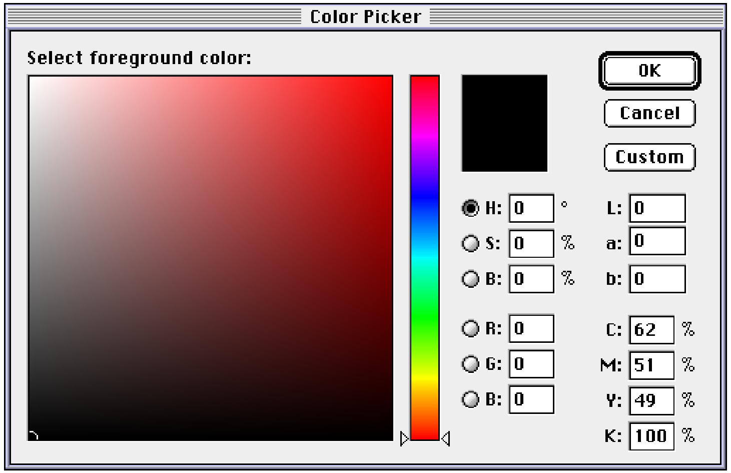
```
OSErr HostCSPickColor ( ColorServicesProc proc,
                        const Str255 inPrompt,
                        int16 *ioSpace,
                        int16 *ioColor)
{
    OSErr err = noErr; // Assume no error, initially.
```

```
if (HostColorServicesAvailable(proc))
{ // Color Services are available. Now populate color services info with stuff.

    if (ioColor != NULL && ioSpace != NULL)
    { // Parameters good.

        ColorServicesInfo           csinfo;

        // Initialize our info structure with default values:
        err = CSInitInfo(&csinfo);

        if (err == noErr)
        { // Init went fine. Now override default values with our own:

            // (1) Set selector to choose color and prompt string:
            csinfo.selector = plugIncolorServicesChooseColor;
            csinfo.selectorParameter.pickerPrompt = (Str255 *)inPrompt;
```

```
        // (2) Set initial color components in source:
        csinfo.sourceSpace = *ioSpace;
        CSCopyColor (csinfo.colorComponents, ioColor);

        // (3) Call convert routine with this info:
        err = (*proc)(&csinfo);

        // (4) If no error, copy the converted colors and user-picked
        // result space:
        if (err == noErr)
        {
            CSCopyColor (ioColor, csinfo.colorComponents);
            *ioSpace = csinfo.resultSpace;
        } // copy.

    } // initinfo.

} // outColor and outSpace.
```

```
        else
        { // outColor or outSpace pointer bad.
            err = errMissingParameter;
        }
    }
    else
    { // color services suite was not available.
        err = errPlugInHostInsufficient;
    }

    return err;

} // end HostCSPickColor
```

Calling **CSPickColor** will show a color picker such as in Figure 8.



Select foreground color:

[✓ H: 0 °]
S: 0 %
B: 0 %
R: 0
G: 0
B: 0
L: 0
a: 0
b: 0
C: 62 %
M: 51 %
Y: 49 %
K: 100 %

Figure 8 Photoshop Color Picker

In the case of Figure 8, the user's choices in the Color Picker reflect, basically, that the user picked nothing or reset everything to 0. Therefore, the return color will be {0, 0, 0, undefined} with a space of **plugIncolorServicesHSBSpace**. Let's assume, in this case, that you'll want to convert that color to the current color space of the document.

Converting between color spaces

What if you want to ask the user for a color then convert it into the document's color space? In a filter plug-in, you can get the document's color mode from the parameter **imageMode**, which will be one of the supported document image modes.

Once you know the image mode, you will want to find that same space's entry for the Color Services suite. There is a facility in **PIUtilities** to do just that: **CSModeToSpace**.

Figure 9 CSModeToSpace routine

```
int16 CSModeToSpace (int16 imageMode)
{
     // Default to same space:
     int16 space = plugIncolorServicesChosenSpace;

     if (imageMode >= plugInModeBitmap && imageMode <= plugInModeRGB48)
     {
          /* static */ const int16 modePerSpace [] =
          {
```

```
    // Little array to map modes to color space values. Make this static if you have
    // A4-global space set up (Macintosh 68k) and this will run faster.
    // If you need A4-space and don't have it set up, and this is made
    // static, you'll have garbage next time it's used:

    /* plugInModeBitmap */          plugIncolorServicesChosenSpace,
    /* plugInModeGrayScale */       plugIncolorServicesGraySpace,
    /* plugInModeIndexedColor */    plugIncolorServicesChosenSpace,
    /* plugInModeRGBColor */        plugIncolorServicesRGBSpace,
    /* plugInModeCMYKColor */       plugIncolorServicesCMYKSpace,
    /* plugInModeHSLColor */        plugIncolorServicesHSLSpace,
    /* plugInModeHSBColor */        plugIncolorServicesHSBSpace,
    /* plugInModeMultichannel */    plugIncolorServicesChosenSpace,
    /* plugInModeDuotone */         plugIncolorServicesGraySpace,
    /* plugInModeLabColor */        plugIncolorServicesLabSpace,
    /* plugInModeGray16 */          plugIncolorServicesGraySpace,
    /* plugInModeRGB48 */           plugIncolorServicesRGBSpace
};
```

```
            space = modePerSpace[imageMode];
        } // If unsupported mode, will return current space.
        return space;
    } // end CSModeToSpace
```

The call to **CSModeToSpace** is straightforward:

```
int16 currentSpace = CSModeToSpace(gStuff->imageMode);
```

The next step is converting from a given space to another. **PIUtilities** contains another routine, **HostCSConvertColor**, with its accompanying macro that you'll want to use to call the routine, **CSConvertColor**, takes a source space, a result space, and a color array, and returns the result space in the color array, or an error if unable to convert the color or missing a parameter.

Figure 10 CSConvertColor routine

```
OSErr HostCSConvertColor ( ColorServicesProc proc,
                           const int16 sourceSpace,
                           const int16 resultSpace,
                           int16 *ioColor)
{
    OSErr err = noErr;

    if (HostColorServicesAvailable(proc))
    { // Color Services available. Now populate color services info with stuff:
        if (ioColor != NULL)
        { // Parameter good.

            ColorServicesInfo                csinfo;

            // Initialize our info structure with default values:
            err = CSInitInfo(&csinfo);
```

```
if (err == noErr)
{ // Init'ed okay.

    // Now override default values with our own.

    // (1) Copy original to colorComponents space then
    // set up space to convert from source to result:
    err = CSCopyColor(&csinfo.colorComponents[0], ioColor);

    if (err == noErr)
    { // Copied okay.
        // (2) Set the source and result space:
        csinfo.sourceSpace = sourceSpace;
        csinfo.resultSpace = resultSpace;

        // (3) Call the convert routine with this info:
        err = (*proc)(&csinfo);
```

```
            // (4) If no error, copy the converted colors:
            if (err == noErr)
               CSCopyColor (ioColor, csinfo.colorComponents);

        } // copy.

    } // initinfo.

} // ioColor

else
{ // ioColor pointer bad.
    err = errMissingParameter;
}
```

```
        }
        else
        { // Color services suite was not available.
                err = errPlugInHostInsufficient;
        }

        return err;

} // end HostCSConvertColor.
```

So to pick then convert the color to the document's color space:

Figure 11 PickThenConvertColor routine

```
void PickThenConvertColor(GPtr globals)
{
        OSErr err = noErr;
```

```
// Define some initial values for the picker to display:
int16    ioColors[4];
int16    ioSpace = plugIncolorServicesHSBSpace;

// Prompt string must be a Pascal string. So, for Windows, do:
#ifdef __PIWin__
    Str255   prompt = "\x0CPick a color";
#else // Mac and Unix understand \p:
    Str255   prompt = "\pPick a color";
#endif

// Set default HSB color. 1=Hue, 2=Saturation, 3=Brightness, 4=undefined:
CSSetColor(ioColors, 0, 255, 255, 0);
err = CSPickColor(prompt, &ioSpace, ioColors);

if (err == noErr)
{ // Picked the color successfully. Now convert it to the right space:
    int16    resultSpace = CSModeToSpace(gStuff->imageMode);
```

```
            CSConvertColor (ioSpace, resultSpace, ioColors);

            // Now ioColors contain the colors in the documents color mode.
            // Do something cool with them here.
        }
        else
        {
            // Some error occurred while picking. Examine and do something
            // with it here.
        }
    } // end PickThenConvertColor.
```

Figure 12 shows the values of the color components of the different color spaces. It is excerpted from the Photoshop SDK Guide from Table B-1. Refer to the guide for more information.

Figure 12 colorComponents array structure

Color space	color Components[0]	color Components[1]	color Components[2]	color Components[3]
RGB	red from 0...255	green from 0...255	blue from 0...255	undefined
HSB	hue from 0...359 degrees	saturation from 0...255 representing 0%...100%	brightness from 0...255 representing 0%...100%	undefined
CMYK	cyan from 0...255 representing 100%...0%	magenta from 0...255 representing 100%...0%	yellow from 0...255 representing 100%...0%	black from 0...255 representing 100%...0%
HSL	hue from 0...359 degrees	saturation from 0...255 representing 0%...100%	luminance from 0...255 representing 0%...100%	undefined
Lab	luminance value from 0...255 representing 0...100	a chromanance from 0...255 representing –128...127	b chromanance from 0...255 representing –128...127	undefined
Grayscale	gray value from 0...255	undefined	undefined	undefined
XYZ	X value from 0...255	Y value from 0...255	Z value from 0...255	undefined

Gotchas

There are some gotchas in the Color Services suite that have resulted in less use over the last years. One is the inability to convert arrays of spaces.

Another difficulty is a legacy bug where no matter what space you convert to, the procedure always returns **`paramErr (-50)`** and converts the requested color to RGB. This has been fixed since Photoshop 3.0.4 software.

The other issue with the Color Services routine is that converting from some spaces to others has not received consistent feedback from third-party developers regarding its validity. Particularly, Lab and XYZ space conversions have been questioned. Unfortunately, those spaces are defined differently in multiple reference guides, and, where one developer might report bad output, another reports those same numbers are exactly what is expected from that space. (Lab, for instance, is based on visual acuity readings of a human subject in a specified environment. The Lab reading is made for the “average” human, with plenty of delta.) Due to that, and time constraints, engineering has only been able to verify the top used spaces. You can, with confidence, use and convert between any of these spaces:

RGB, HSB, CMYK, Gray, HSL

The others *may* be suspect. Try them and see if you get results consistent with your expectations. Let the ADA know if you don't. Especially, let the ADA know what you *expected* if it differs from what you got.

Color Picker plug-ins

With the Color Services suite now under your belt, Color Picker plug-ins should be much more understandable. A Color Picker plug-in's job is to create its own User Interface (UI) to allow the user to pick a color, then return that color in the same color array comprised of four unsigned 16-bit integers, and an enum for what space the result is in.

Color space required for returning a color

While you can implement your own UI to allow a user to pick a color in any space (such as six- and eight-value array spaces, as are becoming popular), you must return a color in the known array space.

This makes sense since the definitions for the different plug-in color services spaces are Photoshop defined, not plug-in defined. There would be no way for a caller to know you're returning a color in six-value space, for instance. So you need to convert that color back into 4-value space, and assign it a space in the Color Services color space enumerations.

Structure for passing color in and out of a picker

The Color Picker plug-in has a simple parameter block that includes a **PickParms** structure that is defined as:

Figure 13 PickParms structure

```
typedef struct PickParms
{
    int16 sourceSpace;     // The colorspace the original color is in.
    int16 resultSpace;     // The colorspace of the returned result
                           // can be plugIncolorServicesChosenSpace.
    // The original color on entry, the returned color on exit:
    unsigned16 colorComponents[4];

    // Pointer to prompt string:
    Str255 *pickerPrompt;

} PickParms;
```

When your Color Picker plug-in is first called, this structure will contain the incoming information for its source space, requested result space, source color, and picker prompt string. You can choose to use or ignore any of those values (for instance, if your plug-in does not need a prompt string).

Color Picker plug-in selectors

The Color Picker plug-in module only gets two selectors, vastly simplifying the dispatching mechanism. They are:

```
pickerSelectorAbout (0)
pickerSelectorPick (1)
```

Obviously, you want to pop your About box on **pickerSelectorAbout**, and pop your UI to pick a color on **pickerSelectorPick**.

Returning a color

It's your responsibility to populate the **PickParms** structure with any new color or space that the user has selected. If you don't do anything, the source color will be returned in the result space.

At minimum, you must copy **sourceSpace** to **resultSpace**. This is because your plug-in may be called with **resultSpace** equalling **plugIncolorServicesChosenSpace**, the command that tells your plug-in to return **resultSpace** as the user's selected space.

plugIncolorServicesChosenSpace makes no sense as a **resultSpace** once execution is returned from your plug-in, especially if it returns from the picker command with no error. So, at least, do:

```
gStuff->pickParms.resultSpace =
    gStuff->pickParms.sourceSpace

return;
```

Popping your UI

Your user interface will be a platform-dependent modal dialog box. Any thermometer or other active tools you'll have to implement yourself. Make sure you don't call the Color Services with a **plugIncolorServicesChooseColor** selector, as that will cause an endless loop!

Example color picker main routine

Here's a sample picker routine from the plug-in example NearestBase from the Photoshop SDK. NearestBase simply cycles through four solid base colors and returns the result in CMYK space:

Figure 14 DoPick routine

```
void DoPick (GPtr globals)
{

    // Set result space to CMYK (see PIGeneral.h for valid spaces):
    gStuff->pickParms.resultSpace = plugIncolorServicesCMYKSpace;

    switch (gCount % 4)
    { // Cycle through four cases:

        case 0: // red.
            gStuff->pickParms.colorComponents[0] = 255 * 0x101;
            gStuff->pickParms.colorComponents[1] = 0 * 0x101;
            gStuff->pickParms.colorComponents[2] = 0 * 0x101;
```

```
        gStuff->pickParms.colorComponents[3] = 255 * 0x101;
        break; // 0.

    case 1: // green.
        gStuff->pickParms.colorComponents[0] = 0 * 0x101;
        gStuff->pickParms.colorComponents[1] = 255 * 0x101;
        gStuff->pickParms.colorComponents[2] = 0 * 0x101;
        gStuff->pickParms.colorComponents[3] = 255 * 0x101;
        break; // 1

    case 2: // blue.
        gStuff->pickParms.colorComponents[0] = 0 * 0x101;
        gStuff->pickParms.colorComponents[1] = 0 * 0x101;
        gStuff->pickParms.colorComponents[2] = 255 * 0x101;
        gStuff->pickParms.colorComponents[3] = 255 * 0x101;
        break; // 2.
```

```
            case 3: // black.
                gStuff->pickParms.colorComponents[0] = 0 * 0x101;
                gStuff->pickParms.colorComponents[1] = 0 * 0x101;
                gStuff->pickParms.colorComponents[2] = 0 * 0x101;
                gStuff->pickParms.colorComponents[3] = 0 * 0x101;
                break; // 3.

            default: // Unsupported, "exercise" error message.
                gResult = pickerBadParameters;
                break;

        } // Switch gCount.

        gCount++;

    } // end DoPick.
```

Color Picker Caveats

There are some things you'll want to watch out for when you write your own Color Picker.

Color Pickers are plug-ins called by plug-ins

Whenever the user clicks on a swatch or a foreground or background color, the chosen Color Picker is displayed. This means your plug-in's UI is called, if the user has selected it in the General Preferences window.

Your Color Picker plug-in, however, can also be called from another plug-in. As you've seen in the previous section on the Color Services suite, a plug-in developer can ask the suite to pop the current Color Picker and return a color. This can get you into a precarious position, having one plug-in being called from another. Here are some things to watch out for, in that event:

1. On the Macintosh, if you are working with resources you will have a couple extra resource forks open besides the application's. You'll have the calling plug-in's resource fork open, as well as your Color Picker plug-in's. Make sure if you're using a string, icon, or any other resources that you use the toolbox call **`Get1Resource()`** instead of **`GetResource`**. By calling **`GetResource()`**, your plug-in will use the currently active resource and drop through the resource chain, instead of just searching one resource deep.

2. Photoshop software is memory intensive, and it's more than likely that your Color Picker plug-in is being called from a memory-intensive plug-in, such as a Filter plug-in. Be as conservative with your memory usage as possible, and mind the **maxData** value carefully and stick to the rule of only using half of it and leaving half of it free for transient operations.
3. The Color Services suite is still available to you to do any sort of conversions, sample grabbing, or special color returning. This makes sense if you want to paint on your UI then grab sample colors from it. However, do not call the Color Services suite with **plugIncolorServicesChooseColor**. Since you're already the active Color Picker, you may have been invoked by the Color Services suite in the first place. Calling it again to choose a color will result in an endless loop.

Other issues and future implementation

Support for 16bpc color

16bpc, stands for 16-bits-per-channel color, which is supported in some of the modes such as **RGB48** and **GrayScale16**.

As Photoshop matures its color services as a whole, you'll see more deep color references and additional functionality to deal with those new color spaces. Adobe is very committed to pushing all their applications to include more deep color, and similar critical color features, with every new version.

Next issue

Next issue we'll delve into the new selection modules and the **ChannelPorts** callback suite, and what it took to make a plug-in that could return a path in the shape of a musical note. §

DEVELOPING WITH

Adobe PageMaker®

A Day in the Life of an Adobe PageMaker Plug-in

Understanding the order of events for a PageMaker plug-in will aid you in building plug-ins. This article will describe the order of events for a plug-in, from the first time it is loaded to when the application is shut down.

A plug-in for the Adobe PageMaker application has only one required entry point, the main function. A plug-in that exports interfaces will have additional entry points for those optional interfaces. The main function receives one parameter, a pointer to the **PMMessage** structure. **PMMessage** is used in determining why the application is calling the plug-in. Before we get started with the events for a plug-in, we will take a quick look at the contents and uses of the **PMMessage** structure. After describing **PMMessage**, we will move on to what happens when the application starts up. Then we will take a tour of what happens during the regular work flow of a plug-in, and finally, what happens at shutdown.

The PMMessage Structure

The **PMMessage** structure contains information that the plug-in needs each time it is called.

```
typedef struct _PMMessage
{
    CIInterfaceManager    *pInterfaceMgr;
    ePMOpcode             opCode;
    long                  id;
    void                  *pPlugInData;
    void                  *pPMData;
} PMMessage;
```

The field that we are concerned with, for the scope of this article, is the **opCode** field. The **opCode** is the message to the plug-in, and the other fields within the **PMMessage** provide the details of the message. The possible codes are: **kPMLoad, kPMUnload, kPMRegister, kPMShutdown, kPMInvoke, kPMEvent, kPMSysEvent, kPMAcquireInterface,** and **kPMReleaseInterface**. Specific explanations of these codes will be provided later in this article as we encounter them in the flow of events for a PageMaker plug-in.

The **pInterfaceMgr** field, which is a pointer to the interface manager, gives a plug-in access to the Component interfaces, and allows the plug-in to add interfaces to the application.

The **id** field identifies which plug-in within a single code module the message is being sent to. It is possible to build several plug-ins within a single DLL or Shared Library, and this `id` indicates which plug-in the message is directed to.

The **pPlugInData** pointer starts out as a null pointer. This pointer is intended for use as a storage facility for your plug-in while PageMaker is still running, but your plug-in is not currently active. A typical use for this pointer is for the plug-in to assign it to a block of memory containing preference settings when it's being 'unloaded', and then to retrieve that information on the next 'load'.

The **pPMData** field gives additional information when the **opCode** isn't enough. One of the messages that rely on the **pPMData** field is the **kPMEvent** message. When a **kPMEvent** message is sent, the plug-in will need to determine which event occurred. To get this information, your plug-in will use the information that PageMaker supplies through the **pPMData** field. Additional information about the **PMMessage** structure can be found in the PageMaker 6.5 SDK.

The following sections describe in more detail each of the messages that the application will send to your plug-in. These messages have been grouped by when they occur:

1. Loading a plug-in.
2. Registering a plug-in.
3. A plug-in at work.
4. Adding interfaces to Adobe PageMaker software.
5. When the application is quitting.

1. Loading a Plug-in: kPMLoad

When a user launches PageMaker, the application reads the resources for your plug-in to determine, among other things, if the plug-in needs to be called at registration time. If the plug-in has the 'register' property set, then the application will call the plug-in at registration time. PageMaker will load a plug-in before sending any other messages to it.

Loading a plug-in, for the Adobe PageMaker application, is a two step process: the code module (DLL or Shared Library) is brought into memory, and then PageMaker sends the 'load' message

(**kPMLoad**) to the plug-in. When the **kPMLoad** message is sent, your plug-in can perform start-up tasks like reading in a preferences file and allocating memory structures. The load message can be sent before the PageMaker start-up process is complete, so you should avoid using the Commands and Queries or any of the interfaces (except the **CIBasic** interface). Plug-ins can function in two basic modes, as far as the loading mechanism is concerned. They can load and unload for each message, or they can stay in memory between messages, waiting until the application is quit before they unload.

2. Registering a Plug-in: kPMRegister

PageMaker uses the **kPMRegister** method to give your plug-in the opportunity to register for PageMaker events, and to create windows and palettes. The register message is sent while PageMaker is still performing its start-up process, and so you should avoid using any of the Commands or Queries, or any of the interfaces other than **CIBasic, CIWindow,** and **CIWinStyle**. Other APIs may function, but their behavior is not guaranteed. If your plug-in is creating a window or floating palette, but you do not want it to show up initially, create the window when your plug-in receives the register method, and leave it hidden, this will ensure that your palette receives the system message that it should. You can register or unregister for events at any point in your plug-in, the register message simply gives you the opportunity to register for the events before any events occur.

The following code was taken out of the OpenCopy sample plug-in, from the Adobe PageMaker 6.5 SDK. This is an example of registering for an event:

```
AcquirePMInterface( ( unsigned long )PMIID_BASIC, ( void** )&basicPtr );
basicPtr->RegisterPMEvent( PMEVT_MENUCOMMAND_BEFORE );
gInterfaceMgr->ReleasePMInterface( basicPtr );
```

'Unregistering' for an event works much the same, you simply use the **UnregisterPMEvent** method.

3. A Plug-in at Work: kPMInvoke, kPMEvent, and kPMSysEvent

The messages that your plug-in will receive during the normal workflow of a PageMaker session are **kPMInvoke, kPMEvent,** and **kPMSysEvent**. In handling these messages, as well as those described in the following section, "Adding Interfaces to PageMaker," you can use all of the APIs in the Adobe PageMaker 6.5 SDK.

When the user selects a plug-in from the plug-ins menu, the **kPMInvoke** message is sent to the plug-in. For many plug-ins this is where the work is done. A typical use for this message is to launch the familiar dialog-based plug-ins.

The **kPMEvent** message is sent to your plug-in any time an event occurs that your plug-in is registered for. To determine which event occurred, and the details of the event (like the specific menu item for the **PMEVT_MENUCOMMAND_BEFORE**) you must look at the **id** and other fields within the **PMMessage**.

The **kPMSysEvent** message notifies your plug-in when a window that it has created is receiving a system message. These system messages are platform-specific messages that a window or palette would normally receive directly from the system. Since your plug-in uses PageMaker APIs to create a window, these messages are directed to your plug-in through the application.

4. Adding Interfaces to PageMaker: kPMAcquireInterface and kPMReleaseInterface

In this section, we will cover two messages that are used when your plug-in adds an interface to the application using the **CIInterfaceManager::AddPMInterface** method, **kPMAcquireInterface** and **kPMReleaseInterface**. When a plug-in requests the interface that your plug-in has added, the application will send the **kPMAcquireInterface** message to your plug-in. When the interface is released by another plug-in, the application will send the **kPMReleaseInterface** to your plug-in.

Typically you will increment a reference count and return a pointer to the interface anytime your plug-in receives the **kPMAcquireInterface** message. On receiving a **kPMReleaseInterface** message you should remove one reference from the reference count, and return the pointer as a null pointer.

5. When the PageMaker application is Closing Down: kPMShutdown and kPMUnload

During the application's shut-down process, your plug-in is given the opportunity to clean up any lingering information, store your preferences on disk, and do any other clean-up tasks necessary for your plug-in. The message that is sent at shut-down is the **kPMShutdown** message. The only message that is sent after the **kPMShutdown** message is the final **kPMUnload** message, which is sent each time the code module that implements your plug-in is about to be removed from memory.

Putting it all together

If you take a look at the sample plug-ins in the PageMaker 6.5 SDK, you will find that the plug-ins respond to the messages according to the task they are designed to perform. The OpenCopy example, for instance, does not use the **kPMInvoke** message, because it is not being launched directly from the plug-ins menu. The ExportInterface plug-in responds to the **KPMAcquireInterface** message because it is adding an interface to PageMaker. The sample plug-ins that are included with PageMaker 6.5 SDK perform very simple tasks, but they do provide a good example of how the **PMMessage** structure, and the messages it can contain, may be used. §

PostScript® Language Technologies

CFF Font Primer

Adobe recently released a specification for a new font format called the Compact Font Format (CFF). For the near future, all Adobe font packages will still use the Type 1 format, but the CFF format will be an integral part of new Adobe PostScript 3™ printer products, and will be the basis for the new OpenType™ font format being jointly developed by Adobe Systems and Microsoft. Below we address some common questions about CFF.

What is a CFF font?

CFF was developed to minimize the amount of storage needed for font programs. The CFF data structure is the equivalent of a PostScript font dictionary, and is used to represent all of a font except the character descriptions (charstrings). The CFF format was designed to work best with the Type 2 charstring format, but other font types may also be used.

Are the specifications of the formats available?

Yes, they are available as Adobe Technical Note #5176, “The Compact Font Format Specification,” and #5177, “The Type 2 Charstring Format.” Both are available at Adobe’s web site at:

```
http://www.adobe.com/supportservice/devrelations/technotes.html.
```

How much smaller are CFF fonts?

A CFF/Type 2 font requires about 40 to 50% less space than the equivalent Type 1 font, on the average. This is without using subroutines, which can further increase the amount of space saved. Similar space savings can be expected for large Asian language fonts that utilize the CID-keyed font format.

How is the space-savings achieved?

Space is saved through a number of optimizations in the encoding method. For example, operators were extended to allow multiple sets of arguments for each operator; commonly used values and character strings can assume a default value; encoding values are more efficiently assigned; and subroutines can be shared among a family of fonts (called a FontSet). The result is that the font file size is significantly smaller, which means less space is used in memory on the host, in documents, and in printer VM, as well as decreasing time required for transmission and for printer downloading.

Is the CFF format similar to the Type 1 format?

Yes, it is based on the same concepts for drawing paths and hinting, but allows a more compact encoding of the font data and procedures. A Type 1 font can be converted into CFF/Type 2 format, and back to Type 1 again, without any loss of quality.

For what products will the CFF/Type 2 format be used?

It is currently used by Adobe Acrobat® Distiller® 3.0 software for embedding fonts in Portable Document Format (PDF) documents (using PDF version 1.2), and it will be the basis for the Type 1 part of the new OpenType font format jointly announced by Adobe and Microsoft. Also, the format will be used for fonts embedded in printer ROMs for PostScript 3 products.

If CFF fonts are used in printers, do I need to change the PostScript language code I generate in my application?

No, a CFF font is represented as a font resource category, which makes them accessible in the standard way via the Language Level 1 and Level 2 font resource operators such as **findfont**, **findresource**, and **resourceforall**.

How is the format used by Adobe Acrobat Software?

Fonts that need to be embedded in a PDF file are first converted to the CFF/Type2 format by the Acrobat Distiller or Acrobat PDFWriter. An additional compression may then be applied, and the result is then stored in the PDF document file. When the document is viewed or printed, the Acrobat viewer converts the font back into the Type 1 format for use by Adobe Type Manager® software, and for inclusion in a PostScript language print file.

Which PostScript interpreters will interpret CFF fonts?

PostScript version 2017 will interpret CFF/Type 2 fonts, but only the single-byte versions. All Adobe PostScript 3 interpreters will support both single- and double-byte CFF fonts.

How can an application or print driver software know if a printer supports the CFF format?

The best way to check for CFF support is for the PostScript language code in the print file to see if the FontType 2 implicit resource is defined. The decision should not be based on the PostScript interpreter version number.

Is the CFF/Type2 format compatible with ATM® and PostScript interpreters?
New versions of ATM, and all Adobe PostScript 3 printers will be able to interpret CFF/Type 2 fonts directly. However, it is not likely that users or applications will encounter a CFF font on their host system, until it is part of an OpenType font.

Will there be commercial third-party font tools that support the format?
There will be commercial tools to create OpenType fonts - which use CFF as part of their structure, although the format is far enough in the future that no products have been announced yet. That is, users will not need to create a "plain" CFF font (without the OpenType wrapper), because the other uses of the format are handled by the printer driver and Acrobat Distiller software.

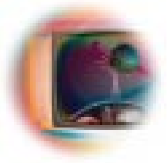
Will the CID-keyed font format utilize the CFF/Type 2 format?
Yes, support for CID-keyed fonts is built in to the format, and there will be OpenType CID-keyed fonts for use with large character set fonts.

How can I get a sample CFF/Type 2 font for test purposes?

Adobe will not be making CFF/Type 2 fonts until that format is used as part of the planned OpenType products. Sample fonts will be available in the PostScript Software Developers Kit (SDK) at that time.

Can I extract a CFF/Type 2 font from a PDF file to use as a sample font?

Yes, but be aware that: 1) the font will not exhibit the full range of CFF/Type 2 features, since the fonts extracted from a PDF file will not be part of a FontSet, will not contain an example of the local and global subroutine features of the format, and may be a subsetted font (containing only the characters used in the document); 2) The font may have Flate or LZW compression applied; and 3) to use the font with an Adobe PostScript 3 interpreter, it would need a PostScript language wrapper added, as described in Adobe Technical Note #5176, "The Compact Font Format Specification." §



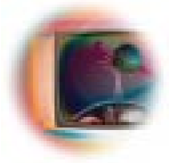
DEVELOPING WITH

Adobe After Effects®

Porting After Effects 3.1 Plug-ins from the Macintosh to Windows® platforms

Adobe After Effects software has been available for the Macintosh since 1992 and has recently been released for the Windows 95 and Windows NT® platforms. The plug-in API for the product has always been quite platform independent, but not until the Windows release have we been able to demonstrate this fact. This article walks through the process of porting a Macintosh-based After Effects 3.1 plug-in to the Windows version of After Effects 3.1. Along the way we discuss some of the few API differences and indicate a number of caveats.

If you'd like to follow along with this discussion you might want to have both the Macintosh and Windows versions of the After Effects Software Development Kits (SDKs), release 4, handy. These can both be downloaded from the Adobe Web site (`http://www.adobe.com/supportservice/devrelations/sdks.html`). The two SDKs contain identical cross platform documentation, sample code, header files, and porting tools. The only differences are in the project files; we provide



Metrowerks CodeWarrior® 11 projects for the Macintosh version and Microsoft® Visual C++™ 4.2 and 5.0 projects for the Windows version.

We'll start our discussion by assuming you have an After Effects plug-in written on the Macintosh that you'd like to port to Windows. On a Macintosh you will need your plug-in's source code, MPW, and the *AE_General.r* file included with the SDK. On a Windows machine you will need the Microsoft Visual C++ 4.2 or 5.0 compiler and the header files and *PiPLTool.exe* included with the SDK.

Convert Resources

The first step is to move the plug-in's resource file, containing a Plug-in Property List resource or **'PiPL'**, from the Mac to Windows. We first need to derez the binary **'PiPL'** resource file on the Mac using the MPW derez tool so we can edit it and move it to Windows. To derez the resource file, use the MPW derez tool, issuing a command that looks something like this:

```
derez 'YourPlugIn PiPL.rsrc' 'AE_General.r' -p > 'YourPlugIn PiPL.r'
```

We then need to hand edit the resulting *.r* file to include a Windows entry point **'PiPL'** property to go along with the Mac PowerPC® ('pwpc') and 68K ('68fp') entry points. If you look at any of the Windows sample files included with the SDK, you'll see examples of how to do this. Look particularly at

the ‘CodeWin32X86’ entry point bracketed with "`#ifdef MSWindows`". When editing the *.r* file, note that its contents are case sensitive.

If you’re creating a plug-in from scratch on Windows, then we suggest using one of the SDK sample *.r* files as a template. **'PiPL'**'s are described in the *Adobe Graphic Application Products, Cross-Application Plug-in Development Resource Guide*, version 1.3 release 4, which is included with both SDKs.

Move the Source Code

The Macintosh-based source code, including the new *PiPL .r* file, can now be moved to a Windows machine. When moving the source files you’ll need to convert the line endings from Macintosh to Windows style. There are many ways to do this, but here are two suggestions:

1) CodeWarrior can work with files with either Mac OS-style (newline) or Windows-style (CR/LF) line endings, so the simplest thing to do is just set the line endings to “DOS”. You can then edit your source code on either platform.

2) An alternative suggestion is to just move your files to Windows and open them in Visual C++. It will warn you and give you the option of converting the line endings. Note, if you attempt to compile a file on Windows without first changing the line endings, you’ll get a considerable number of nebulous error messages.

Create a Visual C++ Project

The next step is to create a Visual C++ project. Again, looking at and copying one of the Windows sample plug-ins is probably the easiest way to see how to set the project up. In general, start with a Win32® DLL project, add your source code, including the *.r* file, and set the various project settings similar to the way we have them set in the SDK sample projects.

The only tricky part is the inclusion of a custom build step, using the PiPLTool (included with both SDKs), which converts the *.r* file to an *.rc* file. To add the custom build step, under “Project Settings” select the *.r* file, click “Custom Build”, and add the following commands:

Build Commands:

```
cl /D “MSWindows” /EP $(ProjDir)\$(InputName).r > $(IntDir)\$(InputName).rr
$(ProjDir)\..\..\..\Resources\PiPLTool $(IntDir)\$(InputName).rr
$(IntDir)\$(InputName).rrc
cl /D “MSWindows” /EP $(IntDir)\$(InputName).rrc > $(ProjDir)\$(InputName).rc
```

Output Files:

```
$(ProjDir)\$(InputName).rc
```

You can now compile the *.r* file by selecting it in the project and then selecting “Compile” from the “Build” menu. If the compile fails, the compiler most likely didn’t find the PiPLTool, so make sure the path to the tool agrees with the command in the second line of the custom build step above. When the compile is successful, add the resulting *.rc* file to your project as well. Now compile the *.rc* file by selecting it in your project and again selecting “Compile” from the “Build” menu. If this fails, one likely reason is a missing “Debug” folder. So create a Debug folder in your projects folder and try again.

Once you’ve added the custom build steps and compiled the resource file, you shouldn’t need to worry about it again. If you make any changes to the *.r* file, Visual C++ should automatically recompile it using the custom build steps.

Finally, create a *.def* file which exports the plug-in’s entry point, using one of the SDK code samples as a template. Add this to your project as well.

Compile and Fix

There are quite likely some source changes you’ll need to make for your plug-in to work cross platform. However, we’ve found it quite expedient to just try compiling and fixing the problems as they arise. If your code pretty much sticks to the After Effects API and avoids platform-specific toolbox calls, you should be able to eliminate the errors quite quickly.

An exception to this rule are plug-ins that use a Custom UI. The nature of Custom UI plug-ins is they can draw their own controls in the Comp, Layer, Effects, and Info windows. Porting these can be quite a lot of work as all the Macintosh toolbox calls will need to be converted to their Windows GDI equivalents. There is an example of this type of plug-in, called “Custom UI In Comp-Layer Window,” included with the SDK.

The compiler will complain about Macintosh types like **Handle, Fixed, Rect, Point, Boolean,** etc. The After Effects SDK headers now contain compatible versions of types such as **PF_Handle, PF_Fixed, PF_Rect, PF_Point,** and **PF_Boolean**. Basically, whenever the compiler complains about something, look in the SDK headers for an equivalent **PF_ type**.

If you do need to include platform-specific code in your plug-in, you should bracket the code with **#ifdef** as the SDK samples demonstrate. For example:

```
#ifndef MSWindows
EnterCodeResource(); /* CodeWarrior (Mac) specific */
#endif
```

Cross-Platform Callbacks

To deal with handles in a cross-platform manner, there is a new set of callbacks which manipulate **PF_Handle**s. We highly recommend the use of these over native code as they are more portable. The new handle callbacks are:

PF_NEW_HANDLE – allocates a new handle

PF_DISPOSE_HANDLE – disposes of a handle

PF_LOCK_HANDLE – locks a handle

PF_UNLOCK_HANDLE – unlocks a handle

PF_GET_HANDLE_SIZE – returns the size (which might be different depending upon the platform your plug-in is running on), in bytes, of a handle.

You should also note the following new callbacks which were added just for Windows-based plug-ins. Do not call these when running on a Macintosh, instead **#ifdef** bracket them as they are presently not defined.

PF_GET_PLATFORM_DATA – retrieves certain platform specific info such as After Effect software's **HWND**.

PF_GET_CGRAF_DATA – allows a plug-ins to retrieve certain platform specific data when given a Mac CGrafPtr. For example, there is a CGrafPtr contained within the **PF_Context** structure and one is returned by the **INFO_GET_POINT** callback.

Futures

As previously mentioned, the After Effects SDK currently supports Metrowerks CodeWarrior 11 on the Macintosh and Microsoft Visual C++ 4.2 and 5.0 on Windows. At the time of this writing we are experimenting with the new Metrowerks CodeWarrior Pro for both Macintosh and Windows. Our initial investigation has encountered minimal problems migrating the Mac SDK examples, so the next release of the SDK will probably include CodeWarrior Pro support for the Macintosh. We think you'll find the ability to nest projects to be of considerable benefit.

Our trials with the Windows version of CodeWarrior Pro have not worked out as well, with the primary problem being the '**PiPL**' resource. We are working with Metrowerks to find a way to invoke the PiPLTool and fix problems with the resource compiler. We're hopeful these changes will make it into the next release so we can supply cross-platform CodeWarrior Pro projects for both Mac and Windows development, in addition to Visual C++ projects for Windows development. §

Translating Multi-Flow Documents to SGML

When a FrameMaker+SGML document is "saved as" SGML, it is not the document as a whole, but a single structured flow, that is saved. While most FrameMaker+SGML documents have a single structured flow, generally the main flow, there is no reason why a document cannot have multiple structured flows.

Note: The potential for more than one structured flow explains why attempting a save as SGML without first establishing an insertion point, results in an error message. Setting the insertion point serves to designate the flow to be translated.

To translate all of the structured data in documents with multiple structured flows, each flow must be saved individually. The end result is multiple SGML files, one per structured flow.

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

If a document contains structured flows that are only loosely related, saving each flow as a separate SGML file may be entirely appropriate. For example, imagine translating a series of newspaper stories, each to a separate SGML instance. On the other hand, if a document's flows bear a predictable relationship and the goal is to construct a single SGML document that preserves this relationship, the default product behavior is problematic.

One obvious solution is to translate each of the flows using the built-in should Save As… command. The resultant SGML files can be combined using a post-processor. Correctly combining the SGML files might be difficult or impossible, however, as the SGML data may not have the information needed to correctly insert the subordinate elements into the document hierarchy.

There is, however, a rather elegant solution to this thorny problem, one that takes advantage of the customizability of FrameMaker+SGML. It requires the creation of a custom replacement for the translator built in to FrameMaker+SGML.

Modifying the Default Translation

When a user saves a FrameMaker+SGML flow as SGML the work of translating that document to SGML is done by **FmTranslator**, a special API client that comes with the standard product. When a flow

is saved as SGML or an SGML document is opened with Frame, it is **FmTranslator** that converts data between the two formats.

In converting documents, **FmTranslator** employs what is *de facto* the default translation. The behaviors that make up this translation are documented in the *FrameMaker+SGML Developer's Guide*.

Setting up an SGML Application

When saving a FrameMaker+SGML document as SGML or opening an SGML document from within FrameMaker+SGML, it is almost always best to employ an SGML application to guide the actions of the default translator. Such an application is defined in a special file, *sgmlapps.doc* and consists of information that can guide **FmTranslator** in its operations. Some key components in an SGML application consist of:

- an SGML declaration
- the DTD for use in constructing an SGML instance on export
- a template for use in constructing a Frame® document on import
- an entry for each document type supported by this application

The application can also reference a read/write rules file and an API client.

Using Read/Write Rules to Modify the Default Translation

Any read/write rules file referenced in the SGML application constitutes the first level of modification to the default translation. Many relatively straight-forward modifications can be made using read/write rules. For example, element or attribute names can be changed on translation; special Frame objects such as markers or table parts can be identified for appropriate translation.

Using a Custom Translator to Modify the Default Translation

While read/write rules allow for important modifications to the default translation, the ultimate tool for customizing the translation between FrameMaker+SGML documents and SGML is a custom replacement for **FmTranslator**.

Custom translators are a special type of Frame API client. They are registered in the same manner as any other API client but are also referenced in the SGML application file in the entry for the application with which they are intended to be used. Adding a **UseAPIClient** element to an **SGMLApplication** entry directs FrameMaker+SGML to substitute the named API client for **FmTranslator**.

Employing a custom translator does not mean bypassing **FmTranslator**'s core functionality. Instead, the custom translator works cooperatively with **FmTranslator**. A custom translator previews the content

proposed for the target document. If that content proves unacceptable, the translator can, by various means, intercede to change the outcome of the translation.

This ability to piggy-back on the built-in translation capabilities of **FmTranslator** is the key to the power of any custom translator. Without it, such a translator would need to do all the work required in a successful translation. With it, the replacement translator need only handle those cases, hopefully small in number, where the default does not produce the desired results.

Event Handlers

The code for FmTranslator is published in the Frame Developer Kit (FDK) *samples* directory. All custom clients begin with the three files that make up FmTranslator: *trnslate.c, import.c* and *export.c.* (Depending upon your product version, these files might have different names, for example, *import.c* might be *rdrevent.c.* The names are not significant.)

Cooperation between a custom translator and **FmTranslator** is made possible by two key event handlers.

- **Sw_EventHandler( )** gives a translator the ability to intervene in the process of writing SGML from a FrameMaker+SGML source.

- **Sr_EventHandler( )** gives a translator the ability to intervene in the writing of a FrameMaker+SGML document from an SGML source.

The Export Event Handler

The import and export event handlers are similar in their basic structure. The file *export.c* consists of the single callback function **Sw_EventHandler( )**. Code for that callback is shown here:

```
SrwErrorT Sw_EventHandler(eventp, swObj)
    SwEventT *eventp;
    SwConvObjT swObj;
{
    return (Sw_Convert(eventp, swObj));
}
```

Key events during the translation process result in calls to the translator via this callback function. The event handler's **eventp** parameter allows the translator client to learn about the data found in the FrameMaker+SGML source. It describes the type of event that has taken place and provides detailed information regarding that event.

The **swObj** parameter, known as the conversion object, allows the client to examine and, if necessary, change the information to be written to the SGML target. The details of the data structure **SwConvObjT** are not documented and it is not possible to directly examine or change conversion objects. Rather, special SGML API functions can be used to examine or alter the conversion objects.

Using FDK Calls in a Custom Translator

Custom replacements for **FmTranslator** have access to the full functionality of the Frame Developer's Kit. When examining conversion objects, it is necessary to utilize special SGML API functions. When examining or changing or creating objects in a Frame document, the FDK is the tool of choice.

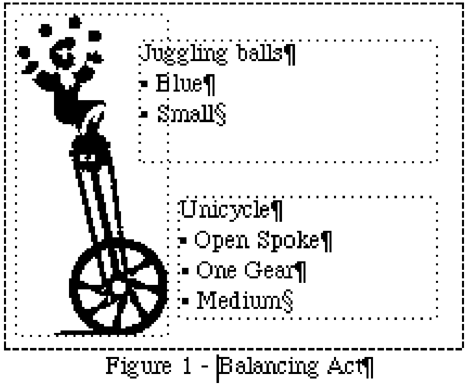
The MultiFlowDoc Application

The **MultiFlowDoc** application is a sample FrameMaker+SGML application. It supports a class of documents that contain multiple structured flows.This application employs the custom translator client **MultiFlowTranslator** to translate documents of this class to an appropriate SGML instance containing the information in each of the structured flows.

Sample Document

Consider a sample document created with the **MultiFlowDoc** application. The document contains illustrations consisting of a caption and imported art overlaid with callouts. The callouts are implemented as small text frames superimposed on the anchored frame that holds an imported graphic.

The following is an example of this type of illustration. The image is an imported graphic. The labels "Juggling balls" and "Unicycle" each occupy a structured flow consisting of a single text frame. The information in each label is augmented by a bulleted list that describes the object in question.



Juggling balls¶
▪ Blue¶
▪ Small§

Unicycle¶
▪ Open Spoke¶
▪ One Gear¶
▪ Medium§

Figure 1 - Balancing Act¶

The MultiFlowDoc EDD

The **MultiFlowDoc** Element Definition Document (EDD) defines the element **Illustration** as shown in the EDD fragment below.

```
Element (Container): Illustration
    General rule: Figure, Caption
Element (Container): Caption
    General rule: <TEXT>
Element (Graphic): Figure
    Initial graphic element format
        1. In all contexts.
            Insert anchored frame.
```

Each **Illustration** element consists of a **Figure** element followed by a **Caption** element. **Callout**, however, is not a possible child of the **Illustration** element but is instead defined as the highest level element in an independent flow. If **Callout** were a sibling of **Figure** and **Caption**, its text would necessarily be part of the document's main flow, an unacceptable option.

The callout data might more naturally be the child of the **Figure** element but this is not possible. This limitation stems from the special nature of those elements that are Frame objects. FrameMaker+SGML makes a strong distinction between container elements and Frame object elements. While container elements can hold other elements or text, object elements (with the exception of footnotes and tables) hold only a single Frame object such as a marker, a cross reference or an anchored frame. Because they cannot hold other elements, they are inevitably terminal nodes in the tree that makes up any structured flow. It is not possible to structure their contents. The **Figure** element can only have a single anchored frame object as its content.

The structure rule for **Callout** is shown below. Note that the **Callout** element is valid at the highest level as it is the root of its own mini-tree of elements constituting a full-fledged structured flow.

```
Element (Container): Callout
    Valid as the highest-level element.
    General rule: Para, List*
```

The MultiFlowDoc DTD

The **MultiFlowDoc** DTD, in contrast to the EDD, can represent the data in one of these illustrations in a rather simple way. The DTD defines the element **Illustration** and its potential children, **Callout, Figure,** and **Caption** as shown. Note that the numerous attributes of the **Figure** element are used to retain information needed to correctly reconstruct the appropriate anchored frame on import.

```
<!ELEMENT Illustration
                    - -  ( Callout*, Figure, Caption ) >
<!ELEMENT Caption   - -  (#PCDATA) >

<!ELEMENT Figure - O  EMPTY>
<!ATTLIST Figure entity       ENTITY    #IMPLIED
                    file      CDATA     #IMPLIED
                    dpi       NUMBER    #IMPLIED
                    impsize   CDATA     #IMPLIED
                    impby     (ref|copy)#IMPLIED
                    sideways  NUMBER    #IMPLIED
                    impang    CDATA     #IMPLIED
```

```
                    xoffset    CDATA     #IMPLIED
                    yoffset    CDATA     #IMPLIED
                    position   NAME      #IMPLIED
                    align      NAME      #IMPLIED
                    cropped    NUMBER    #IMPLIED
                    float      NUMBER    #IMPLIED
                    width      CDATA     #IMPLIED
                    height     CDATA     #IMPLIED
                    angle      CDATA     #IMPLIED
                    bloffset   CDATA     #IMPLIED
                    nsoffset   CDATA     #IMPLIED >

<!ELEMENT Callout    - -  ( Para, List* )>
```

Combining Multiple Structured Flows on Export to SGML

As the sample document shown requires multiple structured flows to be represented in FrameMaker+SGML, and because read/write rules offer no solution, a custom translator is needed to correctly convert this document to SGML. The role of the translator is to convert the **Callout** elements at the appropriate point in the translation process, producing a document instance that preserves the document data and which conforms to the DTD shown.

Controlling Element Translation Order

FrameMaker+SGML flows have a natural translation order that produces the equivalent element tree in the SGML target. A custom client can alter this order to suit a specific DTD-EDD combination. Combining elements in disconnected flows can be thought of as an instance of element translation re-ordering.

By default, **FmTranslator** creates one SGML element for each element found in a FrameMaker+SGML document. If this is not the desired behavior, two SGML API functions make it possible to translate elements in an arbitrary order.

Sw_SetProcessingFlags( ) makes it possible to suppress the translation of an element or of its children or both. Suppressing translation of an element in this way is equivalent to a user doing an unwrap.

With the translation of an element's children suppressed, a custom translator can use a second SGML API function, **Sw_ScanElem( )**, to arbitrarily reorder the translation of that element's children. Calling **Sw_ScanElement( )** with document and element identifiers specified, is much like invoking **FmTranslator** on a subtree of elements. Each of the elements in that tree is converted in the "natural" order, each translation results in the appropriate call to **Sw_EventHandler( )**.

By calling **Sw_EventHandler( )** once for each child, children and their descendents can be translated in any desired order. Obtaining the identifiers for the child elements is a simple matter of employing standard FDK commands to walk the element tree.

Connecting the Callouts to the Figures

As our focus is on exporting FrameMaker+SGML documents to SGML, the **MultiFlowTranslator** needs to modify the **Sw_ExportEventHandler( )** callback in the file *export.c.*

The **MultiFlowTranslator**'s **Sw_ExportEventHandler( )** callback looks for the event **SW_EVT_BEGIN_GRAPHIC**. When this event occurs, it determines if the element in question has the tag **Figure**. Once such an element is found, **MultiFlowTranslator** can use the callback's **eventp** parameter to learn more about the exact nature of the event found. In the case of events involving Frame objects, it is possible to determine the identifier of the associated Frame object. In particular, it is possible to get the identifier of the anchored frame (**FO_AFrame**) associated with the frame element.

FDK functions all require a document identifier to get or set property values for document objects. Once the translator has an object identifier, it is short only the containing document's identifier to put the whole power of the Frame Developer's Kit API at its disposal.

Getting the Document Identifier

The **MultiFlowTranslator** requests the event **SW_EVT_BEGIN_DOC**. The following code obtains the identifier of the document being translated using an SGML API function, **Sw_GetDocId( )**. The document identifier can then be saved as a global variable for later use.

```
    docId = Sw_GetDocId(swObj);
```

Locating the Figure Element

The event **SW_EVT_BEGIN_GRAPHIC** yields an element's identifier. This identifier, together with the document identifier can be used to call FDK functions to get first the definition identifier associated with this element and finally the tag name of the definition. Use the tag name to determine if this is the **Figure** element of concern to the **MultiFlowTranslator**.

```
F_ObjHandleT elementId, defId;
StringT defName;
...
elementId = eventp->fm_elemid;
defId = F_ApiGetId(docId, elementId, FP_ElementDef);
defName = F_ApiGetString(docId, defId, FP_Name);
```

Obtaining the Frame's Identifier

The object identifier for the anchored frame associated with the Frame element can be found from information stored in the **fm_objid** field of the **eventp** parameter.

```
frameId = eventp->fm_objid;
```

Examining the Frame's Content

Using the frame identifier and the identifier of the document undergoing translation it is possible to make use of the FDK functions to examine this frame's contents.The FDK maintains a list of graphics in an anchored frame. Walk the list using the **FP_FirstGraphicInFrame** property of the frame object and the **FP_NextGraphicInFrame** property of the first graphic found.

Use the FDK function, **F_ApiGetObjectType( )** to examine each graphic to determine if it is a text frame. Any text frame found is a potential structured flow. Use the graphic object's **FP_Flow** property to get the identifier of the text frame's flow. If the flow's property **FP_HighestLevelElement** returns an object identifier, this is a structured flow.

```
calloutElemId = F_ApiGetId(docId, flowId, FP_HighestLevelElement);
```

When a structured flow is found, it is a simple matter to call **Sw_ScanElem( )** to convert it.

```
Sw_ScanElem(docId, calloutElemId);
```

SGML Instance Generated

The following is a SGML fragment created from the illustration shown previously using the **MultiFlowTranslator**. The **callout** element is a child of the **illustration** element as desired. The full structure of the **callout** subtree is present, all converted with a single call to **Sw_ScanElem( )**.

```
<illustration>
<callout>
    <para>Juggling balls</para>
    <list>
        <para>Blue</para>
        <para>Small</para></list></callout>
<callout>
    <para>Unicycle</para>
    <list>
        <para>Open Spoke</para>
        <para>One Gear</para>
        <para>Medium</para></list></callout>
```

```
<figure file = “figure1.cgm”
    position = “below” align = “acenter” cropped = “1” float = “0”
    width = “2.944in” height = “2.222in” angle = “0.000”
    bloffset = “0.000in” nsoffset = “0.000in”>
<caption>Figure 1 - Balancing Act</caption></illustration>
```

Reconstructing the FrameMaker+SGML Document on Import

Having successfully translated our multi-flow FrameMaker+SGML document to SGML, the job may only be half done. This is the case if it is necessary to be able to reconstruct the same Frame document from the SGML generated on export. The task of reconstructing multiple flows on import presents a number of challenges. A general sketch of the import side of this translator is presented here. Its full implementation is left as an exercise for the reader.

Each time a `callout` element is encountered in the SGML instance on import, it is necessary to suppress its automatic translation and to have the **MultiFlowTranslator** do the work of creating the appropriate text frame. Suppressing the translation of the callouts can be easily accomplished by

simply failing to call **Sr_Convert( )** when this element is encountered. Constructing the appropriate text frames is a more complex task. As our sample document currently stands, there is insufficient information to accomplish this task.

This situation might be remedied by adding a number of attributes to the **callout** element specification in the DTD. These attributes would hold, for example, the data necessary to reconstruct the text frame in the appropriate location and at the right size. Filling in values for these attributes would require a modification to the export side of the translator.

Once the information needed to create the text frames is in place, creating them is a matter of using the FDK function **F_ApiNewGraphicObject( )**. When the **callout** element is encountered, set the insertion location correctly and allow FrameMaker+SGML to convert the elements that make up the "mini" structured flow.

It is also necessary to prevent Frame from constructing a new graphic file which contains the full image, callouts and all. This job can be done with a read/write rule. Be sure that the **Figure** element in the SGML instance contains information on the appropriate graphic to be used on import (the original imported graphic). Setting the value of this attribute is work that must be done on export. §

DEVELOPING WITH

Adobe Acrobat® Capture®

OLE automation in Acrobat Capture 2.0

Among other new features, the Acrobat Capture 2.0 application offers an OLE automation interface, which allows a Visual Basic® or C/C++ application to control the process of converting paper to Portable Document Format (PDF).

This interface provides the opportunity for document processing vendors to integrate Acrobat Capture with their systems. For instance, a third party could convert paper documents into on-line digital documents that can be viewed by a Web browser. To do this, they could use a scanning server to convert large volumes of paper to TIFF files, convert the TIFF files to PDF files using Acrobat Capture through its OLE automation server, then store the files in a document management system. A CGI script could extract these PDF files and download them to the Web browser. The PDF files would be Web ready, having been optimized for Web downloading by Acrobat Capture.

This article highlights Acrobat Capture's OLE automation interface and shows you how to use it.

Introduction to Capture

The Acrobat Capture application converts image files acquired from a scanner or a fax into PDF, preserving the formatting, page layout, and graphics of the original paper document. Acrobat Capture can apply optical character recognition (OCR) to an image file and produce a PDF file with extractable and searchable text.

Acrobat Capture output is generated from an ACD file, which is an intermediate file format used by Acrobat Capture. The Capture Reviewer application can review and correct ACD files for suspect OCR conversions. Acrobat Capture can also output directly to PDF, various word processing formats, HTML, or a text file. You have several choices for PDF output formats: image only, image plus text, and PDF normal. Image only means the input image file was simply converted to PDF with no OCR; image plus text means image data is supplied, with invisible text below from the OCR engine; and PDF normal is the OCR text without the image data. The last option results in the smallest PDF file since image data is eliminated.

The Acrobat Capture 2.0 application requires a hardware dongle. Each page processed is counted, and the server stops processing when the maximum page count set in the dongle is reached. A variety of dongle packages are available.

OLE automation

Acrobat Capture 2.0 provides an OLE automation server that exports an API to control the Acrobat Capture application's operation. Note that this API is not available with the Acrobat Capture plug-in included with Acrobat 3.0 or 3.01 software.

The OLE automation interface is completely supported by any OLE Automation client, such as Visual Basic, so the API is not limited to C or C++ applications. You can use the Microsoft Foundation Classes (MFC) with this interface to simplify the programming task. The OLE server supports the standard **IDispatch** and **IUnknown** interfaces as well as vtable binding. The Acrobat Capture product provides all the header files you need to use any of the supported development environments.

Your application communicates with Acrobat Capture in two ways: by setting properties and by invoking methods.

The Acrobat Capture OLE server API exports a set of job properties, which govern how Acrobat Capture operates. For instance, a **languageID** property sets the natural language Acrobat Capture uses for optical character recognition. The API provides methods to read and write properties—the get and set methods. (Read-only properties provide only get methods, of course). The "Job Properties" section of this article describes job properties in more detail.

The Acrobat Capture methods tell Acrobat Capture to do something. For example, the **ProcessFile** method tells Acrobat Capture to process a specific file, using the current job properties. The "Methods" section, later in this article, says more about methods.

An Acrobat Capture job is a processing session with a given set of job properties. During a job, Acrobat Capture can create one or more PDF documents, and a document may be created from one or more input files. The methods control the Acrobat Capture application's operation through a set of jobs, documents, and files.

Using the Acrobat Capture application's OLE server involves eight steps. Two of these steps depend on the development environment you are using; the rest do not.

1. Connect to the OLE automation server. How you do this depends on your development environment.

2. Set the job parameters. Set the desired properties by calling job property set methods.

3. Start the job by invoking the **StartJob** method.

4. Start processing a document by calling **StartDoc**.

5. Process the input files for a document. Call **ProcessFile** for each file; several files can produce one PDF file.

6. End processing a document with the **FinishDoc** method.

7. End the job by calling **EndJob**.

8. Disconnect from the server. How you do this depends on your development environment.

Job Properties

Job properties tell Acrobat Capture how to process documents. Each job property has a root name, and each development environment has its own naming convention for referencing that property. A few properties may only be read, such as the application's name; most can be both read and written. Job properties have get and set methods, read-only properties have only get methods. Table 1 shows the ways that the languageID property is referenced in various development environments:

Table 1 Property names in development environments

Root name	Visual Basic	Visual C++	C++ vtable
languageID	languageID	GetLanguageID SetLanguageID	get_languageID put_languageID

Here is an example of setting the **languageID** property in Visual Basic:

```
' English(US) language = 2
CaptureServer.languageID = 2
```

In C++ with MFC, the format is this:

```
CaptureServer.SetLanguageID(OLE_L_ENGLISH_US);
```

If you are using C++ with a vtable, do the following:

```
CaptureServer.put_languageID(OLE_L_ENGLISH_US);
```

Table 2 lists some of the basic job properties supported by Acrobat Capture:

Table 2 Selected Acrobat Capture job properties

Name	**Property**
languageID	language Acrobat Capture uses for optical character recognition (US or UK English, French, German, and others)
orientOpts	page orientations that Acrobat Capture auto-detects (clockwise, 180 degrees, and so on)

Table 2 Selected Capture job properties *(continued)*

Name	Property
outputFormats	output file types that Acrobat Capture creates (PDF Normal, PDF Image Only, PDF Image plus Text)
performPref	controls trade-off between OCR speed and accuracy
sInPath	input files directory
sOutBaseName	name of PDF document Acrobat Capture creates
sOutPath	output file directory
suspectConfThreshold	word confidence threshold used for OCR (words below this confidence level are marked as suspect)
suspectPrefs	controls how Acrobat Capture reports suspect recognition results (low confidence words are output as bitmaps, marked in the ACD file, and so on)
sZoneList	text area locations on page to be recognized (list of rectangles Acrobat Capture examines)
Visible	Controls Acrobat Capture's feedback window visibility

Methods

Job properties tell Acrobat Capture how to process documents; methods tell Acrobat Capture to actually do the processing. OLE automation provides methods to perform all the processing steps (previously listed) to set up the processing environment, convert one or more files to PDF, and cleanly end processing.

The syntax for invoking a method depends on the development environment. Here's how the **StartJob** method is invoked in various environments:

Visual Basic:

```
CaptureServer.StartJob
```

C++ with MFC:

```
CaptureServer.StartJob();
```

C++ Vtable:

```
CaptureServer.StartJob(&capResult);
```

Table 3 lists the most commonly used methods.

Table 3 Selected Capture OLE automation methods

Method	Function
EnableErrorReport	determines whether an error report dialog is displayed
ResetJobInfo	resets all settings in the current job to the default values
SetPDFInfo	sets a PDF file's Info dictionary entry (author, title, subject, keywords)
GetDongleStats	gets information stored in the hardware dongle (total number of pages, number of pages remaining)
GetPageCount	gets the number of pages contained in a selected image file
StartJob	starts a new job with a given set of job properties
StartDoc	starts a new PDF document
ProcessFile	processes a specified file using the current job properties
FinishDoc	finishes processing the current PDF document
EndJob	ends the current processing job
GetErrorString	gets the error string for a Acrobat Capture Server error code

Technical Note #5183, "Acrobat Capture API," provides specifications for all the methods, including individual code samples for most methods.

Visual Basic Example

The Acrobat Capture product includes Visual Basic and C++ samples, showing similar document handling processes. The Visual Basic subroutine here (excerpted from an SDK example) shows how to process one file into a PDF document. This sample code walks through the eight processing steps listed earlier.

```
Private Sub CaptureIt_Click()
Dim rtn As Long
```

Step 1: Connect to the Acrobat Capture Server. This step establishes communication with the Acrobat Capture OLE automation server.

```
Set capture = CreateObject("Adobe\Capture.Server")
```

Step 2: Set up the processing environment. **ResetJobInfo** clears any previous job settings. Use the job property set methods to configure the Acrobat Capture server.

```
' Reset Capture job info
capture.ResetJobInfo
```

```
' Set capture threshold value to 95%.
capture.suspectConfThreshold = 95

' Append pages to the ACD file.
capture.bAppendPgs = True

' Set the output format to PDF.
' "0" = keep the ACD file.
' "1" = PDF/Normal ...
capture.outputFormats = "1"

' Set the Capture language to US English.
' English(UK) = 1.
' English(US) = 2.
' Dutch = 3.
' French = 4.
' German = 5 ...
capture.languageID = 2
```

```
' Set the input directory.
capture.sInPath = InDir.Text

' Set the output directory.
capture.sOutPath = OutDir.Text

' Get dongle statistics for the number of pages left in the dongle.
pgsleft.Text = capture.GetDongleStats(4&)
pglimit.Text = capture.GetDongleStats(5&)

' Set the Info dictionary fields.
capture.SetPDFInfo 1&, "Author Field"
capture.SetPDFInfo 2&, "Keyword Field"
capture.SetPDFInfo 3&, "Title Field"
capture.SetPDFInfo 4&, "Subject Field"
```

Step 3: Start the job using the current settings. Multiple PDF documents can be processed with the same settings.

```
capture.StartJob
```

Step 4: Start processing a document. This step can process several files into a single document.

```
capture.StartDoc
```

Step 5: Process the files for a document, using the current job properties. This example uses only one file.

```
ErrorString.Text = ""
rtn = capture.ProcessFile(InFile.Text)
If (rtn > 0) Then
    ErrorString.Text = capture.GetErrorString(rtn)
End If
```

Step 6: End processing the document. Always call **FinishDoc** to release the resources related to the current document.

```
capture.FinishDoc
```

Step 7: End the job, invalidating the job settings. After this, you can set new job properties and start a new job with **StartJob**.

```
capture.EndJob
```

Step 8: Disconnect from the server.

```
Set capture = Nothing
End Sub
```

Asynchronous Processing

Acrobat Capture may also be controlled through an asynchronous interface; your application can respond to Windows messages or do additional background operations while waiting for the Acrobat Capture server to complete processing. The server spawns a separate thread to handle processing, keeping the OLE interface available for commands. The application is free to do whatever is desired while waiting for processing to finish.

The asynchronous interface allows applications to handle exceptions and recover gracefully. They can terminate a rogue server, retry a page, allow the application to process paint messages, and so forth.

Instead of calling the **ProcessFile** method to process a file, call **ProcessFileEx** to start a processing thread for a range of pages in a file. Periodically call **GetAsyncStatus** to keep track of the thread's status. Other methods allow you to pause, resume, or kill a processing thread. **FinishDocEx** marks the end of a thread's processing; **EndProcessingThread** cleanly terminates the thread.

Using the asynchronous interface with Visual Basic is not recommended since the interface requires Windows API functions that are not directly accessible in Visual Basic.

Conclusion

The Acrobat Capture 2.0 application's OLE automation interface allows Visual Basic, C, and C++ applications to control the process of converting scanned paper documents to PDF. Document processing vendors can integrate Acrobat Capture with their applications, facilitating moving information on paper to an electronic document handling system or the Web.

OLE automation provides methods to set job processing attributes and to perform the steps required to convert an image file to a PDF file with searchable text. Acrobat Capture can also process files in a separate thread, freeing the application for other tasks.

The full Acrobat Capture OLE automation API specification,Technical Note #5183, "Acrobat Capture API", is available on both the Acrobat Capture 2.0 product and the Acrobat 3.01 Plug-ins SDK CD-ROMs. These CD-ROMs also contain Visual Basic and C sample code. The header files required to use OLE automation are provided only with the Acrobat Capture 2.0 product. §

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:

Ursula Ringham

Technical Editor:

Nicole Frees

Art Director:

Min Wang

Designer:

Lorsen Koo

Contributors:

George Ishii, Andrew Coven, Paul Norton, Terry O'Donnell, Brian Andrews, Debra Herman, Gary Staas

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