Adobe

Adobe Photoshop®

Application Programming Interface Guide

Version CS
October 2003

Adobe Photoshop API Guide

Most of the material for this document was derived from earlier works by Thomas Knoll, Mark Hamburg and Zalman Stern. Additional contributions came from David Corboy, Kevin Johnston, Sean Parent and Seetha Narayanan. It was then compiled and edited by Dave Wise and Paul Ferguson. It was later edited for content and updates by Andrew Coven and Thomas Ruark.

1. Introduction

Welcome to the Adobe Photoshop® Software Developers Toolkit!

With this toolkit you can create software, known as *plug–in modules*, that expand the capabilities of Adobe Photoshop.

Audience

This toolkit is for C programmers who wish to write plug–ins for Adobe Photoshop on Macintosh and Windows systems.

This guide assumes you are proficient in the C programming language and its tools. The source code files in this toolkit are written for Metrowerks CodeWarrior on the Macintosh, and Microsoft Visual C++ on Windows.

You should have a working knowledge of Adobe Photoshop, and understand how plug–in modules work from a user's viewpoint. This guide assumes you understand Photoshop terminology such as *paths*, *layers* and *masks*. For more information, consult the *Adobe Photoshop User Guide*.

This guide does not contain information on creating plug–in modules for Unix versions of Photoshop. The Photoshop Unix SDK is available on the Photoshop Unix product CD. You must purchase the product CD to obtain the SDK.

About this guide

This programmer's guide is designed for readability on screen as well as in printed form. The page dimensions were chosen with this in mind. The Frutiger and Minion font families are used throughout the manual.

To print this manual from within Adobe Acrobat Reader, select the "Shrink to Fit" option in the Print dialog.

How to use this guide

This documentation starts with information common to all the plug–in types.

Chapter 2 provides an overview of writing plug–ins, including specific information for Mac OS and Windows development.

Chapter 3 discusses callback routines for the Photoshop host.

Chapters 4 through 9 cover the main six types of plug–in modules (Color Picker, Import, Export, Filter, Format, and Selection) in detail.

Chapter 10 covers all the specific details of Scripting.

The appendices contain specific information about different parameter structures.

An external document, *Plug-in Resources Guide.pdf*, includes valuable tips and tricks for creating Photoshop Plug-ins that function in every major Adobe graphics application.

An external document, *Photoshop File Formats.pdf*, includes all the specifications for the Photoshop, document, and additional file formats.

The best way to use this guide is to first read chapters 1 through 3. Then turn to the chapter containing specific information on the type of plug–in you're going to write.

If writing plug–ins is new for you, we recommend you take some time studying the source code for the sample plug-ins. You may choose to use these source files as the starting point for creating your own plug–in modules.

Contents of the Photoshop plug–in toolkit

The files included with this toolkit include C language header files (`Common/Headers.h`), C language source files (`Common/Sources.c`) and Resource files (`Common/Rez-files.r`) -- these files define the structures, constants and functions you will need to build plug in modules.

The `Examples` directory contains complete source code samples for each type of plug–in.

There is also a directory containing information about the Adobe Developer Association.

SDK discussion mailing list

The Adobe Developers Association maintains an electronic mailing list that is used as peer discussion for developers. It is unmoderated and is populated by developers just like yourself, offering peer discussion of software development kit, Adobe plug-ins, and related issues. The mailing list is for discussion of all of the SDKs that fall under the ADA: Graphics and Publishing, which includes Adobe After Effects, Adobe Illustrator, Adobe PageMaker, Adobe Photoshop, Adobe PhotoDeluxe, and Adobe Premiere; Acrobat; FrameMaker; and PageMaker. To join the discussion send an e-mail to:

```
sdk-requests@adobe.com
```

with the subject:

```
SUBSCRIBE
```

and these fields in your message body:

1. Your full name:
2. Business name:
3. Address:
4. City:
5. State:
6. Country:
7. Country code or Zip:
8. Area code and phone number (business is fine):
9. ADA member number:
   "N/A" if not a member; "Info" if want info.

2. Plug-in Basics

This chapter describes what plug–in modules are and provides information common to all plug–in modules. You should understand this material before proceeding to the chapters detailing the specific types of plug–in modules.

This chapter also contains information about compiling and testing plug–in modules under the Mac OS and Microsoft Windows. Additional compiler–specific information is available in the toolkit header files.

Plug–in modules and plug–in hosts

Adobe Photoshop *plug–in modules* are software programs developed by Adobe Systems and third–party vendors with Adobe Systems to extend the standard Adobe Photoshop program. Plug–in modules can be added or updated independently by end users to customize Photoshop to their particular needs.

This guide also frequently refers to *plug–in hosts*. A plug–in host is responsible for loading plug–in modules into memory and calling them. Adobe Photoshop is a plug–in host.

These Adobe applications function as plug-in hosts: Adobe After Effects, Adobe Premiere, Adobe Illustrator, Adobe PageMaker, and Adobe PhotoDeluxe. Most of these applications support some, but not all, Photoshop plug–in modules. Many applications from third–party developers support the use of Photoshop plug–in modules, as well.

Most plug–in hosts are application programs, but this not a requirement. A plug–in host may itself be a plug–in module. A good example of this is the "Photoshop Adapter" which allows Adobe Illustrator 6.0 to host Photoshop Format and Filter modules.

This toolkit and guide are not designed for developers interested in creating plug–in hosts; the emphasis and goal for this guide is presenting information pertinent to creating plug–in modules.

Unless otherwise stated, Adobe Photoshop is assumed to be the plug–in host throughout this manual. Other hosts may or may not support all the callbacks, properties and functionality described in this guide.

A short history lesson

Plug–ins are not unique to Photoshop. Many Macintosh and Windows applications support some form of plug–in extensions.

Perhaps the best known example of an application that supports a plug-in architecture is Apple's HyperCard, with its support for XCMDs and XFCNs. One of the first companies to incorporate plug–in modules into their products was Silicon Beach, in its Digital Darkroom and SuperPaint products.

Silicon Beach spent a lot of time developing a newer, better designed plug–in implementation. In HyperCard, developers have to paste their plug-ins into the host application using ResEdit. Silicon Beach's design for plug–in modules has the code residing in individual files. This allows the plug-in files to be placed anywhere, not just in the System Folder. Silicon Beach's design

also incorporated the concept of version numbering, which allowed for smooth migration as new functionality was added to the interface.

Adobe Photoshop's implementation of plug–in modules loosely resembles that used by Silicon Beach. It uses a similar calling sequence, and the same version number scheme.

However, the similarity ends there. As Photoshop's plug–in architecture evolved, the detailed interface for Photoshop's plug–in modules became completely different from that used by Silicon Beach. The differences were required primarily to support color images and Adobe Photoshop's virtual memory scheme.

A great overview of Macintosh programming with code fragments is provided in *A Fragment of Your Imagination*, by Joe Zobkiw (1995, Addison-Wesley, NY, ISBN 0-201-483358-0). Chapter ten of Zobkiw's book is solely about writing Photoshop filters.

Macintosh and Windows development

The original plug–in interface was designed when Adobe Photoshop was a Macintosh-only product. This heritage is still apparent today, and affects Windows developers building plug–ins. While you can build plug–in modules for Windows without needing a Macintosh, there are a number of data structures and Mac toolbox–like calls that will appear in your Windows code. The good news is that this makes building plug–ins that work across both Mac OS and Windows easier. The bad news is that if you're developing only for the Windows platform, some of the terminology may be unfamiliar.

Another important difference between the Macintosh and Windows is byte ordering. Motorola and PowerPC processors store pointers, 16–, and 32– bit numbers in big endian format, while Intel processors use little endian format. An example of this is the number 65298, which would disassemble as hex word $FF 12 on a Motorola or PowerPC processor, and $12 FF on an Intel processor. $12 FF on the Intel could be mistaken by beginning Macintosh programmers as 4863, when it in fact is 65298.

Because many Photoshop files are designed to work across both platforms, the Photoshop engineering team chose to standardize on big endian format (Photoshop's heritage shows through again). When programming under Windows, you must be careful to handle byte ordering properly.

Version Information

2.5 & 3.0

The plug–in interface changed significantly with the release of Adobe Photoshop 3.0. The main difference is the use of '`PiPL`' resources to describe plug–in module information. This replaces the older '`PiMI`' resources, although Photoshop still fully supports `PiMI` based plug–in modules. `PiPL` and `PiMI` resources are discussed in the document *Plug-in Resource Guide.pdf*.

The other significant change in version 3.0 is the introduction of the `AdvanceStateProc` callback function. This callback provides improved performance for plug–in modules that handle large images. The `AdvanceStateProc` callback is discussed in chapter 3.

3.0.4

In Photoshop version 3.0.4, the plug–in architecture was again enhanced. You can now set certain properties of a plug–in host using the `SetPropertyProc` callback. The `GetPropertyProc` and `SetPropertyProc` callbacks were grouped together to form a new callback suite. See chapter 3 for details.

Version 3.0.4 also adds a new callback suite: the *image services* suite. The two callback functions in this suite allow you to resample image data, and are useful for various types of filter, import, and export modules. See chapter 3 for details.

3.0.5
Version 3.0.5 offers bug fixes and compatibility updates for PowerPC Macintoshes and Windows 95. There are no new API changes or additions in 3.0.5.

4.0
Version 4.0 expands the API to include two new plug-in module types: *color pickers* and *selections*, and an associated set of callback functions in the new *Channel Ports* suite. There is also an added AppleEvent/AppleScript resource, '`aete`', which describes your plug-in parameters to the Actions palette. Accompanying that is a set of callback functions in the new *Descriptor* suite. The parameter blocks for Import, Export, Filter, and Format have grown to include the new callback suites, where appropriate. The module name for "Acquire" modules was renamed "Import" to be consistent with other Adobe products. This change is cosmetic—the code parameter and function names remain the same, such as `acquireSelectorContinue`.

4.0.1

4.0.1 was mostly a bug fix release, but it does introduce new parameters in `PIAcquire.h` to allow for importing images with transparency.

5.0

Version 5.0 expands the API to include a new plug-in module type: *automation*. These plug-ins can call any scriptable Photoshop command and external filter modules, but cannot manipulate pixel data. See *Photoshop Actions Guide.pdf* included in the SDK for detailed information on automation plug-ins.

5.0.2

Version 5.0.2 was mostly a bug fix release. The canUseICCProfiles flag, ICC profiles handed to Photoshop, and file formats writing 16 bit files were fixed in Version 5.0.2.

5.5

Version 5.5 has very few modifications to the API. A lutCount variable was added to the end of the Import, Export, and File Format module paramater blocks. A new ProgressText PiPL property was added for special handling of the progress bar. The File Format modules also got a new Format Flag for transparency information.

6.0

Version 6.0 has very few modifications to the API. A new routine was added to the Color Space Suite. Three new parameters were added to the Filter Paramater block, iccProfileData, iccProfileSize, and canUseICCProfile. More information is found in the Filter chapter and in the PIFilter.h header file.

7.0

File format plug-ins can now display a thumbnail view in the Open dialog. See the SimpleFormat example fmtCanCreateThumbnail and openForPreview Photoshop has now gone to a new system for the serial number. Old serial

numbers can still be input through the Preferences panel. The Scripting Support plug-in developed for Illustrator is now available for Photoshop. You can now create JavaScript, AppleScript, and Visual Basic scrripts. Scripting Support comes in a separate package and installer.Plug ins now have read access to all the layers in a Photoshop document. See the PoorMansTypeTool example. Share Data Section must be off in your project settings for Macintosh. The AutomationFilter automation plug in and Hidden filter plug in work together to access all layers and all channels of a document. This release moves the pixel read and write inside the Hidden plug in. This allows the Hidden filter to work faster and to use tiling. Photoshop File Formats document is updated to reflect new options in version 7.0. New properties available, see PIProperties.h and the Propetizer example: propTargetLayerLock, propXMP, propSerialString2, propSelectedSliceID, propSelectedSliceIndex, propVersion, propPlayInProgress, propUnicodeName,propUniStr255Name, and propUnicodeNameWithoutExtension. If your dialog uses a custom WDEF your dialog will show up behind the Photoshop palettes on OS X. Use the following OS call to fix the problem. See the Apple documentation for more details. SetWindowGroup(myWindowRef, GetWindowGroupOfClass(kModalWindowClass)); The Project settings for Windows now use Blend for Processor..

CS (8.0)

Documents larger than 30,000 x 30,000 pixels. All records have been updated for 32 bit coordinates for document rects and points. Automation plug-ins can now set a FileBrowseAware flag. Automation plug-ins with this PiPL property will show up in a File Browser menu item as well as the File->Automate menu item. MaxSize and MinSize PiPL properties have been added for setting the max and min size of a document your plug-in will work on. New properties for unicode support. See PIProperties.h

Types of plug–in modules

Adobe Photoshop plug–in modules are separate files containing code that extend Photoshop without modifying the base application.

Photoshop supports nine types of plug–in modules:

Automation
Automation modules access all Photoshop scriptable events. These modules appear under the **Automate** menu or **Help** menus. These modules are documented in *Photoshop Actions Guide.pdf*.

Color Picker
Color Picker modules provide a plug-in interface for implementation of different color picker's in addition to Photoshop's and the system's color pickers. They appear whenever the user requests a unique or custom color (such as clicking on the foreground or background colors in the tools palette) and are selected in the **Preferences...** General dialog. These modules are documented in chapter *5. Color Picker Modules*, on page 83.

Import
Import modules open an image in a new window. Import modules can be used to interface to scanners or frame grabbers, read images in unsupported or compressed file formats, or to generate synthetic images. These modules are accessed through the **Import** sub–menu. These modules are documented in chapter *6. Import Modules*, on page 89.

Export
Export modules output an existing image. Export modules can be used to print to Mac OS printers that do not have Chooser–level driver support, or to save images in unsupported or compressed file formats. These modules are accessed through the **Export** sub–menu. These modules are documented in chapter *7. Export Modules*, on page 88.

Extension
Extension modules allow implementation of session-start and session-end features, such initializing devices. They are called once at application execution, once at application quit time, and usually have no user interface. Their interface is not public.

Filter
Filter modules modify a selected area of an existing image. These modules appear under the **Filter** menu. Filter modules are the plug-ins that the majority of Photoshop users are most familiar with. These modules are documented in chapter *8. Filter Modules*, on page 96.

Format
Format modules, also called File Format and Image Format modules, provide support for reading and writing additional image formats. These appear in the format pop–up menu in the **Open...**, **Save As...** and **Save a Copy...** dialogs. These modules are documented in chapter *9. Format Modules*, on page 122.

Parser
Parser modules are similar to Import and Export modules, and provide support for manipuling data between Photoshop and other (usually vector) formats such as Adobe Illustrator™ or Adobe® PageMaker™. Their interface is not public.

Selection
Selection modules modify which pixels are chosen in an existing image and can return either path or pixel selections. These modules appear under the **Selection** menu. These modules are documented in chapter *10. Selection Modules*, on page 108.

Plug–in module files

Plug–in module files must reside in specific directories for Adobe Photoshop to recognize them. Under the Mac OS, plug–in files must be in:

1. the same folder as the Adobe Photoshop application, or
2. the folder identified in the Photoshop preferences dialog, or
3. a sub–folder of the folder identified in the Photoshop prefs.

Under Windows, plug in files must be in the directory identified by the `PLUGINDIRECTORY` profile string in the Photoshop INI file.

Table 2-1: Names for the Photoshop INI file

Version	Filename
2.x	PHOTOSHP.INI
3.x	PHOTOS30.INI
4.x	PHOTOS40.INI
5.x	PHOTOS50.INI

Usually, a plug–in module file contains a single plug–in. You can create files with multiple plug–in modules. This is discouraged, because it reduces the user's control of which modules are installed.

There are situations when it may be appropriate to have more than one module in a single plug–in file. One example is matched import/export modules, although these are usually implemented as a file format module. Another example is a set of closely related filters, since the reduction of user control is offset by the increased ease of plug-in file management.

Plug–in file types and extensions

Plug–in module files should follow the guidelines in table 2-2 for the type identifier under the Mac OS, and the file extension under Windows. While these are only recommendations with Adobe Photoshop 3.0, these must be used if your plug–in module runs with earlier versions of Photoshop.

Table 2-2: Plug–in file types and extensions

Plug–in Type	Macintosh File Type	Windows File Extension
General (any type of plug–in)	`8BPI`	`.8BP`
Automation	`8LIZ`	`.8LI`
Color Picker	`8BCM`	`.8BC`
Import	`8BAM`	`.8BA`
Export	`8BEM`	`.8BE`
Extension	`8BXM`	`.8BX`
Filter	`8BFM`	`.8BF`

Table 2-2: Plug–in file types and extensions

Plug–in Type	Macintosh File Type	Windows File Extension
File Format	`8BIF`	`.8BI`
Parser	`8BYM`	`.8BY`
Selection	`8BSM`	`.8BS`

On the Macintosh, plug-ins with the same creator ID as Adobe Photoshop ('`8BIM`') will appear with the standard plug–in icons defined in Photoshop.

Basic data types

The basic types shown in table 2-3 are commonly used in the Photoshop plug–in API. Most of these are declared in `PITypes.h`.

Table 2-3: Basic data types

Name	Description
int8, int16, int32	These are 8, 16 and 32 bit integers respectively.
uint8, uint16, uint32, unsigned8, unsigned16, unsigned32	These are unsigned 8, 16 and 32 bit integers respectively.
short	Same as int16.
long	Same as int32.
Boolean	Single byte flag where 0=`FALSE`; any other value=`TRUE`.
OSType	`int32` denoting Mac OS style 4-character code like '`PiPL`'.
TypeCreatorPair	Two OSTypes denoting filetype then creator code.
FlagSet	Array of boolean values where the first entry is contained in the high order bit of the first byte. The ninth entry would be in the high–order bit of the second byte, etc.
CString	C-style string where the content bytes are terminated by a trailing `NULL` byte.
PString	Pascal style string where the first byte gives the length of the string and the content bytes follow.
Str255	Pascal style string where the first byte gives the length of the string and the content bytes follow, with a maximum of 255 content bytes.
Structures	Structures are typically represented the same way they would be in memory on the target platform. Native padding and alignment constraints are observed. Several common structures, such as `RGBtuple`, are declared in `PITypes.h`.
VPoint, VRect	Like Mac OS `Point` and `Rect` structures, but have 32–bit coordinates.

The plug–in module interface

A plug–in host calls a plug–in module in response to a user action. Generally, executing a user command results in a series of calls from the plug–in host to the plug–in module. All calls from the host to the module are done through a single entry point, the `main()` routine of the plug–in module. The prototype for the main entry point is:

```
#if MSWindows
void ENTRYPOINT (
    short    selector,
    void*    pluginParamBlock,
    long*    pluginData,
    short*   result);
#else
pascal void main (
    short    selector,
    Ptr      pluginParamBlock,
    long*    pluginData,
    short*   result);
#endif
```

selector
The *selector* parameter indicates the type of operation requested by the plug–in host. Selector=0 always means display an About box. Other selector values are discussed in later chapters for each type of plug–in module.

A plug-in's main function is typically a switch statement that dispatches the `pluginParamBlock`, `pluginData`, and `result` parameters to different handlers for each selector that the plug–in module responds to. The example plug–in modules show one style of dispatching to selector handlers.

pluginParamBlock
The *pluginParamBlock* parameter points to a large structure that is used to pass information back and forth between the host and the plug–in module. The fields of this parameter block changes depending on the type of plug–in module. Refer to chapters 6 through 9 for descriptions of the parameter block for each type of plug–in module.

pluginData
The *pluginData* parameter points to a long integer (32–bit value), which Photoshop maintains for the plug–in module across invocations.

One standard use for this field is to store a handle to a block of memory used to reference the plug–in's "global" data. It will be zero the first time the plug–in module is called.

result
The *result* parameter points to a short integer (16–bit value). Each time a plug–in module is called, it must set `result`; do not count on the `result` parameter containing a valid value when called. Returning a value of zero indicates that no error occurred within the plug–in module's code.

Error reporting

Returning a non–zero number in the `result` field indicates to the plug–in host that some sort of error occurred. It may also indicate that the user cancelled the operation somewhere during execution of the plug–in.

Returning a positive value indicates the plug–in encountered an error and an appropriate error message has already been displayed to the user. If the user cancels the operation in any way, the plug–in should return a positive value without reporting an error to the user.

Returning a negative value means that the plug–in encountered an error, and the plug–in host should display its standard error dialog.

Table 2-4 shows the common error code ranges used by the different types of plug–in modules, as well as some commonly used examples and their values. Refer to the header files and specific chapter for the plug-in type you're designing for more details.

Table 2-4: Error codes

Module	Error Range	Definitions	Value
Color Picker	–30800 to –30899	`pickerBadParameters`	-30800

Import	–30000 to –30099	`acquireBadParameters`
		`acquireNoScanner`
		`acquireScanner`
			-30000
			-30001
			-30002

Export	–30200 to –30299	`exportBadParameters` `exportBadMode`	-30200 -30201
Filter	–30100 to –30199	`filterBadParameters` `filterBadMode`	-30100 -30101
Format	–30500 to –30599	`formatBadParameters` `formatBadMode`	-30500 -30501
Selection	–30700 to –30799	`selectionBadParameters` `selectionBadMode`	-30700 -30701

General Errors	–30900 to –30999	`errPlugInHostInsufficient`
		`errPlugInPropertyUndefined`
		`errHostDoesNotSupportColStep`
		`errInvalidsamplePoint`
			-30900
			-30901
			-30902
			-30903

The plug-in may also return Mac and Windows operating system error codes to the plug–in host. In `PITypes.h`, several common Mac OS error codes are defined for use in Windows, simplifying cross–platform programming.

About boxes

All plug–ins should respond to a selector value of zero, which means display an *About* box. The about box may be of any design. To fit in smoothly with the Adobe Photoshop interface, follow these conventions:

1. The About box should be centered on the main (menu bar) screen, with 1/3 of the remaining space above the dialog, and 2/3 below. Be sure to take into account the menu bar height. System 7 or later of the Mac OS has a flag in the '`DLOG`' resource that automatically positions the about box in the window.

2. The window should not have an **OK** button, but should instead respond to a click anywhere in its dialog.

3. It should respond to the *return* and *enter* keys.

Note: Adobe PhotoDeluxe has very specific dialog design requirements. A sample About box is available in the `Photoshop-WDEF` folder in the `Examples` folder.

The parameter block at selectorAbout

On the about selector call, the parameter block for the module is not passed. Instead, in its place, a structure of type `AboutRecord` is passed. This is

described in `PIAbout.h`. Because of this, access to any of the standard parameters for the module during `selectorAbout` is unavailable.

Multiple plug-ins and selectorAbout
When Photoshop attempts to bring up the about box for a plug–in module, it makes the about box selector call to each of the plug–ins in the same file. If there is more than one plug–in compiled in a file, only one of them should respond to the call by displaying an about box that describes all the plug-ins. All other plug–in modules should ignore the call and return to the plug–in host.

Memory management strategies

In most cases, the first action a plug–in takes is to negotiate with Photoshop for memory. Other plug–in hosts may not support the same memory options.

The negotiation begins when Photoshop sets the *maxData* or *maxSpace* field of the `pluginParamBlock` to indicate the maximum number of bytes it would be able to free up. The plug–in then has the option of reducing this number. Reserving memory by reducing `maxData` can speed up most plug-in operations. Requesting the maximum amount of memory for the plug-in requires Photoshop to move all current image data out of of RAM and into its virtual memory file. This allows the plug-in to run from RAM as much as possible.

If your plug–in's memory requirements are small—if it can process the image data in pieces, or if the image size is small—only reduce `maxData`/`maxSpace` to those specific requirements. This permits many plug–in operations to be performed entirely in RAM with a minimum of swapping. In many cases, your plug–in only needs a small amount of memory, but will operate faster if given more. Experiment to find a suitable balance.

One strategy is to divide `maxData`/`maxSpace` by 2, thus allocating half the memory to Photoshop and half to the plug–in. Another good strategy is to reduce `maxData`/`maxSpace` to zero, and then use the Buffer and Handle suites to allocate memory as needed. Often, this is most efficient from Photoshop's viewpoint, but requires additional programming.

If performance is a concern, you may want to perform quantitative tests of your plug–in module to compare different memory strategies.

maxSpace vs. bufferSpace on the Macintosh

Photoshop has a couple different mechanisms for reserving memory. There are also mechanisms for reporting available memory. Together, they allow you to calculate what sort of processing parameters you will need to use, such as chunk sizes, etc. This section is designed to detail two specific fields that indicate available space: `maxSpace` and `bufferSpace`, from the filter parameter block.

The amount of free space available in the Macintosh heap is returned in `maxSpace`. Photoshop uses its own linear bank code when the available memory goes over a threshold, generally around 32 mb. This is done because the Mac Memory manager gets very inefficient with large heaps.

Photoshop sets aside an area of memory, called the *linear bank*. The Mac OS sets aside an area of memory for the application, called the *heap space*. `BufferSpaceProc()` in the Buffer suite, and `bufferSpace` in the `filterParamBlock` return the amount of space available in the linear bank, or the heap space if the linear bank is not present. This memory is available via the Buffer suite.

`bufferSpace` and `max`Space will be about the same up to 32 mb (they both reserve different amounts of padding) and then `maxSpace` will step back. `bufferSpace` will also step back, but grow as memory is available.

Neither `maxSpace` nor `bufferSpace` guarantee continguous space.

Table 2-5: Photoshop memory and maxData/maxSpace

Photoshop Memory	maxData/maxSpace
1 mb	4 mb
16 mb	6 mb
26 mb	14 mb
31 mb	19 mb
32 mb+	9 mb

Creating plug–in modules for the Mac OS

Photoshop plug–in modules for the Macintosh can be created using any of the popular C compilers including Apple MPW, Symantec C++, or Metrowerks CodeWarrior. The example plug–ins in this toolkit include both MPW makefiles and CodeWarrior project files.

You can create plug–in modules for 680x0, PowerPC, or both (fat binaries). If your plug–in module uses floating point arithmetic, you can create plug–in code that is optimized for Macintosh systems with floating–point units (FPU). If you desire, you can also provide a version of your code that does not require an FPU, and Photoshop will execute the proper version depending on whether an FPU is present.

Plug–in modules use code resources on 680x0 Macs and shared libraries (the code fragment manager) on PowerPC systems.

When the user performs an action that causes a plug–in module to be called, Photoshop opens the resource fork of the file the module resides in, loads the code resource (68k) or shared library (PowerPC) into memory. On 680x0 systems, the entry point is assumed as the first byte of the resource.

Hardware and system software configuration

Adobe Photoshop plug–ins assume that the Macintosh has 128K or larger ROMs, System 6.0.2 or later. Photoshop 3.0 and later requires System 7.

Photoshop 5.0 and later are PPC only.

Many users still work with older versions of Photoshop. If you choose to support versions of Photoshop prior to 3.0, your plug–in may be called from machines as old as the Mac Plus. You should use the Gestalt routines to check for 68020 or 68030 processors, math co–processors, 256K ROMs, and Color or 32–Bit QuickDraw if they are required.

If your plug-in only runs with Photoshop 3.0, you can assume the features are present that are requirements of Photoshop 3.0: a 68020 or better, Color QuickDraw, and 32–Bit QuickDraw.

Resources in a plug–in module

Besides 680x0 code resources, a plug–in module may include a variety of resources for the plug–in's user interface, stored preferences, and any other useful resource.

Every plug–in module must include either a complex data structure stored in a `PiPL` resource or a simpler structure in a `PiMI` resource. These resources provide information that Adobe Photoshop uses to identify plug–ins when Photoshop is first launched and when a plug–in is executed by the user. All the examples in this toolkit build both resources for downward- and cross-application compatibility.

`PiMI`s, `PiPL`s and other resources are discussed in detail in the document *Plug-in Resource Guide.pdf* which is included with this kit. *Plug-in Resource Guide.pdf* discusses cross-application plug-in development and describes the different file and code resources recognized by Photoshop and other host applications.

Global variables

Most Macintosh applications reference global variables as negative offsets from register A5 on 680x0 processors. If a plug–in module declares any

global variables, their location would overlap Photoshop's global variable space. Using them generally results in a quick and spectacular crash.

Often you can end up using them without even realizing it. Explicit literal string assignments, for instance, take up global space when first initialized. One way around this is to store strings in a 'STR ' or 'STR#' resource and access them using the Macintosh toolbox calls `GetString()` and `GetIndString()`.

Metrowerks CodeWarrior A4-globals
Code resources can avoid the A5 problem by using the A4 register in place of the A5 register. The Metrowerks CodeWarrior C compiler contains header files (`SetupA4.h`, `A4Stuff.h`) and pre–compiled libraries designed for A4 register usage. The examples in this toolkit all initialize and set-up the A4 register, so you can refer to them for more detail.

If you are building a plug–in module to run on 680x0 systems you should not declare any global variables in your plug–in module code unless you specifically use the A4 support provided with your compiler. Refer to your compiler documentation for more details.

Plug–in modules that are compiled native for PowerPC systems do not have this limitation, since they use the code fragment manager (CFM) instead of code resources. If your plug–in module only runs on PowerPC, you may safely declare and use global variables. Refer to the appropriate Apple documentation for more information.

If you need global data in your 680x0 compatible plug–in module, one alternative to using A4 is to dynamically allocate a new block of memory at initialization time using the Photoshop Handle or Buffer suite routines, and return this to Photoshop in the `data` parameter. Photoshop will save this reference and return it to your plug–in each time it is called subsequently. The example plug-ins in this toolkit all use this approach.

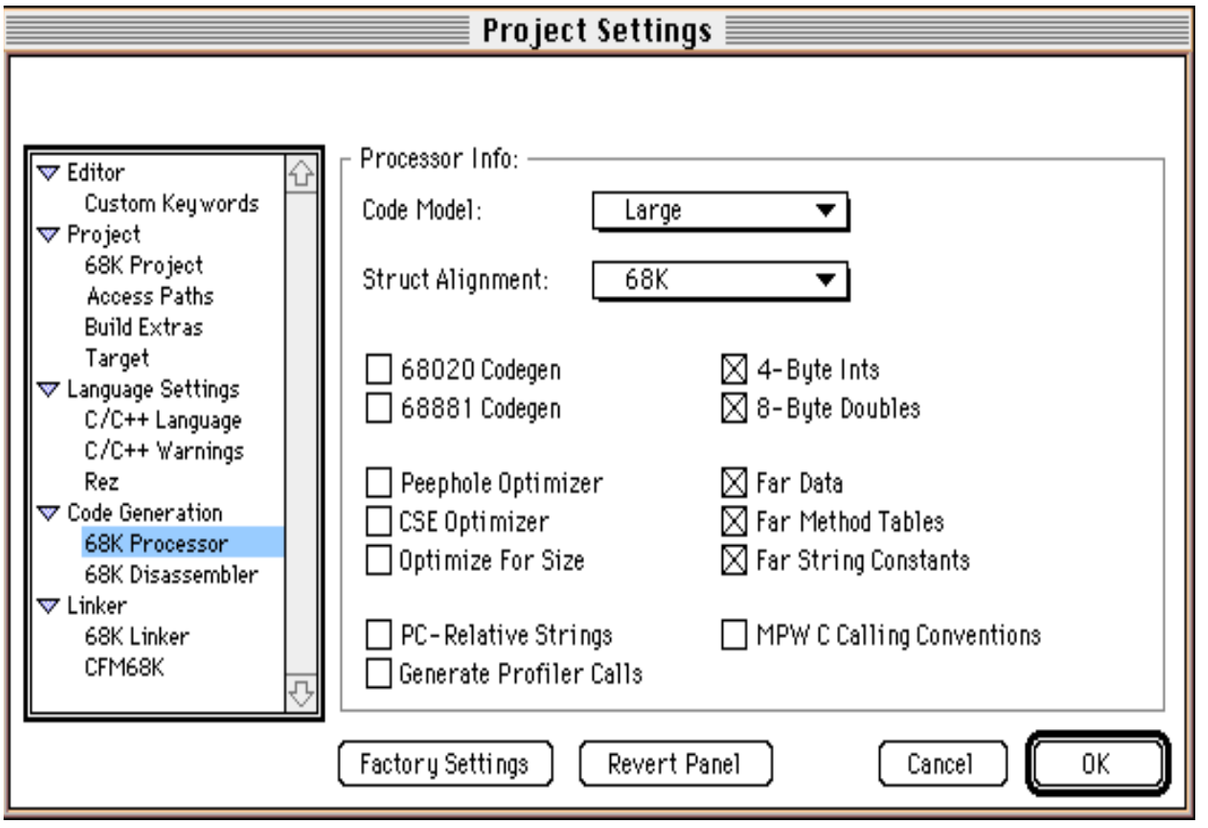
Segmentation

Macintosh 680x0 applications have a special code segment called the jump table. When a routine in one segment calls a routine in another segment, it actually calls a small glue routine in the jump table segment. This glue routine loads the routine's segment into memory if needed, and jumps to its actual location.

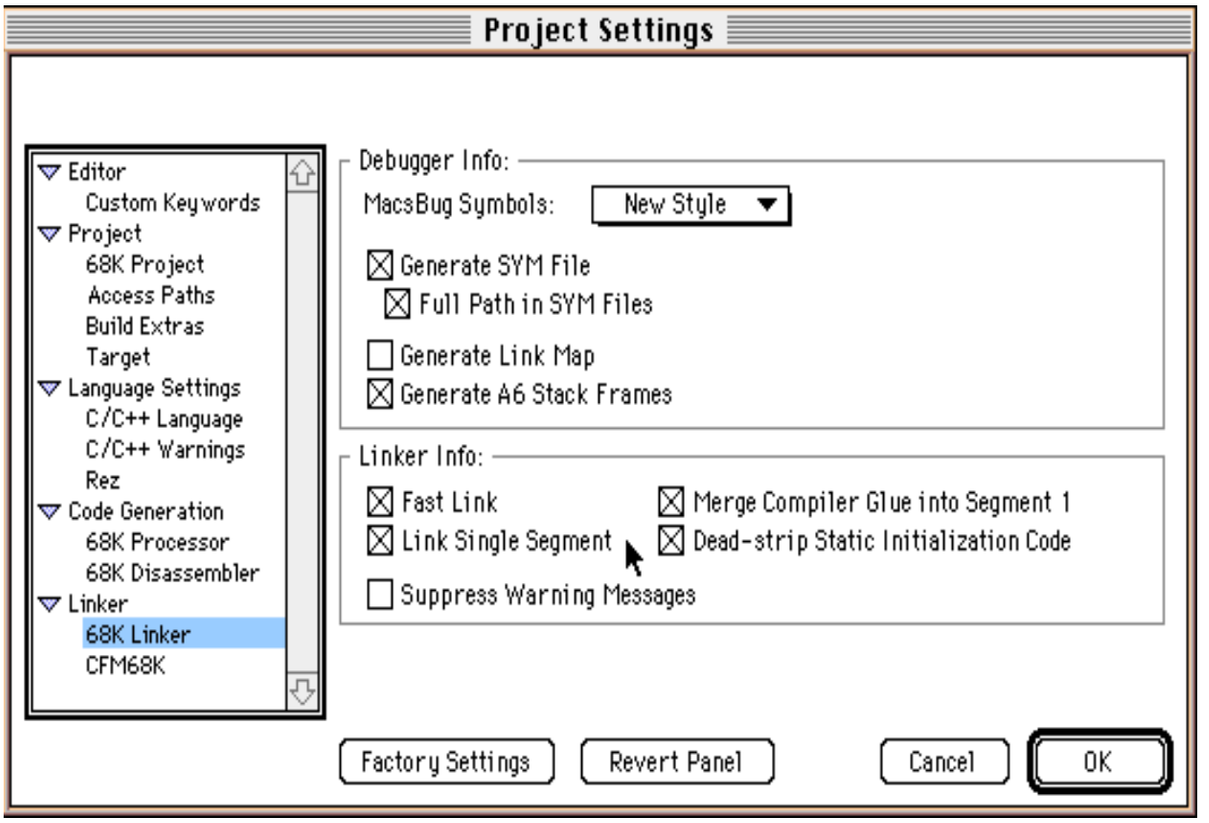
The jump table is accessed using positive offsets from register A5. Since Photoshop is already using A5 for its jump table, the plug–in cannot use a jump table in the standard way.

The simplest way to solve this is to link all the plug–in's code into a single segment. This usually requires setting optional compilation/link flags in your development environment if the resultant segment exceeds 32k.

Metrowerks CodeWarrior link flags for plug-ins over 32k
For over 32k length plug-ins under Metrowerks CodeWarrior, enable these processor preferences: *code model: large; far data; far method tables;* and *far string constants*:



Enable the linker preference *link single segment*:



Installing plug–in modules

To install a plug–in module, drag the module's icon to either the same folder as the Adobe Photoshop application, or the plug–ins folder designated in your Photoshop preferences file. Photoshop 3.0 searches for plug–ins in the application folder, and throughout the tree of folders underneath the designated plug–ins folder. Aliases are followed during the search process. Folders with names beginning with "¬" (Option–L on the Macintosh keyboard) are ignored.

Finding the plug-in directory for the Mac OS

To find the plug-in directory on Mac OS in versions of Photoshop before 5.0 you have to search for the application and try to find a folder called `Plug-ins`. In Photoshop 5.0 and later:

1. In the `System Folder`, in the `Preferences` folder is a file with paths to the application and plug-ins folder:

Table 2-6: Photoshop 5 Paths filenames

Application	Paths filename
Adobe Photoshop 5.0	Adobe Photoshop 5 Paths
Adobe Photoshop 5.0 Limited Edition	Adobe Photoshop LE Paths
Adobe Photoshop 5.0 Tryout/Demo	Adobe Photoshop 5 Demo Paths

2. The Photoshop 5 Paths file contains two Macintosh `FSSPec` structures one right after the other. The first is to the application; the second is to the plug-in directory. There is no padding or sentinal characters.

What's in this toolkit for the Mac OS?

This toolkit contains documentation, and literature on the Adobe Developers Association, and examples specifically written for the Mac OS.

Examples

The plug–ins included with this toolkit can be built using Apple MPW or Metrowerks CodeWarrior. They have been tested against the latest Metrowerks CodeWarrior. Version notes are in the SDK `Readme` file.

Sources.c and Headers.h

`PIGeneral.h` and `PITypes.h` contain definitions useful across multiple plug–ins. `PIAbout.h` contains the information for the about box call for all plug–in types. `PIActions.h` contains the information for the Actions suite callbacks for all plug–in types. `PIAcquire.h`, `PIExport.h`, `PIFilter.h`, `PIFormat.h`, `PIPicker.h`, and `PISelection.h` are the header files for the respective types of plug–in modules.

Utilities

`DialogUtilities.c` and `DialogUtilities.h` provide general support for doing things with dialogs including creating movable modal dialogs which make appropriate calls back to the host to update windows, as well as simple support for putting data back into the dialog for display, such as `StuffNumber()` and `StuffText()`.

`PIUtilities.c` and `PIUtilities.h` contain various routines and macros to make it easier to use the host callbacks. The macros make assumptions about how global variables are being handled and declared; refer to the example source code to see how `PIUtilities` is used.

Documentation

Photoshop SDK Guide.pdf is this guide. *Plug-in Resource Guide.pdf* is a reference tool for developing Photoshop plug-ins that work with all of Adobe's major graphical applications. It also includes information on host applications and their use of different code and file resources such as `PiMI` and `PiPL` resources. *Actions Event Guide.pdf* is a reference tool for developing a Photoshop 5.0 automation plug-in.

Developer Services

The Developer Services directory provides information and an application for the Adobe Developers Association, which provides not only support for this

and the other Adobe toolkits, but marketing and business resources for third party developers.

Fat and PPC-only plug-ins and the cfrg resource

Adobe Photoshop 4.0 and 5.0 use the `PiPL` resource (see *Plug-in Resource Guide.pdf*) to identify the type of processor for which the plug-in module was compiled: 680x0, PowerPC or both. The Macintosh OS uses a different resource, '`cfrg`', to indicate the presence of code for the PowerPC microprocessor. The `cfrg` resource is automatically generated by the Metrowerks CodeWarrior development environment.

Normally, this is not a problem. It could become a problem, however, if you or a user of your plug–in run an application to reduce a fat binary (680x0 and PowerPC) plug–in to 680x0 only. Fat stripper applications search for `cfrg` resources and when found remove any PowerPC code and the `cfrg` resource. These applications are not aware of the `PiPL` resource; the resulting 680x0–only plug–in will still indicate that it contains PowerPC code.

After you create a PowerPC plug–in module, you should manually remove the `cfrg` resource with a resource editor to prevent someone from accidently deleting the PowerPC code by stripping.

You should always be sure to specify the correct `PiPL` code descriptors when building a plug–in. All of the plug–ins in the examples folder have `PiPL` resources with both code descriptors, as follows:

```
#if Macintosh
    Code68K { '8BIF', $$ID },
    CodePowerPC { 0, 0, "" },
#endif
```

If your plug-in module includes code only for the 680x0 or only for the PowerPC, remove the other code descriptor before compiling the `.r` file. For instance, a PowerPC–only plug–in module's `PiPL` source file would have these lines in the `PiPL` descriptor:

```
#if Macintosh
    CodePowerPC { 0, 0, "" },
#endif
```

Building your plug-in in older IDE than CodeWarrior Pro

Metrowerks CodeWarrior Pro introduced multiple targets. You can now build 68K and PPC code from within one project.

Building your plug-ins with Metrowerks CodeWarrior in older IDEs is a two-step process. Open and build the 680x0 project to create the `.rsrc` and 68k code for the plug-in. Then, you must open and build the PowerPC project to create the actual plug-in module and roll the 68k and PPC resources together in the plug-in.

If you do not build the 680x0 project, the initial `.rsrc` resource file with the appropriate dialog and `PiPL` resources will not be built. If you do not build the PowerPC project, the actual plug-in file will not be created. To build from one project file (68k-only) see the following information for CodeWarrior Bronze users.

Building 680x0–only plug-ins

The sample CodeWarrior project files in this toolkit are designed for CodeWarrior Gold to create "fat" binaries. If you use CodeWarrior to build 680x0–only plug–in modules, you should make two changes to the sample files.

First, you should change the Project preferences to output a plug–in file with the correct file name, creator, and type. (The 68K project files included in the toolkit output resource files which are then used by the PPC project files, as explained above.)

For example, the 680x0 project in the `Filters` sample is set to output a code resource named "`Dissolve.rsrc`" with creator '`Doug`' and type '`RSRC`'. You should change these to "`Dissolve`", '`8BIM`', and '`8BFM`' respectively.

Second, you should recompile the `PiPL` resource after removing the PowerPC code descriptor. The `PiPL` statement:

```
#if Macintosh
    Code68K { '8BIF', $$ID },
    CodePowerPC { 0, 0, "" },
#endif
```

should be changed to:

```
#if Macintosh
    Code68K { '8BIF', $$ID },
#endif
```

See *Plug-in Resource Guide.pdf* for more information about `PiPL` resources.

Debugging code resources in Metrowerks CodeWarrior

To use the Metrowerks debugger with Photoshop 680x0 plug-ins:

1. Drag the `.SYM` file out of the development directory where it is linked to the `.rsrc` file. The desktop is fine.
2. Double-click the `.SYM` file to run the Metrowerks Debugger.
3. When it asks "Where is my resource?" select your plug-in in the Photoshop plug-in folder. Even if your plug-in is an alias, select the one that is in Photoshop's Plug-ins folder, *not* the one that may be sitting in your development directory (next to your `.rsrc` file).
4. Drag the Photoshop icon on top of the Metrowerks Debugger to link Photoshop to the debugger.
5. Double-click the Photoshop icon to launch the application.
6. Set your breakpoints in your `.SYM` window in the debugger.
7. Bring the Photoshop debugger window to the front and choose **Run**.
8. In Photoshop, run your plug-in. You should hit your break-point and go back to the debugger automatically.

Note:
Your variables may not read their true values correctly as you step through your code. They may be valid only at your breakpoints.

Creating plug–in modules for Windows

Photoshop plug–ins for Windows can be created using Microsoft® Visual C++, version 2.0 or later (which requires Windows NT version 3.5 or later, or Windows 95). This toolkit has been checked under Visual C++ 5.0 and Windows NT 4.0.

When the user performs an action that causes a plug–in module to be called, Photoshop does a `LoadLibrary` call to load the module into memory. For each `PiPL` resource found in the file, Photoshop calls `GetProcAddress(`*routineName*`)` where *routineName* is the name associated with the `PIWin32X86CodeProperty` property to get the routine's address.

If the file contains only `PiMI` resources and no `PiPL`s, Photoshop does a `GetProcAddress` for each `PiMI` resource found in the file looking for the entry point `ENTRYPOINT`*%* where *%* is the integer `nameID` of the `PiMI` resource to get the routine's address.

Hardware and software configuration

Adobe Photoshop plug–ins may assume Windows 3.1 in standard or enhanced mode, Windows NT 3.5 or later, or Windows 95. Adobe Photoshop requires at least an 80386 processor.

Photoshop 5.0 and later does not support Windows 3.1.

For development, you must have Windows NT or Windows 95. You cannot create Windows plug–ins with this toolkit under Windows 3.1.

Structure packing

Structure packing for all plug–in parameter blocks, `FilterRecord`, `FormatRecord`, `AcquireRecord`, `ExportRecord`, `SelectionRecord`, `PickerRecord` and `AboutRecord`, should be the default for the target system. The Info structures such as `FilterInfo` and `FormatInfo` must be packed to byte boundaries. The `PiMI` resource should be byte aligned.

These packing changes are reflected in the appropriate header files using `#pragma pack(1)` to set byte packing and `#pragma pack()` to restore default packing. These pragmas work only on Microsoft Visual C++ and Windows 32 bit SDK environment tools. If you are using a different compiler, such as Symantec C++ or Borland C++, you must modify the header files with appropriate pragmas. The Borland `#pragmas` still appear in the header files as they did in the 16–bit plug–in kit, but are untested.

Resources

The notion of resources is central to the Macintosh, and this carries through to Photoshop. The `PiPL` resource (described in *Plug-in Resource Guide.pdf*) introduced with Photoshop 3.0 and the older `PiMI` resource are declared in Macintosh Rez format in the file `PIGeneral.r`.

Windows has a similar notion of resources, although they are not the same as on the Macintosh.

Creating or modifying PiPL resources in Windows

Even under Windows, you are encouraged to create and edit `PiPL` resources in the Macintosh format, and then use the `CNVTPIPL.EXE` utility. For a complete discussion of creating or modifying `PiPL` resources in Windows-only development environments, refer to the *Plug-in Resource Guide.pdf*.

Calling a Windows plug–in

You need a `DLLInit()` function prototyped as

```
BOOL APIENTRY DLLInit(HANDLE, DWORD, LPVOID);
```

The actual name of this entry point is provided to the linker by the

```
PSDLLENTRY=DLLInit@12
```

assignment in the sample makefiles.

The way that messages are packed into `wParam` and `lParam` changed for Win32. You will need to insure that your window procedures extract the appropriate information correctly. A new header file `WinUtil.h` defines all the Win32 message crackers for cross–compilation or you may simply change your extractions to the Win32 versions. See the Microsoft document, *Win32 Application Programming Interface: An Overview* for more information on Win32 message parameter packing.

Be sure that the definitions for your Windows callback functions such as dialog box functions conform to the Win32 model. A common problem is to use of `WORD wParam` for callback functions. The plug–in examples use

```
BOOL WINAPI MyDlgProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
```

which will work correctly for both 16 and 32 bit compilation.

Installing plug–in modules

To install a plug–in module, copy the `.8B*` files into the directory referred to in the Photoshop INI file with the profile string `PLUGINDIRECTORY`.

When Adobe Photoshop first executes, it searches the files in the `PLUGINDIRECTORY`, looking for plug–in modules. When it finds a plug–in, it checks its version number. If the version is supported, it adds name of the plug–in to the appropriate menu or to the list of extensions to be executed.

Each kind of plug–in module has its own 4–byte resource type. For example, acquisition modules have the code `8BAM`.

The actual resource type must be specified as `_8BAM` in your resource files to avoid a syntax error caused by the first character being a number.

For example, Adobe Photoshop searches for Import modules by examining the resources of all files in `PLUGINDIRECTORY` with file extension `.8B*` for resource type `_8BAM`. For each `8BAM`, the integer value which uniquely identifies the resource, `nameID`, must be consecutively numbered starting at 1.

Finding the plug-in directory in Windows

To find the plug-in directory in Windows in versions of Photoshop before 5.0:

1. Do a registry search for the `Photoshp.exe` key:
   `HKEY_LOCAL_MACHINE\SOFTWARE\Adobe\Photoshop`
2. The directory that Photoshop is in will have the folder `Prefs` which contains the Photoshop INI file.
3. Open the Photoshop INI file and do a search for the `PLUGINDIRECTORY` tag.

4. If the tag exists, it will return the path to the plug-in folder. If the tag does not exist, you can assume the plug-in folder is the default: in the same folder with the Photoshop executable under the name `Plugins`.

To find the plug-in directory in Windows in Adobe Photoshop 5.0 and later:

1. Do a registry search for the `Photoshp.exe` key: `HKEY_LOCAL_MACHINE\SOFTWARE\Adobe\Photoshop`

2. The directory that Photoshop is in will have a directory, `Adobe Photoshop Settings`. That folder contains the Photoshop INI file.

3. Open the Photoshop INI file and do a search for the `PLUGINDIRECTORY` tag.

4. If the tag exists, it will return the path to the plug-in folder. If the tag does not exist, you can assume the plug-in folder is the default: in the same folder with the Photoshop executable under the name `Plug-Ins`.

Or, easier:

1. Do a registry search for the `PlugInPath` key: `HKEY_LCOAL_MACHINE\SOFTWARE\Adobe\Photoshop\5.0\PlugIn-Path`

2. If the tag exists, it will return the path to the plug-in folder. If the tag does not exist, you can assume the plug-in folder is the default: in the same folder with the Photoshop executable under the name `Plug-Ins`.

What's in this toolkit for Windows?

This toolkit contains documentation, and literature on the Adobe Developers Association, and examples specifically written for Windows.

Examples

The sample plug–ins included with this toolkit can be built using Visual C++ 2.0 or Visual C++ 4.0. There are project files for both compilers. The Visual C++ 4.0 project file is the same name as the example with "`40`" after it (such as `Dissolve40.mdp`).

Sources.c and Headers.h

`WinUtils.c` provides support for some Mac Toolbox functions used in `PIUtilities.c`, including memory management functions such as `NewHandle()`. The header file `PITypes.h` contains definitions for common Mac result codes, data types, and structures. These simplify writing plug–in modules for both the Mac OS and Windows. `PIAbout.h` contains the information for the about box call for all plug–in types. `PIActions.h` contains the information for the Actions suite callbacks for all plug–in types. `PIAcquire.h`, `PIExport.h`, `PIExtension.h`, `PIFilter.h`, `PIFormat.h`, `PIPicker.h`, and `PISelection.h` are the header files for the respective types of plug–in modules.

Utilities

`WinDialogUtils.c` and `WinDialogUtils.h` provide general support for doing things with dialogs including creating movable modal dialogs which make appropriate calls back to the host to update windows, as well as simple support for putting data back into the dialog for display, such as `StuffNumber()` and `StuffText()`.

`PIUtilities.c` and `PIUtilities.h` contain various routines and macros to make it easier to use the host callbacks. The macros make assumptions about how global variables are being handled and declared; refer to the sample source code to see how `PIUtilities` is used.

The Windows version of this toolkit also includes two handy utility programs: MACTODOS.EXE and CNVTPIPL.EXE.

MACTODOS.EXE converts Macintosh text files into PC text files by changing the line ending characters.

CNVTPIPL.EXE converts PiPL resources in Macintosh Rez format (ASCII format which conforms to the PiPL resource template) into the Windows PiPL format. Refer to *Plug-in Resource Guide.pdf* for more information about PiPL resources.

To use CNVTPIPL.EXE, you need to pre–process your *plugin*.r file using the standard C pre–processor and pipe the output through CNVTPIPL.EXE. The example makefiles illustrate the process.

Documentation
Photoshop SDK Guide.pdf is this guide. *Plug-in Resource Guide.pdf* is a reference tool for developing Photoshop plug-ins that work with all of Adobe's major graphical applications. It also includes information on host applications and their use of different code and file resources such as PiMI and PiPL resources.

Developer Services
The Developer Services directory provides information and an application for the Adobe Developers Association, which provides not only support for this and the other Adobe toolkits, but marketing and business resources for third party developers.

3. Plug-in Host Callbacks

Plug–in hosts execute plug–in modules by calling the module's main entry point, passing a selector, parameter block, and pointer to the module's data.

Plug–in modules can make calls back to the plug–in host by means of callback function pointers that are provided in the plug–in's parameter block. These callbacks provide specific services that your plug–in module may need. This chapter discusses these callbacks and how to use them.

Callbacks fall into two categories: callback pointers that are hard–coded into the parameter block structures (direct callbacks), and callbacks that are accessed through callback suites.

Some of these callback routines are new and may not be provided by other plug–in hosts, including earlier versions of Photoshop. If a host does not provide a particular routine or suite, the relevant pointer will be `NULL`.

Photoshop 3.0 and above includes an error code to indicate that the host does not supply necessary functionality:

```
#define errPlugInHostInsufficient  -30900
```

Under the Mac OS, callback functions use Pascal calling conventions; Windows callbacks use C calling conventions. In the following function prototypes, this is indicated by the macro "`MACPASCAL`".

A complete list of callback function declarations can be found in `PIGeneral.h` and `PIActions.h`.

Note: If a host does not provide a particular routine or suite, the relevant pointer in the parameter block will be `NULL`.

Direct callbacks

These callbacks are found directly in the various plug–in parameter block structures.

AdvanceStateProc( )

```
MACPASCAL OSErr (*AdvanceStateProc) (void);
```

This callback provides a more efficient way for plug–ins to interact with a host. The plug–in asks the host to update, "*advance* the *state* of," the various data structures used for communicating between the host and the plug–in.

Use `AdvanceStateProc` where you expect your plug–in to be called repeatedly. An example is a scanner module that scans and delivers images in chunks. When working with large images (larger than available RAM), plug–ins should process the image in pieces.

Without `AdvanceStateProc`, your plug-in is called from, and returns to, the host for each chunk of data. Each repeated call must go through your plug–in's `main()` entry point.

With `AdvanceStateProc`, your plug–in can complete its general operation within a single call from the plug–in host. This does not include setup interaction with the user, or normal clean-up.

The plug–in host returns `noErr` if successful and a non–zero error code if something went wrong. If an error is returned, you should not call `AdvanceStateProc` again, but instead return the error code to the plug–in host back through `main()`.

The precise behavior of this callback varies depending on what type of plug–in module is executing. Refer to chapters 6–9 on specific plug–in types for information on how to use this callback.

The `AdvanceStateProc` callback is available in Adobe Photoshop version 3.0 and later.

AdvanceState, Buffers, Proxies, and DisplayPixels
Proxies really put AdvanceState to work, because if you allow your user to drag around your image, they're constantly updating and asking for more image data. To keep your lag time down, and the update watch from appearing often in `DisplayPixels`, keep these items in mind:

1. AdvanceState buffers as much of your image as it can, so make your first call for your inRect as large as you can. In subsequent calls, as long as you're within inRect, the image data will come right out of the buffer.

2. As soon as you set `inRect=0,0,0,0` or you call for one pixel outside of the buffer area (the first calling rect you passed) AdvanceState will flush the buffer and load new image data from the VM system.

AdvanceState error codes
If the user cancels by pressing *Escape* in Windows or *Command-period* on Macintosh, `AdvanceState` will return `userCanceledErr` (-128) and `inData` and `outData` will both return `NULL`, no matter what `inRect` or `outRect` is requested.

ColorServicesProc( )

```
MACPASCAL OSErr (*ColorServicesProc) (ColorServicesInfo *info);
```

This callback provides your plug–in module access to common color services within Photoshop. It can be used to perform one of four operations:

1. choose a color using the user's preferred color picker (Photoshop's, the Systems, or any Color Picker plug-in module),
2. convert color values from one color space to another,
3. return the current sample point,
4. return either the foreground or background color.

Refer to Appendix A for the `ColorServicesInfo` structure. Refer to Chapter 4 for the Color Picker plug-in module type.

Note: `ColorServices` has a bug in versions of Photoshop prior to 3.0.4. When converting from one color space to another they return error `paramErr=-50` and convert the requested color to RGB, regardless of the target color space.

`ColorServices` also may have errors converting between any color space and Lab, XYZ, or HSL. RGB, CMYK, and HSB have been proven and are correct, but we caution you check the numbers on the others before relying on them.

DisplayPixelsProc( )

```
MACPASCAL OSErr (*DisplayPixelsProc) (const PSPixelMap *source,
    const VRect *srcRect, int32 dstRow, int32 dstCol,
    unsigned32 platformContext);
```

This callback routine is used to display pixels in various image modes. It takes a structure describing a block of pixels to display.

The routine will do the appropriate color space conversion and copy the results to the screen with dithering. It will leave the original data intact. If it is successful, it will return `noErr`. Non–success is generally due to unsupported color modes.

To suppress the watch cursor during updates, see `propWatchSuspension` in the Properties suite.

source
The *source* parameter points to a `PSPixelMap` structure containing the pixels to be displayed. This structure is documented in Appendix A.

srcRect
The *srcRect* parameter points to a `VRect` that indicates the rectangle of the source pixel map to display.

dstRow / dstCol
The *dstRow* and *dstCol* parameters provide the coordinates of the top-left destination pixel in the current port (i.e., the destination pixel which will correspond to the top-left pixel in `srcRect`). The display routines do not scale the pixels, so specifying the top left corner is sufficient to specify the destination.

platformContext
The *platformContext* parameter is not used under the Mac OS since the display routines simply assume that the target is the current port. On Windows, `platformContext` should be the target `hDC`, cast to an `unsigned32`.

HostProc( )

```
MACPASCAL void HostProc(int16 selector, int32 * data);
```

This callback contains a pointer to a host–defined function. Plug–ins should verify the host's signature (in the parameter block's `hostSig` field) before calling this procedure. This provides a mechanism for hosts to extend the plug–in interface to support application specific features.

Adobe Photoshop version 3.0.4 and later does not perform any tasks in this callback. Earlier versions of Photoshop used `Host` for private communication between Photoshop and some plug–in modules.

ProcessEventProc( )

```
MACPASCAL void (*ProcessEventProc) (EventRecord *event);
```

This callback is only useful under the Mac OS; *ProcessEvent* in the Windows version of Adobe Photoshop does nothing.

Adobe Photoshop provides this callback function to allow Macintosh plug–in modules to pass standard `EventRecord` pointers to Photoshop. For example, when a plug–in receives a deactivate event for one of Photoshop's windows, it should pass this event on to Photoshop.

This routine can also be used to force Photoshop to update its own windows by passing relevant update and `NULL` events.

SpaceProc( )

```
MACPASCAL int32 SpaceProc (void);
```

This callback examines `imageMode`, `imageSize`, `depth`, and `planes` and returns the number of bytes of scratch disk space required to hold the image. Returns -1 if the values are not valid.

This callback is only available to Acquire plug–in modules.

TestAbortProc( )

```
MACPASCAL Boolean (*TestAbortProc) ( );
```

Your plug–in should call this function several times a second during long operations to allow the user to abort the operation. If the function returns `TRUE`, the operations should be aborted. As a side effect, this changes the cursor to a watch and moves the watch hands periodically.

UpdateProgressProc( )

```
MACPASCAL void (*UpdateProgressProc) (long done, long total);
```

Your plug–in may call this two–argument procedure periodically to update a progress indicator. The first parameter is the number of operations completed; the second is the total number of operations.

This procedure should only be called in the actual main operation of the plug–in, not while long operations are executing during the preliminary user interface.

Photoshop automatically suppresses display of the progress bar during short operations.

Callback suites

Note:
Photoshop 5.0 contains at least a dozen brand new suites that are documented in the next chapter. This chapter documents Photoshop 4.0 and earlier callbacks, which are still supported in the old parameter blocks.

The rest of the callback routines are organized into “suites,” collections of related routines which implement a particular functionality. The suites are described by a pointer to a record containing:

1. a 2 byte version number for the suite,
2. a 2 byte count of the number of routines in the suite,
3. a series of function pointers for the callback routines.

Before calling a callback defined in the suite, the plug–in needs to check the following conditions:

1. The suite pointer must not be `NULL`.
2. The suite version number must match the version number the plug–in wishes to use. (Adobe does not expect to change suite version numbers often.)
3. The number of routines defined in the suite must be great enough to include the routine of interest.
4. The pointer for the routine of interest must not be `NULL`.

If these conditions are not met, your plug–in module should put up an error dialog to alert the user and return a positive result code.

The suites that are currently implemented in the parameter blocks by Adobe Photoshop 5.0 are:

- the Buffer suite
- the Channel Ports suite
- the Descriptor suite
- the Handle suite
- the Image Services suite
- the Property suite
- the Pseudo–Resource suite

These are described next.

Buffer suite

Current version: 2; Adobe Photoshop: 5.0; Routines: 5.

The Buffer suite provides an alternative to the memory management functions available in previous versions of Photoshop's plug–in specification. It provides a set of routines to request that the host allocate and dispose of memory out of a pool which it manages.

Photoshop 2.5, for example, goes to a fair amount of trouble to balance the need for buffers of various sizes against the space needed for the tiles in its virtual memory system. Growing the space needed for buffers will result in Photoshop shrinking the number of tiles it keeps in memory.

Previous versions of the plug–in specification provide a simple mechanism for interacting with Photoshop's virtual memory system by letting a plug–in specify a certain amount of memory which the host should reserve for the plug–in.

This approach has two problems. First, the memory is reserved throughout the execution of the plug–in. Second, the plug–in may still run up against limitations imposed by the host. For example, Photoshop 2.5 will, in large memory configurations, allocate most of memory at startup via a `NewPtr` call, and this memory will never be available to the plug–in other than through the Buffer suite. Under Windows, Photoshop's memory scheme is designed so that it allocates just enough memory to prevent Windows' virtual memory manager from kicking in.

If a plug–in module allocates lots of memory using `GlobalAlloc` (Windows) or `NewPtr` (Mac OS), this scheme will be defeated and Photoshop will begin double–swapping, thereby degrading performance. Using the Buffer suite, a plug–in can avoid some of the memory accounting. This simplifies the prepare phase for Acquire, Filter, and Format plug–ins.

For most types of plug–ins, buffer allocations can be delayed until they are actually needed. Unfortunately, Export modules must track the buffer for the data requested from the host even though the host allocates the buffer. This means that the Buffer suite routines do not provide much help for Export modules.

BufferSpaceProc( )

```
MACPASCAL int32 (*BufferSpaceProc) (void);
```

This routine returns the amount of space available for buffers. This space may be fragmented so an attempt to allocate all of the space as a single buffer may fail.

AllocateBufferProc( )

```
MACPASCAL OSErr (*AllocateBufferProc) (int32 size, BufferID *buffer);
```

Buffers are identified by pointers to an opaque type called `BufferID`.

This routine sets `buffer` to be the ID for a buffer of the requested size. It returns `noErr` if allocation is successful, and an error code if allocation is unsuccessful. Buffer allocation is more likely to fail during phases where other blocks of memory are locked down for the plug–in's benefit, such as the `continue` calls to Filter and Export plug–ins.

FreeBufferProc( )

```
MACPASCAL void (*FreeBufferProc) (BufferID buffer);
```

This routine releases the storage associated with a buffer. Use of the buffer's ID after calling *FreeBufferProc* will probably result in glorious crashes.

LockBufferProc( )

```
MACPASCAL Ptr (*LockBufferProc) (BufferID buffer, Boolean moveHigh);
```

This locks the buffer so that it won't move in memory and returns a pointer to the beginning of the buffer. Under the Mac OS, the `moveHigh` flag indicates whether you want the memory blocked moved to the high end of memory to avoid fragmentation. The `moveHigh` flag has no effect with Windows.

UnlockBufferProc( )

```
MACPASCAL void (*UnlockBufferProc) (BufferID buffer);
```

This is the corresponding routine to unlock a buffer. Buffer locking uses a reference counting scheme; a buffer may be locked multiple times and only the final balancing unlock call will actually unlock it.

Channel Ports suite

Current version: 1; Adobe Photoshop: 5.0; Routines: 3.

Channel Ports are access points for reading and writing data from Photoshop's internal selection data structures. There are two types of ports: *read* ports and *write* ports. You can retrieve a read port corresponding to a write port, but you cannot retrieve a write port from a read port. The API does allow for write-only ports, although none exist as of this version of the suite.

These structures are used to get at merged pixel information, such as iterating through the merged data of the current layer or entire document, to be able to return a selection or use for a preview proxy.

For more information, refer to chapter *10. Selection Modules*, on page 36.

ReadPixelsProc( )

```
MACPASCAL OSErr (*ReadPixelsProc) (ChannelReadPort port, const PSScaling
*scaling, const VRect *writeRect, const PixelMemoryDesc *destination, VRect
*wroteRect);
```

This routine takes a read port, a scaling, a destination rectangle, a description of the memory to be written, and a pointer to another rectangle.

PSScaling

```
typedef struct PSScaling
    {
    VRect sourceRect;
    VRect destinationRect;
    }
PSScaling;
```

PSScaling is a rectangle in source space and a corresponding rectangle in destination space. Equal rectangles result in direct mapping. Unequal rectangles can be used to up- or down-sample. First, the destination space rectangle is projected back to the source space. Then the overlap with the given channel is copied to the specified memory.

PixelMemoryDesc

Memory is described using *PixelMemoryDesc*.

```
typedef struct PixelMemoryDesc
{
    void *              data;
    int32               rowBits;
    int32               colBits;
    int32               bitOffset;
    int32               depth;
} PixelMemoryDesc;
```

Table 3–1: PixelMemoryDesc structure

Type	Field	Description
void *	data	Coordinate of the first byte of the first pixel.
int32	rowBits	Number of bits per row. Should be multiple of `depth` (and generally should be a multiple of 8).

Table 3–1: PixelMemoryDesc structure

Type	Field	Description
int32	colBits	Number of bits per column. Should be multiple of `depth`. If `depth=1` then set `colBits=1`.
int32	bitOffset	Bit offset from the pointer value.
int32	depth	Pixel depth.

wroteRect
The last parameter to `ReadPixels` is a pointer to a rectangle that will be filled in by the host with the rectangle in the destination space that was actually written. If the plug-in reads an area that fits entirely within the channel, this will match the destination rectangle. If the plug-in reads an area that doesn't fit entirely within the channel, the destination pixels without corresponding source pixels won't be written and `wroteRect` won't include them.

WriteBasePixelsProc( )

```
MACPASCAL OSErr (*WriteBasePixelsProc) (ChannelWritePort port, const VRect
*writeRect, const PixelMemoryDesc *source);
```

This routine requires a write port, a rectangle to write, and a memory descriptor indicating the source. It does not support scaling. Any pixels in the rectangle that are beyond the bounds of the port won't be written.

ReadPortForWritePortProc( )

```
MACPASCAL OSErr (*ReadPortForWritePortProc) (ChannelReadPort *readPort,
ChannelWritePort writePort);
```

This routine returns the read port corresponding to a write port.

Descriptor suite

Current version: 0; Adobe Photoshop 5.0; Routines: 3.

The Descriptor suite provides all the callbacks related to scripting. It is divided into two sub-suites located in its parameter block.

For more information, refer to chapter *11. Scripting Plug-ins*, on page 115.

PIDescriptorParameters

```
typedef struct PIDescriptorParameters
{
    int16                       descriptorParametersVersion;
    int16                       playInfo;
    int16                       recordInfo;

    PIDescriptorHandle          descriptor;

    WriteDescriptorProcs*       writeDescriptorProcs;
    ReadDescriptorProcs*        readDescriptorProcs;
} PIDescriptorParameters;
```

Table 3–2: PIDescriptorParameters structure

Type	Field	Description
int16	descriptorParametersVersion	Minimum version required to process structure.
int16	playInfo	Flags for playback: 0=plugInDialogOptional 1=plugInDialogRequired 2=plugInDialogNone
int16	recordInfo	Flags for recording: 0=plugInDialogDontDisplay 1=plugInDialogDisplay 2=plugInDialogSilent
PIDescriptorHandle	descriptor	Handle to actual descriptor key/value pairs.
WriteDescriptorProcs*	writeDescriptorProcs	WriteDescriptorProcs sub-suite.
ReadDescriptorProcs*	readDescriptorProcs	ReadDescriptorProcs sub-suite.

ReadDescriptorProcs suite

Current version: 0; Adobe Photoshop: 5.0; Routines: 18.

The *ReadDescriptorProcs* suite is a sub-suite of the Descriptor suite that handles all the `Get` functionality for scripting. Make sure to check its version number and number of routines for compatibility before using its callbacks.

OpenReadDescriptorProc( )

```
MACPASCAL PIReadDescriptor (*OpenReadDescriptorProc) (PIDescriptorHandle,
DescriptorKeyIDArray);
```

This routine creates a `PIReadDescriptor` structure from a `PIDescriptorParameters` structure pointed to by `PIDescriptorHandle`. It returns `NULL` if unable to allocate the memory for the new handle.

`DescriptorKeyIDArray` is a `NULL`-terminated array you may provide to automatically track which keys have been returned. As each key is returned (via `GetKeyProc`) it will be changed to null-string ("`\0`") in the array. If you get to `CloseReadDescriptorProc` and your array is not empty, that indicates any keys you expected but were not given. You can subsequently coerce missing information or request it in a dialog from the user (as long as `playInfo | plugInDialogSilent`).

CloseReadDescriptorProc( )

```
MACPASCAL OSErr (*CloseReadDescriptorProc) (PIReadDescriptor);
```

This routine closes the `PIReadDescriptor` handle. It returns the most major error that occurred during reading, if any.

GetAliasProc( )

```
MACPASCAL OSErr (*GetAliasProc) (PIReadDescriptor descriptor, AliasHandle
*data);
```

This routine returns an alias from a descriptor structure.

GetBooleanProc( )

```
MACPASCAL OSErr (*GetBooleanProc) (PIReadDescriptor descriptor, Boolean
*data);
```

This routine returns a Boolean value from a descriptor structure.

GetClassProc( )

```
MACPASCAL OSErr (*GetClassProc) (PIReadDescriptor descriptor, DescType *type);
```

This routine returns a class description type from a descriptor structure.

GetCountProc( )

```
MACPASCAL OSErr (*GetCountProc) (PIReadDescriptor descriptor, uint32 *count);
```

This routine returns an unsigned long integer with the number of descriptors (the count) from a descriptor structure.

GetEnumeratedProc( )

```
MACPASCAL OSErr (*GetFloatProc) (PIReadDescriptor descriptor, DescType *type);
```

This routine returns an enumerated description type from a descriptor structure.

GetFloatProc( )

```
MACPASCAL OSErr (*GetFloatProc) (PIReadDescriptor descriptor, double *data);
```

This routine returns a floating point number from a descriptor structure.

GetIntegerProc( )

```
MACPASCAL OSErr (*GetIntegerProc) (PIReadDescriptor descriptor, int32 *data);
```

This routine returns a long integer from a descriptor structure.

GetKeyProc( )

```
MACPASCAL Boolean (*GetKeyProc) (PIReadDescriptor descriptor, DescriptorKeyID
*key, DescType *type, int16 *flags);
```

This routine returns a key ID, description type, and flags from a descriptor structure.

Table 3–3: Flags returned by GetKey

Name	Value
actionSimpleParameter	0x00000000L
actionEnumeratedParameter	0x00002000L
actionListParameter	0x00004000L
actionOptionalParameter	0x00008000L
actionObjectParameter	0x80000000L
actionScopedParameter	0x40000000L
actionStringIDParameter	0x20000000L

GetSimpleReferenceProc( )

```
MACPASCAL OSErr (*GetSimpleReferenceProc) (PIReadDescriptor descriptor,
PIDescriptorSimpleReference *ref);
```

This routine returns a basic reference from a descriptor structure:

PIDescriptorSimpleReference

```
typedef struct PIDescriptorSimpleReference
{
    DescType             desiredClass;
    DescType             keyForm;
    struct _keyData
    {
       Str255            name;
       uint32            index;
    } keyData;
} PIDescriptorSimpleReference;
```

Table 3–4: PIDescriptorSimpleReference structure

Type	Field	Description
DescType	desiredClass	Desired target class.
DescType	keyForm	Form for key ID.
_keyData	keyData	Key information. See table 3–5.

Table 3–5: _keyData structure

Type	Field	Description
Str255	name	Key name.
uint32	index	Unsigned long integer, index number.

GetObjectProc( )

```
MACPASCAL OSErr (*GetObjectProc) (PIReadDescriptor descriptor, DescType *type,
PIDescriptorHandle *data);
```

This routine returns a descriptor type and handle to corresponding object from a descriptor structure.

GetPinnedFloatProc( )

```
MACPASCAL OSErr (*GetPinnedFloatProc) (PIReadDescriptor descriptor, const
double *min, const double *max, double *floatNumber);
```

This routine returns a floating point number from a descriptor structure. If the value is out of range, it returns `coercedParam` and stores either the minimum or maximum value in `floatNumber`, whichever is closer.

GetPinnedIntegerProc( )

```
MACPASCAL OSErr (*GetPinnedIntegerProc) (PIReadDescriptor descriptor, int32
min, int32 max, int32 *intNumber);
```

This routine returns a long integer from a descriptor structure. If the value is out of range, it returns `coercedParam` and stores either the minimum or maximum value in `intNumber`, whichever is closer.

GetPinnedUnitFloatProc( )

```
MACPASCAL OSErr (*GetPinnedUnitFloatProc) (PIReadDescriptor descriptor, const
double *min, const double *max, DescriptorUnitID *units, double *floatNumber);
```

This routine returns a floating point unit-specified number from a descriptor structure. If the value is out of range, it returns `coercedParam` and stores either the minimum or maximum value in `floatNumber`, whichever is closer.

Table 3–6: Predefined units

Name	Value
unitDistance	'#Rlt'
unitAngle	'#Ang'
unitDensity	'#Rsl'
unitPixels	'#Pxl'
unitPercent	'#Prc'

GetStringProc( )

```
MACPASCAL OSErr (*GetStringProc) (PIReadDescriptor descriptor, Str255 *data);
```

This routine returns a string from a descriptor structure.

GetTextProc( )

```
MACPASCAL OSErr (*GetTextProc) (PIReadDescriptor descriptor, Handle *data);
```

This routine returns a handle to text from a descriptor structure.

GetUnitFloatProc( )

```
MACPASCAL OSErr (*GetUnitFloatProc) (PIReadDescriptor descriptor,
DescriptorUnitID *units, double *floatNumber);
```

This routine returns a unit-based floating point number from a descriptor structure.

WriteDescriptorProc suite

Current version: 0; Adobe Photoshop 5.0; Routines: 16.

The *WriteDescriptorProc* suite is a sub-suite of the Descriptor suite that handles all the `Put` functionality for scripting. Make sure to check its version number and number of routines for compatibility before using its callbacks.

OpenWriteDescriptorProc( )

```
MACPASCAL PIWriteDescriptor (*OpenWriteDescriptorProc) (void);
```

This routine opens `PIWriteDescriptor` handle for access to its descriptor array, or `NULL` if unable to allocate the memory for the handle.

CloseWriteDescriptorProc( )

```
MACPASCAL OSErr (*CloseWriteDescriptorProc) (PIWriteDescriptor descriptor,
PIDescriptorHandle *newDescriptor);
```

This routine creates a new `PIDescriptorHandle` and closes the `PIWriteDescriptor` handle. Return the `PIDescriptorHandle` to the host in `PIDescriptorParameters`. If the routine returns `NULL` then it was unable to allocate the memory for the new handle.

PutAliasProc( )

```
MACPASCAL OSErr (*PutAliasProc) (PIWriteDescriptor descriptor, DescriptorKeyID
key, AliasHandle data);
```

This routine stores an ID and corresponding alias into a descriptor structure.

PutBooleanProc( )

```
MACPASCAL OSErr (*PutBooleanProc) (PIWriteDescriptor descriptor,
DescriptorKeyID key, Boolean data);
```

This routine stores an ID and corresponding Boolean value into a descriptor structure.

PutClassProc( )

```
MACPASCAL OSErr (*PutClassProc) (PIWriteDescriptor descriptor, DescriptorKeyID
key, DescType type);
```

This routine stores an ID and corresponding class description type into a descriptor structure.

PutCountProc( )

```
MACPASCAL OSErr (*PutCountProc) (PIWriteDescriptor descriptor, DescriptorKeyID
```

```
key, uint32 count);
```

This routine stores an ID and corresponding unsigned long integer into a descriptor structure.

PutEnumeratedProc( )

```
MACPASCAL OSErr (*PutFloatProc) (PIWriteDescriptor descriptor, DescriptorKeyID
key, DescType type, DescType value);
```

This routine stores an ID and corresponding type and enumeration into a descriptor structure.

PutFloatProc( )

```
MACPASCAL OSErr (*PutFloatProc) (PIWriteDescriptor descriptor, DescriptorKeyID
key, const double *data);
```

This routine stores an ID and corresponding floating point number into a descriptor structure.

PutIntegerProc( )

```
MACPASCAL OSErr (*PutIntegerProc) (PIWriteDescriptor descriptor,
DescriptorKeyID key, int32 data);
```

This routine stores an ID and corresponding integer into a descriptor structure.

PutSimpleReferenceProc( )

```
MACPASCAL OSErr (*PutSimpleReferenceProc) (PIWriteDescriptor descriptor,
DescriptorKeyID key, const PIDescriptorSimpleReference *ref);
```

This routine stores a basic reference class, type, name, and index into a descriptor structure. See table 3–4.

PutObjectProc( )

```
MACPASCAL OSErr (*PutObjectProc) (PIWriteDescriptor descriptor,
DescriptorKeyID key, PIDescriptorHandle data);
```

This routine stores an ID and corresponding object into a descriptor structure.

PutStringProc( )

```
MACPASCAL OSErr (*PutStringProc) (PIWriteDescriptor descriptor,
DescriptorKeyID key, ConstStr255Param data);
```

This routine stores an ID and corresponding string into a descriptor structure.

PutTextProc( )

```
MACPASCAL OSErr (*PutTextProc) (PIWriteDescriptor descriptor, DescriptorKeyID
key, Handle data);
```

This routine stores an ID and corresponding text into a descriptor structure.

Handle suite

Current version: 1; Adobe Photoshop: 5.0; Routines: 7.

The use of handles in the Pseudo–Resource suite poses a problem under Windows, where a direct equivalent does not exist. In this situation, Photoshop implements a handle model which is very similar to handles under the Mac OS.

The following suite of routines is used primarily for cross–platform support. Although you can allocate handles directly using the Macintosh Toolbox, these callbacks are recommended, instead. When you use these callbacks, Photoshop will account for these handles in its virtual memory space calculations.

If your plug–in is intended to run only with Photoshop 3.0 or later, the Buffer suite routines are more effective for memory allocation than the Handle suite. The Buffer suite may have access to memory unavailable to the Handle suite. You should use the Handle suite, however, if the data you are managing is a Mac OS handle.

NewPIHandleProc( )

```
MACPASCAL Handle (*NewPIHandleProc) (int32 size);
```

This routine allocates a handle of the indicated size. It returns `NULL` if the handle could not be allocated.

DisposePIHandleProc( )

```
MACPASCAL void (*DisposePIHandleProc) (Handle h);
```

This routine disposes of the indicated handle.

GetPIHandleSizeProc( )

```
MACPASCAL int32 (*GetPIHandleSizeProc) (Handle h);
```

This routine returns the size of the indicated handle.

SetPIHandleSizeProc( )

```
MACPASCAL OSErr (*SetPIHandleSizeProc) (Handle h, int32 newSize);
```

This routine attempts to resize the indicated handle. It returns `noErr` if successful and an error code if unsuccessful.

LockPIHandleProc( )

```
MACPASCAL Ptr (*LockPIHandleProc) (Handle h, Boolean moveHigh);
```

This routine locks and dereferences the handle. Optionally, the routine will move the handle to the high end of memory before locking it.

UnlockPIHandleProc( )

```
MACPASCAL void (*UnlockPIHandleProc) (Handle h);
```

This routine unlocks the handle. Unlike the routines for buffers, the lock and unlock calls for handles do not nest. A single unlock call unlocks the handle no matter how many times it has been locked.

RecoverSpaceProc( )

```
MACPASCAL void (*RecoverSpaceProc) (int32 size);
```

All handles allocated through the Handle suite have their space accounted for in Photoshop's estimates of how much image data it can make resident at one time.

If you obtain a handle via the Handle suite or some other mechanism in Photoshop, you should dispose of it using the `DisposePIHandle` callback. If you dispose of in some other way (e.g., use the handle as the parameter to `AddResource` and then close the resource file), then you can use this call to tell Photoshop to decrease its handle memory pool estimate.

Image Services suite

Current version: 1; Adobe Photoshop: 5.0; Routines: 1.

The Image Services suite is available in Adobe Photoshop version 3.0.4 and later. It provides access to some image procession routines inside Photoshop. Currently it includes two resampling routines; future versions may provide access to other functions. Acquire, Export, and Filter plug–in modules have access to these callbacks.

These routines are used in the distortion filters that ship with Adobe Photoshop 5.0

The `PSImagePlane` structure describes the 8–bit plane of pixel data used by the image services callback functions.

```
typedef struct PSImagePlane
{
    void *              data;
    Rect                bounds;
    int32               rowBytes;
    int32               colBytes;
} PSImagePlane;
```

Table 3–7: PSImagePlane structure

Type	Field	Description
void *	data	Pointer to the byte containing the value of the top left pixel.
Rect	bounds	Coordinate systems for the pixels.
int32	rowBytes	Step values to access individual pixels.
int32	colBytes	

To calculate a point's address, use the algorithm:

```
unsigned8 * GetPixelAddress(PSImagePlane * plane, Point pt)
{
    // should do some bounds checking here!
    return (unsigned8 *) (((long) plane->data +
                          (pt.v - plane->bounds.top ) * plane->rowBytes +
                          (pt.h - plane->bounds.left) * plane->colBytes);
}
```

PIResampleProc( )

```
MACPASCAL OSErr (*PIResampleProc)  (PSImagePlane *source,
                                    PSImagePlane *destination,
                                    Rect *area,
                                    Fixed *coords,
                                    int16 method);
```

The image services suite contains two callbacks with this function type: *interpolate1D* and *interpolate2D*. These are explained in detail below.

source / destination
The *source* and *destination* parameters point to the source and destination images, respectively.

area
The *area* parameter points to an area in the destination image plane that you wish to modify. The `area` rectangle must be contained within `destination->bounds`.

coords
The *coords* parameter points to an array you create that controls the image resampling. The array will contain either one or two fixed point numbers for each pixel in the `area` rectangle (see below).

method
The *method* parameter indicates the sampling method to use. `method=0` indicates point sampling, `method=1` indicates linear interpolation.

For a source coordinate `<fv, fh>`, Photoshop will write to the destination plane if and only if:

```
source->bounds.top <= fv <= source.bounds.bottom - 1
```

and

```
source->bounds.left <= fh <= source.bounds.right - 1
```

If `fv` and/or `fh` are not integers, using point sampling, `method=0`, Photoshop rounds to the nearest integer. Interpolation, `method=1`, performs the appropriate bilinear interpolation using up to four source pixels.

The two `PIResampleProc` callback functions differ in how they generate the sample coordinates for each pixel in the target area.

interpolate1DProc( )

This routine uses a coordinate list that contains one fixed point value for each pixel in the target plane, in top to bottom, left to right order. The sample coordinate is formed by taking the vertical coordinate of the destination pixel and the horizontal coordinate from the list. Thus

```
SampleLoc1D(v, h) = <v, coords[(h - area->left) +
    (v - area->top) * (area->right - area->left)]>
```

interpolate2DProc( )

This routine uses a coordinate list that contains a pair of fixed point values for each pixel in the area containing the vertical and horizontal sample coordinate.

```
SampleLoc2D(v, h) =
    <coords[2*((h - area->left) +
        (v - area->top) * (area->right - area->left))],
     coords[2*((h - area->left) +
        (v - area->top) * (area->right - area->left)) + 1]>
```

You can build a destination using relatively small input buffers by passing in a series of input buffers, since these callbacks will leave any pixels whose sample coordinates are out of bounds untouched.

Make sure that you have appropriate overlap between the `source` buffers so that sample coordinates don't "fall through the cracks." This matters even when point sampling, since the coordinate test is applied without regard to the `method` parameter. This is done so that you get consistent results when switching between point sampling and linear interpolation. If Photoshop didn't do this, you could end up modifying pixels using point sampling that wouldn't get modified when using linear interpolation.

You also want to pin coordinates to the overall source bounds so that you will manage to write everything in the destination.

To determine whether you should use point sampling or linear interpolation, you may want to check what the user has set in their Photoshop preferences. This is set in the **General Preferences** dialog, under the **Interpolation** pop–up menu. You can retrieve this value using the `GetProperty` callback with the `propInterpolationMethod` key.

Note:
This version of the resampling callback does not support the bicubic interpolation method.

Property suite

Current version: 1; Adobe Photoshop: 5.0; Routines: 2.

The Property suite allows your plug–in module to get and set certain values in the plug–in host. The property suite is available to all plug-ins.

Note: The term *property* is used with two different meanings in this toolkit. Besides its use in the Property suite, the term is also a part of the `PiPL` data structure, documented in *Plug-in Resource Guide.pdf*. There is no connection between `PiPL` properties and the Property suite.

Properties are returned as a 32 bit integer, `simpleProperty`, or a handle, `complexProperty`. In the case of a complex, handle based property, your plug–in is responsible for disposing the handle. Use the `DisposePIHandleProc` callback defined in the Handle suite.

Properties involving strings—such as channel names and path names—are returned in a Photoshop handle. The length of the handle and size of the string is obtained with `PIGetHandleSizeProc`. There is no length byte, nor is the string zero terminated.

Properties are identified by a signature and key, which form a pair to identify the property of interest. Some properties, like channel names and path names, are also indexed; you must supply the signature, key, and index (zero–based) to access or update these properties.

Adobe Photoshop's signature is always `'8BIM'` (`0x3842494D`).

The EXIF property is controlled by The Japan Electronic Industry Development Association (JEIDA) and Electronic Industries Association of Japan (EIAJ) which merged in Novemeber of 2000. The EXIF specification can be downloaded from their web site at the following location.

http://it.jeita.or.jp/jhistory/document/standard/exif_eng/jeida49eng.htm

GetPropertyProc( )

```
MACPASCAL OSErr (*GetPropertyProc) (OSType signature, OSType key, int32 index,
int32 * simpleProperty, Handle * complexProperty);
```

This routine allows you to get information about the document currently being processed.

Note: This callback replaces the direct callback, which has been renamed "`getPropertyObsolete`". The obsolete callback pointer is still correct, and is maintained for backwards compatibility.

SetPropertyProc( )

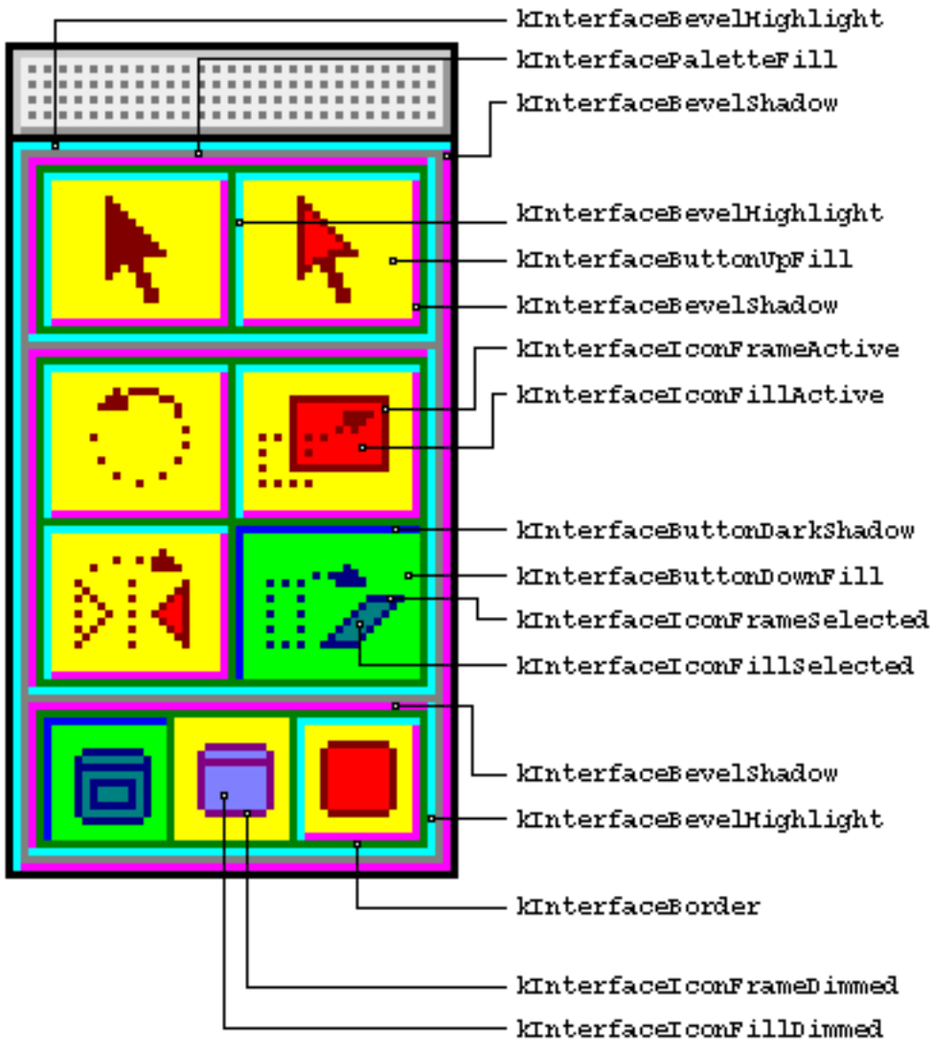
```
MACPASCAL OSErr (*SetPropertyProc) (OSType signature, OSType key, int32 index,
int32 simpleProperty, Handle complexProperty);
```

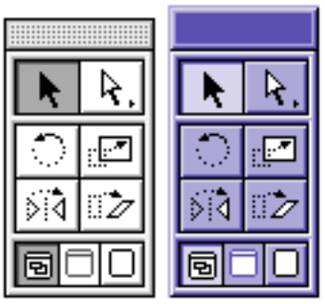
This routine allows you to update information in the plug–in host about the document currently being processed.

propInterfaceColor

Adobe Photoshop 4.0 and 5.0 include a new property, `propInterfaceColor`, which allows your interface to mimic system colors. Currently, user-selected system colors are supported on Windows; when they are available on Macintosh, they will likely be supported in future versions of Photoshop through this same mechanism.

The propInterfaceColor properties pass the user-selected interface color scheme to your plug-in according to the following diagram:





Constants get remapped to create the system look.

Use table 3–8 to draw PICTs using the index values.

Note: Until the Macintosh provides user-selected interface colors, use the file `ColorScheme-CLUT` in the `propInterfaceColor` folder in `Examples/Resources` to look-up the color values.

See the illustration above for details on each of these values.

Table 3–8: propInterfaceColor index

Name	Value
kInterfaceWhite	0
kInterfaceButtonUpFill	1
kInterfaceBevelShadow	2
kInterfaceIconFillActive	3
kInterfaceIconFillDimmed	4

Table 3–8: propInterfaceColor index (Continued)

Name	Value
kInterfacePaletteFill	5
kInterfaceIconFrameDimmed	6
kInterfaceIconFrameActive	7
kInterfaceBevelHighlight	8
kInterfaceButtonDownFill	9
kInterfaceIconFillSelected	10
kInterfaceBorder	11
kInterfaceButtonDarkShadow	12
kInterfaceIconFrameSelected	13
kInterfaceBlack	14
kInterfaceRed	15

Property Keys

Properties marked *"mod"* in table 3–9 are modifiable and can be altered with `SetProperty`.

Table 3–9: Property keys recognized by Property Suite callbacks

Property Name	ID	Type	Description
propNumberOfChannels	`nuch`	simple	Number of channels in the document. This count will include the transparency mask and the layer mask for the target layer if these are present.
propChannelName	`nmch`	complex string	Name of the channel. The channels are indexed from zero and consist of the composite channels, the transpareny mask, the layer mask, and the alpha channels.
propImageMode	`mode`	simple	Mode of the image.
propNumberOfPaths	`nupa`	simple	Number of paths in the document.
propPathName	`nmpa`	complex string	Name of the indexed path. The paths are indexed starting with zero.
propPathContents	`path`	complex data structure	Contents of the indexed path in the format documented in the path resources documentation. The data is stored in big endian form. Refer to chapter 10 for more information on path data.
propWorkPathIndex	`wkpa`	simple	Index of the work path. –1=no path.
propClippingPathIndex	`clpa`	simple	Index of the clipping path. –1=no path.
propTargetPathIndex	`tgpa`	simple	Index of the target path. –1=no path.
propCaption	`capt`	complex *mod*	File meta information in a IPTC-NAA record. For more information, see chapter *Photoshop File Formats.pdf.*
propBigNudgeH	`bndH`	simple *mod*	Horizonal component of the nudge distance, represented as a 16.16 value. This is the value used when moving around using the shift key. The default value is ten pixels.
propBigNudgeV	`bndV`	simple *mod*	Vertical component of the nudge distance, represented as a 16.16 value. This is the value used when moving around using the shift key. The default value is ten pixels.

Table 3–9: Property keys recognized by Property Suite callbacks (Continued)

Property Name	ID	Type	Description
propInterpolationMethod	intp	simple	Current interpolation method: 1=point sample, 2=bilinear, 3=bicubic.
propRulerUnits	rulr	simple	Current ruler units.
propRulerOriginH	rorH	simple *mod*	Horizontal component of the current ruler origin, represented as a 16.16 value.
propRulerOriginV	rorV	simple *mod*	Vertical component of the current ruler origin, represented as a 16.16 value.
propGridMajor	grmj	simple *mod*	The current major grid rules, in inches, unless propRulerUnits is pixels, and then pixels. Represented as a 16.16 value.
propGridMinor	grmn	simple *mod*	The current number of grid subdivisions per major rule.
propSerialString	sstr	complex string	Serial number of the plug–in host as a string. You can use this to implement copy protection for your plug–in module.
propHardwareGammaTable	hgam	complex	Hardware gamma table (Windows only).
propInterfaceColor	iclr	complex	Property interface color. See above.
propWatchSuspension	wtch	simple *mod*	The watch suspension level. When non-zero, you can make callbacks to the host without fear that the watch will start spinning. It is reset to zero at the beginning of each call from the host to the plug-in.
propCopyright	cpyr or cpyR	simple *mod*	Whether the current image is considered copyrighted.
propURL	URL	complex *mod*	The URL for the current image.
propTitle	titl	complex	The title of the current image.
propPathContentsAI	paAI	complex	Returns the contents of the path as Illustrator data. (zero-based)
propWatermark	watr	simple *mod*	Indicate whether a digital signature or watermark is present. The (c) copyright symbol will appear if this is set, or if the user has checked the copyrightproperty in the File Info dialog. Do not turn the copyright flag off, ever. Use to indicate if you've found your digital signature.
propDocumentWidth	docW	simple	The width of the current document in pixels.
propDocumentHeight	docH	simple	The height of the current document in pixels.
propToolTips	tltp	simple	0 or 1 for whether tool tips are displayed.
propPaintCursorKind	PCrK	simple	0 = standard, 1 = precise, 2 = brush size
propSlices	slcs	complex *mod*	Slices. See the Slices resource format documented in *Photoshop File Formats.pdf*
propEXIFData	EXIF	complex *mod*	Camera and device data. See above for more details.

Pseudo–Resource suite

Current version: 3; Adobe Photoshop: 5.0; Routines: 4.

This suite of callback routines provides support for storing and retrieving data from a document. These routines provide pseudo–resources which plug–in modules can attach to documents and use to communicate with each other.

Each resource is a handle of data and is identified by a 4 character code ResType and a one–based index. The maximum number of pseudo-resources in a document for Photoshop is 1000.

CountPIResourcesProc( )

```
MACPASCAL int16 (*CountPIResourcesProc) (ResType ofType);
```

This routine returns a count of the number of resources of a given type.

GetPIResourceProc( )

```
MACPASCAL Handle (*GetPIResourceProc) (ResType ofType, int16 index);
```

This routine returns the indicated resource for the current document or `NULL` if no resource exists with that type and index. The plug-in host owns the returned handle. The handle should be treated as read-only.

AddPIResourceProc( )

```
MACPASCAL OSErr (*AddPIResourceProc) (ResType ofType, Handle data);
```

This routine adds a resource of the given type at the end of the list for that type. The contents of `data` are duplicated so that the plug–in retains control over the original handle. If there is not enough memory or the document already has too many plug–in resources, this routine will return `memFullErr`.

DeletePIResourceProc( )

```
MACPASCAL void (*DeletePIResourceProc) (ResType ofType, int16 index);
```

This routine deletes the indicated resource in the current document. Note that since resources are identified by index rather than ID, this will cause subsequent resources to be renumbered.

4. Suite PEA Callbacks

Suite PEA is a plug-in architecture used by a number of Adobe Systems applications. A plug-in is a file containing a computer program and resources that extend the functionality of a host application. Suite PEA provides a common plug-in management core to the host application and a standard interface for plug-ins.

The host's application programming interface (API) is exposed to plug-ins via "suites." A suite is simply a pointer to a data structure that provides an interface to some common object, often a collection of function pointers. Plug-ins can extend the host API by providing their own function suites.

Before they can be used, suites must be "acquired"; when no longer needed, suites are "released". This guarantees that the functions are available to the plug-in. An acquired suite is actually a pointer to a structure with the suite's function pointers. To call one of the suite functions, the syntax is:

```
sSuite->function();
```

So to use a suite function, you do something like this:

```
ADMBasicSuite *sADMBasic;
filterParamBlock->sSPBasic->AcquireSuite(
                          kADMBasicSuite,
                          kADMBasicSuiteVersion,
                          &sADMBasic );
sADMBasic->Beep( );
filterParamBlock->sSPBasic->ReleaseSuite(
                          kADMBasicSuite,
                          kADMBasicSuiteVersion );
```

The convention used by this SDK is for suite variables to be global in scope and indicated by a small 's' followed by the suite name, e.g. sADMBasic as shown above.

Suite PEA plug-ins will be loaded into and unloaded from memory as needed. When a plug-in adds an ADM dialog it will remain in memory until the dialog is disposed.

Accessing Suites

Each of the suite chapters will have a section of this name with suite constants and an example of how the suites are acquired. The example will look something like this:

```
ADMDialogSuite *sADMDialog;
error = filterParamBlock->sSPBasic->AcquireSuite(
                          kADMDialogSuite,
                          kADMDialogSuiteVersion,
                          &sADMDialog );
if ( error ) goto error;
```

The suites that are currently implemented via Suite PEA by Adobe Photoshop 5.5 are:

- Buffer suite
- Channel Ports suite
- Color Space suite
- Error suite
- Handle suite
- UI Hooks suite
- Progress suite
- GetFileList suite
- GetPath suite
- ZString suite
- ADM suite(s) - referenced in ADM.pdf
- Action suite(s) - referenced in Photoshop Actions Guide.pdf
- PICA suite(s) - referenced in PICA.pdf

These are described next.

Suite PEA Buffer suite

Current version: 1; Adobe Photoshop: 5.0; Routines: 4.

The Buffer suite provides an alternative to the memory management functions available in previous versions of Photoshop's plug–in specification. It provides a set of routines to request that the host allocate and dispose of memory out of a pool which it manages.

Photoshop 2.5, for example, goes to a fair amount of trouble to balance the need for buffers of various sizes against the space needed for the tiles in its virtual memory system. Growing the space needed for buffers will result in Photoshop shrinking the number of tiles it keeps in memory.

Previous versions of the plug–in specification provide a simple mechanism for interacting with Photoshop's virtual memory system by letting a plug–in specify a certain amount of memory which the host should reserve for the plug–in.

This approach has two problems. First, the memory is reserved throughout the execution of the plug–in. Second, the plug–in may still run up against limitations imposed by the host. For example, Photoshop 2.5 will, in large memory configurations, allocate most of memory at startup via a `NewPtr` call, and this memory will never be available to the plug–in other than through the Buffer suite. Under Windows, Photoshop's memory scheme is designed so that it allocates just enough memory to prevent Windows' virtual memory manager from kicking in.

If a plug–in module allocates lots of memory using `GlobalAlloc` (Windows) or `NewPtr` (Mac OS), this scheme will be defeated and Photoshop will begin double–swapping, thereby degrading performance. Using the Buffer suite, a plug–in can avoid some of the memory accounting. This simplifies the prepare phase for Acquire, Filter, and Format plug–ins.

For most types of plug–ins, buffer allocations can be delayed until they are actually needed. Unfortunately, Export modules must track the buffer for the data requested from the host even though the host allocates the buffer. This means that the Buffer suite routines do not provide much help for Export modules.

BufferNewProc( )

```
SPAIP Ptr (*BufferNewProc) (size_t *pRequestedSize, size_t minimumSize);
```

This routine attempts to allocate the number of bytes specified by the variable pointed to by the requested size parameter and if this many bytes cannot be allocated, then the largest possible number (greater than minimumSize) will be allocated and the number of bytes actually allocated will be returned in the requestedSize variable. NULL may be passed as the pointer-to-requestedSize parameter, in which case the specified minimum number of bytes will be allocated. If this minimum number of bytes cannot be allocated, then the function will fail and return NULL.

BufferDisposeProc( )

```
SPAPI void (*BufferDisposeProc) (Ptr *ppBuffer);
```

This routine disposes of the buffer and sets the variable that refers to it to NULL. Does nothing if the buffer pointer is already NULL.

BufferGetSizeProc( )

```
SPAPI size_t (*BufferGetSizeProc) (Ptr pBuffer);
```

This routine returns the size of the buffer or zero if the buffer pointer is invalid.

BufferGetSpaceProc( )

```
SPAPI size_t (*BufferGetSpaceProc) (void);
```

This routine returns the amount of remaining space available - may not be contiguous.

Suite PEA Channel Ports suite

Current version: 3; Adobe Photoshop: 5.0; Routines: 22.

Channel Ports are access points for reading and writing data from Photoshop's internal selection data structures. There are two types of ports: *read* ports and *write* ports. You can retrieve a read port corresponding to a write port, but you cannot retrieve a write port from a read port. The API does allow for write-only ports, although none exist as of this version of the suite.

These structures are used to get at merged pixel information, such as iterating through the merged data of the current layer or entire document, to be able to return a selection or use for a preview proxy.

For more information, refer to chapter *10. Selection Modules*, on page 62.

CountLevels( )

```
SPAPI SPErr (*CountLevels) (PIChannelPort port, int32 *count);
```

This routine determine how many levels we have. Zero if the port is invalid.

GetDepth( )

```
SPAPI SPErr (*GetDepth) (PIChannelPort port, int32 level, int32 *depth);
```

This routine gets the depth at a given level. Zero if the port or level is invalid.

GetDataBounds( )

```
SPAPI SPErr (*GetDataBounds) (PIChannelPort port, int32 level, VRect
*writeBounds);
```

This routine gets the bounds for the pixel data. Returns an empty rectangle if the parameters are invalid.

GetWriteLimit( )

```
SPAPI SPErr (*GetWriteLimit) (PIChannelPort port, int32 level, VRect
*writeBounds);
```

This routine gets the bounds to which we can write at a given level.

GetTilingGrid()

```
SPAPI SPErr (*GetTilingGrid) (PIChannelPort port, int32 level, VPoint
*tileOrogin, VPoint *tileSize);
```

This routine gets the tiling information at a given level.

GetSupportRect( )

```
SPAPI SPErr (*GetSupportRect) (PIChannelPort port, int32 level const, VRect
```

```
*bounds, int32 *supportlevel, VRect *supportBounds);
```

This routine finds the rectangle used as the downsample source for a particular level in the pyramid. Level is set to -1 if no support rectangle exists.

GetDependentRect( )

```
SPAPI SPErr (*GetDependentRect) (PIChannelPort port, int32 sourceLevel, const
VRect *sourceBounds, int32 dependentLevel, VRect *dependentBounds);
```

This routine gets the dependent rectangle at a particular level.

CanRead( )

```
SPAPI SPErr (*CanRead) (PIChannelPort port, Boolean *canRead);
```

This routine tells if we can read from this port.

CanWrite( )

```
SPAPI SPErr (*CanWrite) (PIChannelPort port, Boolean *canWrite);
```

This routine tells if we can write to this port.

ReadPixelsFromLevel( )

```
SPAPI SPErr (*ReadPixelsFromLevel) (PIChannelPort port, int32 level, VRect
*bounds, const PixelMemoryDesc *destination);
```

This routine reads pixels from a given level of the port. If the result is noErr, then bounds will reflect the pixels actually read. If it reflects an error, then the value of bounds is undefined.

WritePixelsToBaseLevel( )

```
SPAPI SPErr (*WritePixelsToBaseLevel) (PIChannelPort port, VRect *bounds,
const PixelMemoryDesc *source);
```

This routine writes to a level in the pyramid.

ReadScaledPixels( )

```
SPAPI SPErr (*ReadScaledPixels) (PIChannelPort port, VRect *readRect, const
PSScaling *scaling, const PixelMemoryDesc *destination);
```

This routine reads scaled data from the pyramid. Adjusts readRect to reflect the area actually read.

FindSourceForScaledRead( )

```
SPAPI SPErr (*FindSourceForScaledRead) (PIChannelPort port, const VRect
*readRect, const PSScaling *scaling, int32 dstDepth, int32 *sourceLevel, VRect
*sourceRect, VRect *sourceScalingBounds);
```

If we just want to find out what level would be used for a given scaling, we can use the following routine. sourceLevel and sourceRect describe where in the pyramid we will be reading from. sourceScalingBounds is the bounds used for scaling from this level to the final result.

New( )

```
SPAPI SPErr (*New) (PIChannelPort *port, const VRect *rect, int32 depth,
Boolean globalScope);
```

This routine creates a pixel array and the port to go with it.

The following routines are new in version 3.

We wrap a variety of filtering operations into two callbacks. One checks to see whether an operation is supported and the other actually applies the operation from a source port to a destination port. We do this rather than adding new routines for each operation because it keeps us from having to repeatedly revise the suite number. The list of operations is provided in PIChannelPortOperations.h.

SupportsOperation( )

```
SPAPI SPErr (*SupportsOperation) (const char *operation, Boolean *supported);
```

ApplyOperation( )

```
SPAPI SPErr (*ApplyOperation) (const char *operation, PIChannelPort
sourcePort, PIChannelPort destinationPort, PIChannelPort maskPort, void
*parameters, VRect *rect);
```

The void * parameters is the exact structure of which will depend on the operation and the VRect *rect is a bounding rectangle. Could also be an output.

AddOperation( )

```
SPAPI SPErr (*AddOperation) (const char *operation, SPErr (*proc)
(PIChannelPort, PIChannelPort, PIChannelPort, void *, VRect *, void *refCon),
void *refCon);
```

This routine adds an operation.

RemoveOperation( )

```
SPAPI SPErr (*RemoveOperation) (const char *operation, void **refCon);
```

This routine removes an operation.

NewCopyOnWrite( )

```
SPAPI SPErr (*NewCopyOnWrite) (PIChannelPort *result, PIChannelPort basePort,
VRect *writeLimit, Boolean globalScope);
```

This routine provides support for using the copy-on-write mechanism. The base port must be frozen. writeLimit can be NULL in which case writing will be allowed everywhere.

Freeze( )

```
SPAPI SPErr (*Freeze) (PIChannelPort port);
```

This routine freezes the data associated with a channel port. This should generally only be used in conjunction with ports allocated via NewCopyOnWrite.

Restore( )

```
SPAPI SPErr (*Restore) (PIChannelPort port, VRect *area);
```

This routine restores an area within a copy-on-write port to its initial state. Passing null will restore everything.

Suite PEA Color Space suite

Current version: 1; Adobe Photoshop 6.0; Routines: 15.

The Color Space suite provides all the callbacks related to color space management.

ColorSpace_Make( )

```
SPAPI SPErr (*ColorSpace_Make) (ColorID *id);
```

ColorSpace_Delete( )

```
SPAPI SPErr (*ColorSpace_Delete) (ColorID *id);
```

ColorSpace_StuffComponents( )

```
SPAPI SPErr (*ColorSpace_StuffComponents) (ColorID id, short colorSpace,
unsigned char component0, unsigned char component1, unsigned char component2,
unsigned char component3);
```

ColorSpace_ExtractComponents( )

```
SPAPI SPErr (*ColorSpace_ExtractComponents) (ColorID id, short colorSpace,
unsigned char *component0, unsigned char *component1,unsigned char
*component2, unsigned char *component3, Boolean *gamutFlag);
```

ColorSpace_StuffXYZ( )

```
SPAPI SPErr (*ColorSpace_StuffXYZ) (ColorID id, CS_XYZColor xyz);
```

ColorSpace_ExtractXYZ( )

```
SPAPI SPErr (*ColorSpace_ExtractXYZ) (ColorID id, CS_XYZColor *xyz);
```

ColorSpace_GetNativeSpace( )

```
SPAPI SPErr (*ColorSpace_GetNativeSpace) (ColorID id, short *colorSpace);
```

ColorSpace_Convert8( )

```
SPAPI SPErr (*ColorSpace_Convert8) (short inputCSpace, short outputCSpace,
Color8 *colorArray, short count);
```

ColorSpace_Convert16( )

```
SPAPI SPErr (*ColorSpace_Convert16) (short inputCSpace, short outputCSpace,
```

```
Color16 *colorArray, short count);
```

ColorSpace_IsBookColor( )

```
SPAPI SPErr (*ColorSpace_IsBookColor) (ColorID id, Boolean *isBookColor);
```

ColorSpace_ExtractColorName( )

```
SPAPI SPErr (*ColorSpace_ExtractColorName) (ColorID id, ASZString *colorName);
```

ColorSpace_PickColor( )

```
SPAPI SPErr (*ColorSpace_PickColor) (ColorID *id, ASZString promptString);
```

The following routines use the Photoshop internal 16 bit range of [0..32768] data can be single or multiple channels, as long as the count includes all of it.

ColorSpace_Convert8to16( )

```
SPAPI SPErr (*ColorSpace_Convert8to16)( unsigned char *input_data, unsigned
short *output_data, short count );
```

ColorSpace_Convert16to8( )

```
SPAPI SPErr (*ColorSpace_Convert16to8)( unsigned short *input_data, unsigned
char *output_data, short count );
```

ColorSpace_ConvertToMonitorRGB( )

```
SPAPI SPErr (*ColorSpace_ConvertMonitorRGB)( short inputCSpace, Color8
*input_data, Color8 *output_data, short count );
```

Suite PEA Handle suite

Current version: 1; Adobe Photoshop: 5.0; Routines: 6.

The use of handles in the Pseudo–Resource suite poses a problem under Windows, where a direct equivalent does not exist. In this situation, Photoshop implements a handle model which is very similar to handles under the Mac OS.

The following suite of routines is used primarily for cross–platform support. Although you can allocate handles directly using the Macintosh Toolbox, these callbacks are recommended, instead. When you use these callbacks, Photoshop will account for these handles in its virtual memory space calculations.

If your plug–in is intended to run only with Photoshop 3.0 or later, the Buffer suite routines are more effective for memory allocation than the Handle suite. The Buffer suite may have access to memory unavailable to the Handle suite. You should use the Handle suite, however, if the data you are managing is a Mac OS handle.

NewPIHandleProc( )

```
MACPASCAL Handle (*NewPIHandleProc) (int32 size);
```

This routine allocates a handle of the indicated size. It returns `NULL` if the handle could not be allocated.

DisposePIHandleProc( )

```
MACPASCAL void (*DisposePIHandleProc) (Handle h);
```

This routine disposes of the indicated handle.

SetPIHandleLockProc( )

```
MACPASCAL Ptr (*SetPIHandleLockProc) (Handle h, Boolean lock, Ptr *address,
Boolean *oldLock);
```

This routine locks and dereferences the handle. Optionally, the routine will move the handle to the high end of memory before locking it.

GetPIHandleSizeProc( )

```
MACPASCAL int32 (*GetPIHandleSizeProc) (Handle h);
```

This routine returns the size of the indicated handle.

SetPIHandleSizeProc( )

```
MACPASCAL OSErr (*SetPIHandleSizeProc) (Handle h, int32 newSize);
```

This routine attempts to resize the indicated handle. It returns `noErr` if successful and an error code if unsuccessful.

RecoverSpaceProc( )

```
MACPASCAL void (*RecoverSpaceProc) (int32 size);
```

All handles allocated through the Handle suite have their space accounted for in Photoshop's estimates of how much image data it can make resident at one time.

If you obtain a handle via the Handle suite or some other mechanism in Photoshop, you should dispose of it using the `DisposePIHandle` callback. If you dispose of in some other way (e.g., use the handle as the parameter to `AddResource` and then close the resource file), then you can use this call to tell Photoshop to decrease its handle memory pool estimate.

Suite PEA Error suite

Current version: 1; Adobe Photoshop: 5.0; Routines: 3.

The Error suite is available in Adobe Photoshop version 5.0 and later. It provides a way to display error strings to the user in any format. Photoshop will always make a copy of the string passed in.

SetErrorFromPStringProc( )

```
SPAPI SPErr (*SetErrorFromPStringProc) (const Str255 errorString);
```

This routine hands the host a Pascal string containing the error string to display to the user. The host will make a copy:

SetErrorFromCStringProc( )

```
SPAPI SPErr (*SetErrorFromCStringProc) (const char* errorString);
```

This routine hands the host a C string containing the error string to display to the user.The host will make a copy:

SetErrorFromZStringProc( )

```
SPAPI SPErr (*SetErrorFromZStringProc) (const ASZString zString);
```

This routine hands the host a ZString id containing the error string to display to the user. The host will make a copy:

Suite PEA UI Hooks suite

Current version: 1; Adobe Photoshop: 5.0; Routines: 8.

The UI Hooks suite is a set of utilities that allows your plug–in module to get and set certain values in the Photoshop.

ProcessEventProc( )

```
MACPASCAL void (*ProcessEventProc) (EventRecord *event);
```

This callback is only useful under the Mac OS; *ProcessEvent* in the Windows version of Adobe Photoshop does nothing.

Adobe Photoshop provides this callback function to allow Macintosh plug–in modules to pass standard `EventRecord` pointers to Photoshop. For example, when a plug–in receives a deactivate event for one of Photoshop's windows, it should pass this event on to Photoshop.

This routine can also be used to force Photoshop to update its own windows by passing relevant update and `NULL` events.

DisplayPixelsProc( )

```
MACPASCAL OSErr (*DisplayPixelsProc) (const PSPixelMap *source,
    const VRect *srcRect, int32 dstRow, int32 dstCol,
    unsigned32 platformContext);
```

This callback routine is used to display pixels in various image modes. It takes a structure describing a block of pixels to display.

The routine will do the appropriate color space conversion and copy the results to the screen with dithering. It will leave the original data intact. If it is successful, it will return `noErr`. Non–success is generally due to unsupported color modes.

To suppress the watch cursor during updates, see `propWatchSuspension` in the Properties suite.

source
The *source* parameter points to a `PSPixelMap` structure containing the pixels to be displayed. This structure is documented in Appendix A.

srcRect
The *srcRect* parameter points to a `VRect` that indicates the rectangle of the source pixel map to display.

dstRow / dstCol
The *dstRow* and *dstCol* parameters provide the coordinates of the top-left destination pixel in the current port (i.e., the destination pixel which will correspond to the top-left pixel in `srcRect`). The display routines do not scale the pixels, so specifying the top left corner is sufficient to specify the destination.

platformContext
The *platformContext* parameter is not used under the Mac OS since the display routines simply assume that the target is the current port. On Windows, `platformContext` should be the target `hDC`, cast to an `unsigned32`.

ProgressProc( )

```
MACPASCAL void (*ProgressProc) (long done, long total);
```

Your plug–in may call this two–argument procedure periodically to update a progress indicator. The first parameter is the number of operations completed; the second is the total number of operations.

This procedure should only be called in the actual main operation of the plug–in, not while long operations are executing during the preliminary user interface.

Photoshop automatically suppresses display of the progress bar during short operations.

TestAbortProc( )

```
MACPASCAL Boolean (*TestAbortProc) ( );
```

Your plug–in should call this function several times a second during long operations to allow the user to abort the operation. If the function returns `TRUE`, the operations should be aborted. As a side effect, this changes the cursor to a watch and moves the watch hands periodically.

MainAppWindowProc( )

```
SPAPI long (*MainAppWindowProc) (void);
```

Returns Windows parent window HWND, NULL on Mac.

HostSetCursorProc( )

```
SPAPI SPErr (*HostSetCursorProc) (const PICursor_t cursorID);
```

Sets cursor to some popular Photoshop cursors. Refer to the header file for the valid cursor IDs.

HostTickCountProc( )

```
SPAPI unsigned long (*HostTickCountProc) (void);
```

Gets the tick count as 60 ticks per second.

PluginNameProc( )

```
SPAPI SPErr (*PluginNameProc) (SPPluginRef pluginRef, ASZString *pluginName);
```

Returns the name of the plugins specified by pluginRef.

Suite PEA Progress suite

Current version: 1; Adobe Photoshop: 5.0; Routines: 10.

Progress_DoProgress( )

```
SPErr (*Progress_DoProgress) (const ASZString zs, SPErr (*proc) (void *), void
*refCon);
```

Progress_DoTask( )

```
SPErr (*Progress_DoTask) (double taskLength, SPErr (*proc) (void *), void
*refCon);
```

This callback sections off a portion of the unused progress bar for execution of a subtask. The taskLength should be between 0.0 and 1.0. This routine returns theerrorcode from proc.

Progress_DoSegmentTask( )

```
SPErr (*Progress_DoSegmentTask) (int32 segmentLength, int32 *done,int32
total,SPErr (*proc) (void *),void *refCon);
```

This callback limits progress to a section of the progress bar based on executing segmentLength out of total steps. done is a counter of how much we've accomplished so far and will be incremented by segmentLength if the task succeeds. (We need done because as with DoTask, what we care about is the percentage of the remaining progress bar.)

Progress_ChangeProgressText( )

```
void (*Progress_ChangeProgressText) (const ASZString zs);
```

This callback changes the current text in the progress bar.

Progress_DoPreviewTask( )

```
SPErr (*Progress_DoPreviewTask) (const char *selector, SPErr (*proc) (void *),
void *refCon);
```

This callback performs a task using the preview sniffing logic. This aborts if we encounter an event meeting the indicated conditions. selector should be one of the following:

“up"-- Process until mouse down or key stroke

"down"-- Process until mouse is released

"paused"-- Process until mouse is moved or released

Passing null will cause the code to choose between "up" and "paused" dependent on the current mouse state. All other strings result in an immediate error.

Progress_DoWatchTask( )

```
SPErr (*Progress_DoWatchTask) (SPErr (*proc) (void *), void *refCon);
```

This callback runs a task with the watch cursor up.

Progress_DoSuspendedWatchTask( )

```
SPErr (*Progress_DoSuspendedWatchTask) (SPErr (*proc) (void *), void *refCon);
```

This callback runs a task without the watch cursor. This doesn't actually take the watch cursor down, but it will keep the watch from spinning.

Progress_ContinueWatchCursor( )

```
void (*Progress_ContinueWatchCursor) (void);
```

The following are taken from from the parameter block and are defined in PIGeneral.h

TestAbortProc( )

ProgressProc( )

Suite PEA GetFileList suite

Current version: 3; Adobe Photoshop: 5.5; Routines: 7.

GetFileHandleListProc( )

```
SPAPI SPErr (*GetFileHandleListProc) (PIActionDescriptor& des,FSSpec*
fileSpec, bool recurse);
```

This callback gets a list of file handles from FSSpec. If the second parameter is true (recursive), it will go through all the folders and subfolders and put all the files in a list and sort them and remove the duplicate entries. The return value PIActionDescriptor will have PIActionList which contains all the file handles in the right order.

GetBrowserNameListProc( )

```
SPAPI SPErr (*GetBrowserNameListProc) (PIActionDescriptor& des);
```

This callback gets a list of browser names in the Helper\Preview In folder. All the browser names will be ZStrings in PIActionList

BrowseUrlWithIndexBrowserProc( )

```
SPAPI SPErr (*BrowseUrlWithIndexBrowserProc) (uint16 index, const char* url);
```

This callback browses a given url with a given index for the browsers in the "Preview In" folder. The index parameter is zero based

BrowseUrlProc( )

```
SPAPI SPErr (*BrowseUrlProc) (const char* url);
```

This callback browses a url with the default browser

GetBrowserFileSpecProc( )

```
SPAPI SPErr (*GetBrowserFileSpecProc)(uint16 index, SPPlatformFileSpecifica-
tion* fileSpec);
```

This callback gets the browser's SPPlatformFileSpecification using GetBrowserFileSpec method. The method takes the zero_based index for the browser.

The following routine is new in version 2.

GetDefaultSystemScriptProc( )

```
SPAPI SPErr (*GetDefaultSystemScriptProc)(int16& script);
```

The following routine is new in version 3.

HasDoubleByteInStringProc( )

```
SPAPI SPErr (*HasDoubleByteInStringProc)(const char* charString, bool& hasDou-
bleByte);
```

Suite PEA GetPath suite

Current version: 1; Adobe Photoshop: 5.5 ;Routines: 1.

The GetPath suite gets the path from FSSpec.

GetPathNameProc( )

```
SPAPI void (*GetPathNameProc) (SPPlatformFileSpecification* fileSpec, char*
path, int16 maxLength );
```

Suite ZString suite

Current version: 1; Adobe Photoshop: 5.5 ;Routines: 23.

MakeFromUnicode( )

```
ASErr ASAPI(*MakeFromUnicode)( ASUnicode *src, size_t byteCount, ASZString
*newZString );
```

MakeFromCString( )

```
ASErr ASAPI(*MakeFromCString)( const char *src, size_t byteCount, ASZString
*newZString );
```

MakeFromPascalString( )

```
ASErr ASAPI(*MakeFromPascalString)( const unsigned char *src, size_t
byteCount, ASZString *newZString );
```

MakeRomanizationOfInteger( )

```
ASErr ASAPI (*MakeRomanizationOfInteger)( ASInt32 value, ASZString *newZString
);
```

MakeRomanizationOfFixed( )

```
ASErr ASAPI (*MakeRomanizationOfFixed)( ASInt32 value, ASInt16 places,
ASBoolean trim,ASBoolean isSigned, ASZString *newZString);
```

MakeRomanizationOfDouble( )

```
ASErr ASAPI (*MakeRomanizationOfDouble)( double value, ASZString *newZString
);
```

GetEmpty( )

```
ASZString ASAPI (*GetEmpty)();
```

Copy( )

```
ASErr ASAPI (*Copy)( ASZString source, ASZString *copy);
```

Replace( )

```
ASErr ASAPI (*Replace)( ASZString zstr, ASUInt32 index, ASZString
replacement);
```

TrimEllipsis( )

```
ASErr ASAPI (*TrimEllipsis)( ASZString zstr );
```

TrimSpaces( )

```
ASErr ASAPI (*TrimSpaces)( ASZString zstr );
```

RemoveAccelerators( )

```
ASErr ASAPI (*RemoveAccelerators)( ASZString zstr );
```

AddRef( )

```
ASErr ASAPI (*AddRef)( ASZString zstr);
```

Release( )

```
ASErr ASAPI (*Release)( ASZString zstr );
```

IsAllWhiteSpace( )

```
ASBoolean ASAPI (*IsAllWhiteSpace)( ASZString zstr );
```

IsEmpty( )

```
ASBoolean ASAPI (*IsEmpty)( ASZString zstr );
```

WillReplace( )

```
ASBoolean ASAPI (*WillReplace)( ASZString zstr, ASUInt32 index );
```

LengthAsUnicodeCString( )

```
ASUInt32 ASAPI (*LengthAsUnicodeCString)( ASZString zstr );
```

AsUnicodeCString( )

```
ASErr ASAPI (*AsUnicodeCString)( ASZString zstr, ASUnicode *str, ASUInt32
strSize, ASBoolean checkStrSize );
```

LengthAsCString( )

```
ASUInt32 ASAPI (*LengthAsCString)( ASZString zstr );
```

AsCString( )

```
ASErr ASAPI (*AsCString)( ASZString zstr, char *str, ASUInt32 strSize,
ASBoolean checkStrSize );
```

LengthAsPascalString( )

```
ASUInt32 ASAPI (*LengthAsPascalString)( ASZString zstr );
```

AsPascalString( )

ASErr ASAPI (*AsPascalString)(ASZString zstr, char *str, ASUInt32 strBufferSize, ASBoolean checkBufferSize);

5. Color Picker Modules

Color Picker plug-in modules return a selected color, allowing a plug-in to be used as a method for the user to pick colors. They are accessed under the **File** menu, **Preferences...**, General dialog.

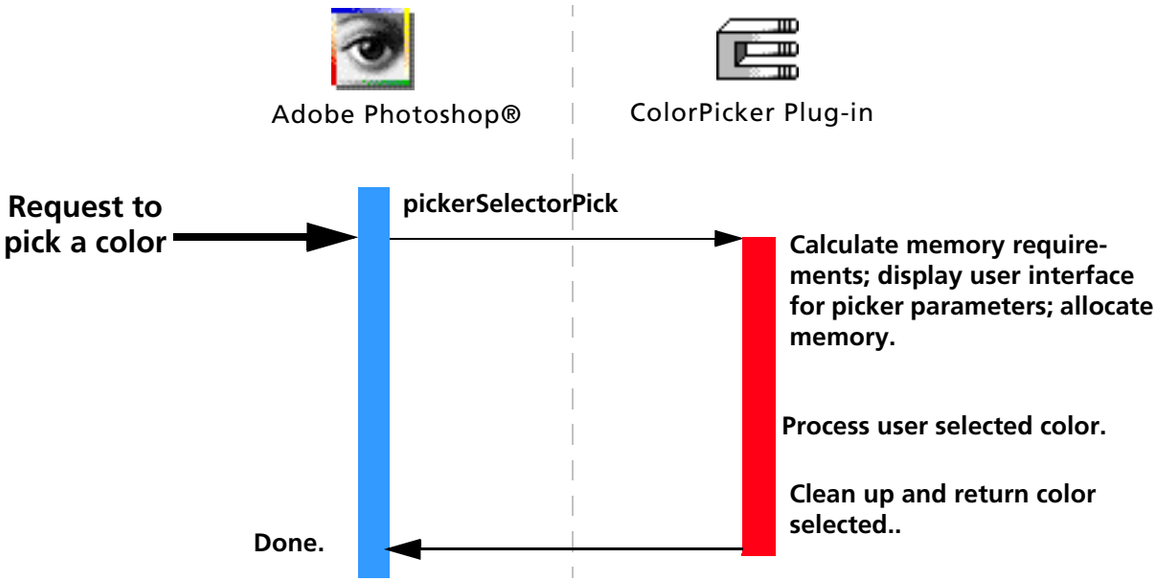
Table 4–1: Color Picker file types

OS	Filetype/extension
Mac OS	`8BCM`
Windows	`.8BC`

SampleCode/ColorPicker/NearestBase

NearestBase is a sample color picker plug–in which demonstrates simple returning of colors with no user interface.

Calling sequence



When the user invokes a Color Picker plug–in by selecting its name in the **Preferences...** General dialog and then trying to pick a custom color (such as clicking on the foreground or background colors in the tools palette), Adobe Photoshop calls your plug-in once with `pickerSelectorExecute`. The recommended sequence of actions for your plug-in to take is described next.

pickerSelectorPick

Unlike other plug-ins, a Color Picker Module only gets one execute call, and is expected to do all the work during that call. However, it's recommended you follow this order:

1. Prompt for parameters

If the plug–in has any parameters that the user can set, it should prompt the user and save the values through the recording parameters for the scriptable handle accessed through the parameters structure. Photoshop initializes the parameters field to `NULL` when starting up.

Adobe Photoshop's scripting routines save the information pointed to by the recording parameters field, so that it can operate the selection without user input during play back.

Your plug–in should validate the contents of its playback parameters when it starts processing if there is a danger of it crashing from bad parameters.

You may wish to design your plug-in so that you store default values or the last used set of values in the plug-in's Mac OS resource fork or a Windows external file. This way, you can save preference settings for your plug–in that will be consistent every time the host application runs.

2. Allocate memory

Use the buffer and handle suites to allocate any extra memory needed for your computations. See chapter 2 and 3 for a discussion on `maxData` and `bufferSpace`.

3. Compute your color space based on the user input

Compute whatever color conversions are required to return the user input to the host in the proper form.

4. Finish, clean up, and hand back your results
Clean up after your operation. Dispose any handles you created, etc., then hand back your color to the host for use.

Behavior and caveats

Color Pickers and Macintosh resource forks

Color Pickers are very special plug-ins because they can be called by other plug-ins. This means that you must be extra careful to make sure you're reading the correct resources, when you ask for them, since multiple resource forks may be available.

For instance, a Filter module uses the Color Services callback suite and requests it pop the "Choose a color" interface. The user has selected your Color Picker module as the chosen Color Picker. Now, the host's resource fork is open, the Filter module's resource fork is open, and then, once your Color Picker module is loaded, its resource fork is open.

If you need a resource in your Color Picker's resource fork, make sure to use the Macintosh toolbox call `Get1Resource`, which will look only at the most recent open resource fork, as opposed to `GetResource`, which will walk all the resource forks.

PickParms structure

This structure is used by the Color Picker plug-in to return the color selected by the user.

```
typedef struct PickParms
{
    int16 sourceSpace;
    int16 resultSpace;
    unsigned16 colorComponents[4];
    Str255 *pickerPrompt;
} PickParms;
```

Table 4–2: PickParms structure

Type	Field	Description
int16	sourceSpace	The colorspace the original color is in. See `ColorServicesInfo` in Appendix A.
int16	resultSpace	The colorspace of the returned result. See ColorServicesInfo in Appendix A. (Can be `plugInColorServicesChosenSpace`.)
unsigned16	colorComponents[4]	On selectorPick, the initial color. When exiting, set this to the color you wish to return.
Str255 *	pickerPrompt	Prompt string, supplied by ColorServices suite. See Chapter 2.

Error return values

The plug–in module may return standard operating system error codes, or report its own errors, in which case it can return any positive integer. These errors and more detail is available in `PISelection.h`.

```
#define pickerBadParameters      -30800  // a problem with the interface
```

The Color Picker parameter block

The `pluginParamBlock` parameter passed to your plug–in module's entry point contains a pointer to a `PIPickerParams` structure with the following fields. This structure is declared in `PIPicker.h`.

Table 4–3: PIPickerParams structure

Type	Field	Description
int32	serialNumber	This field contains Adobe Photoshop's serial number. Your plug–in module can use this value for copy protection, if desired.
TestAbortProc	abortProc	`TestAbort` callback. See chapter 3.
ProgressProc	progressProc	This field contains a pointer to the `UpdateProgress` callback documented in chapter 3. This procedure should only be called during the actual main operation of the plug–in, not during long operations during the preliminary user interface such as building a preview.
OSType	hostSig	The plug–in host provides its signature to your plug–in module in this field. Photoshop's signature is `8BIM`.
HostProc	hostProc	If not `NULL`, this field contains a pointer to a host–defined callback procedure that can do anything the host wishes. Plug–ins should verify `hostSig` before calling this procedure. This provides a mechanism for hosts to extend the plug–in interface to support application specific features.
BufferProcs	*bufferProcs	Buffer callback suite. See chapter 3.
ResourceProcs	*resourceProcs	Pseudo–Resource callback suite. See chapter 3.
ProcessEventProc	processEvent	`ProcessEvent` callback. See chapter 3.
DisplayPixelsProc	displayPixels	`DisplayPixels` callback. See chapter 3.
HandleProcs	*handleProcs	Handle callback suite. See chapter 3.
ColorServicesProc	colorServices	Color Services callback suite. See chapter 3.
ImageServicesProcs	*imageServicesProcs	Image Services callback suite. See chapter 3.
PropertyProcs	*propertyProcs	Property callback suite. See chapter 3.
ChannelPortProcs	*channelPortProcs	Channel Ports callback suite. See chapter 3.
PIDescriptorParameters	*descriptorParameters	Descriptor callback suite. See chapter 3.
Str255	errorString	Error string.
PlugInMonitor	monitor	Monitor setup info. See appendix A.
void *	platformData	Pointer to platform specific data. Not used in Mac OS.
char[4]	reserved	Reserved for future use. Set to zero.
These fields are new since version 5.0 of Adobe Photoshop.		
SPBasicSuite	*sSPBasic	PICA basic suite.
void	*plugInRef	Plugin reference used by PICA.
char[252]	reservedBlock	Reserved for future use. Set to zero.

7. Export Modules

Export plug–in modules are used to output an image from an open Photoshop document. They can be used to print to printers that do not have Mac OS Chooser–level driver support.

Export modules can also be used to save images in unsupported or compressed file formats, although File Format modules (see chapter 6) often are better suited for this purpose. File Format modules are accessed directly from the **Save** and **Save As...** commands, while Export modules use the **Export** sub–menu.

Table 5–1: Export file types

OS	Filetype/extension
Mac OS	`8BEM`
Windows	`.8BE`

SampleCode/Export/History

History is a sample Export module primarily concerned with demonstrating the Pseudo–Resource callbacks. It works in conjunction with the *Propetizer* plug–in to maintain a series of history strings for a file.

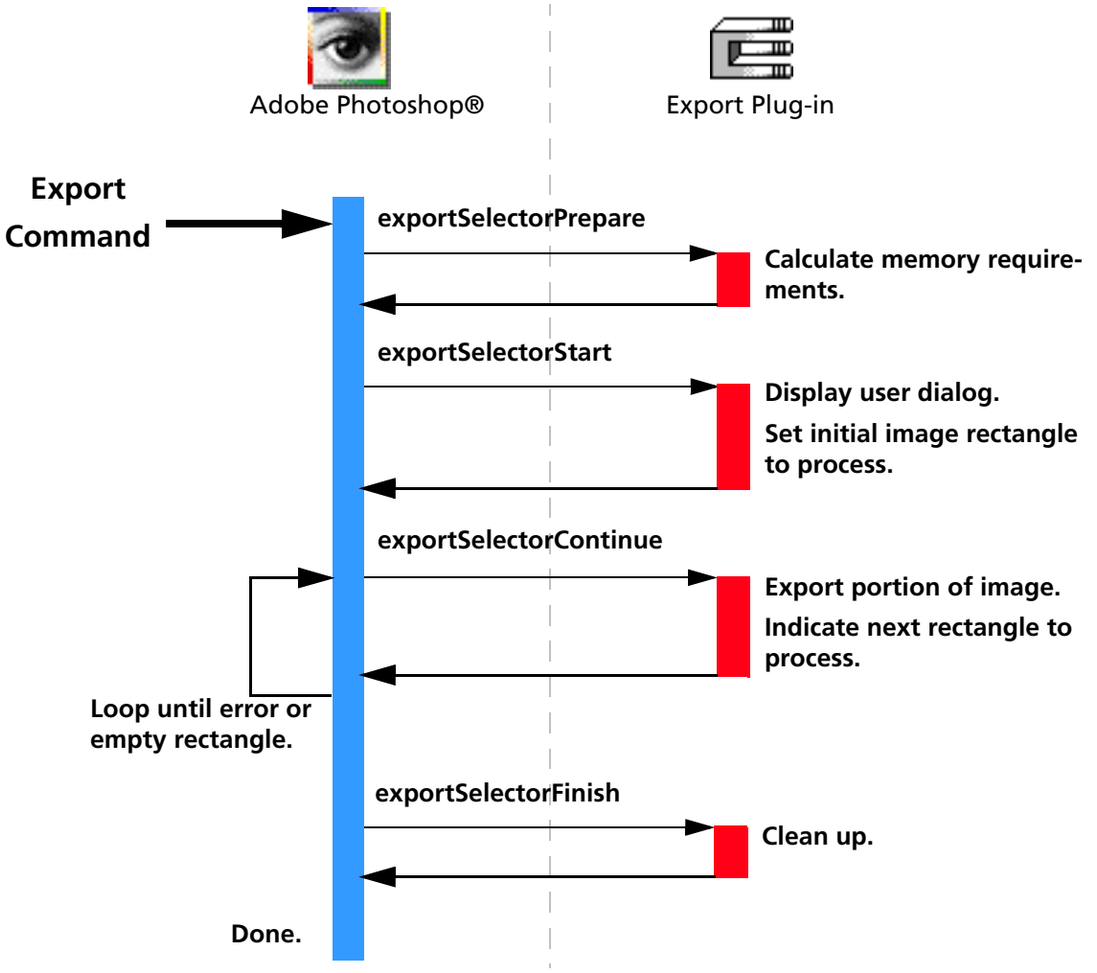
SampleCode/Export/PathsToPostScript

PathsToPostScript demonstrates using the `getProperties` callback and exporting pen path information. The sample only works on Macintosh platforms. Borrowing the porting concepts from the other examples, it is fairly straightforward port `IllustratorExport` to Windows. Please read the comments inside the sample source for important information regarding pen paths and byte ordering.

SampleCode/Export/Outbound

Outbound is a sample export module that writes a very basic image file from the data passed to it by the host.

Calling sequence



When the user invokes an Export plug–in by selecting its name from the **Export** submenu, Photoshop calls it with the sequence of selector values shown in the figure above. The actions for these selectors are discussed next.

exportSelectorPrepare

The *exportSelectorPrepare* selector calls allow your plug–in module to adjust Photoshop's memory allocation algorithm. Photoshop sets `maxData` to the maximum number of bytes it can allocate to your plug–in. You may want to reduce `maxData` for increased efficiency. Refer to chapter 3 for details on memory management strategies.

Note: Your plug–in should validate the contents of its globals and parameters whenever it starts processing if there is a danger of it crashing from bad parameters.

Globals and scripting

The scripting system passes its parameters at every selector call. While it is possible to use the scripting system to store all your parameters, for backwards compatibility, it is recommended you track your parameters with your own globals. Once your globals are initialized, you should read your scripting-passed parameters and override your globals with them.

The most effective way to do this is:

1. First call a *ValidateMyParameters* routine to validate (and initialize if necessary) your global parameters.

2. Then call a *ReadScriptingParameters* routine to read the scripting parameters and then write them to your global structure.

This way, the scripting system overrides your parameters, but you can use the initial values if the scripting system is unavailable or has parameter errors, and you can use your globals to pass between your functions.

exportSelectorStart

Most plug–ins will display their dialog box, if any, during this call.

theRect or theRect32, loPlane & hiPlane

During this call, your plug–in module should set *theRect or theRect32*, *loPlane* and *hiPlane* to let Photoshop know what area of the image it wishes to process first.

The total number of bytes requested should be less than `maxData`. If the image is larger than `maxData`, the plug–in must process the image in pieces. There are no restrictions on how the pieces tile the image: horizontal strips, vertical strips, or a grid of tiles.

exportSelectorContinue

During this routine, your plug–in module should process the image data pointed to by `data`. You should then adjust `theRect or theRect32`, `loPlane` and `hiPlane` to let Photoshop know what area of the image you wish to process next. If the entire image has been processed, set `theRect or theRect32` to an empty rectangle.

The requested image data is pointed to by `data`. If more than one plane has been requested (`loPlane`¦`hiPlane`), the data is interleaved. The offset from one row to the next is indicated by `rowBytes`. This is not necessarily equal to the width of `theRect or theRect32`; there may be additional pad bytes at the end of each row.

exportSelectorFinish

This call allows your plug–in module to clean up after an image export. This call is made if and only if the `exportSelectorStart` routine returns without error, even if the `exportSelectorContinue` routine returns an error.

If Photoshop detects *Command–period* in the Mac OS or *Escape* in Windows between calls to the `exportSelectorContinue` routine, it will call the `exportSelectorFinish` routine.

Note: Be careful processing user-cancel events during `exportSelectorContinue`. Normally your plug-in would be expecting another `exportSelectorContinue` call. If the user cancels, the next call will be `exportSelectorFinish`, *not* `exportSelectorContinue`!

Scripting at exportSelectorFinish

If your plug-in is scripting-aware and you've changed any initial parameters, you should pass a complete descriptor back to the scripting system in the `PIDescriptorParameters` structure.

Behavior and caveats

If `exportSelectorStart` succeeds then Photoshop guarantees that `exportSelectorFinish` will be called.

Photoshop may call `exportSelectorFinish` instead of `exportSelectorContinue` if it detects a need to terminate while building the requested buffer.

`advanceState` can be called from either `exportSelectorStart` or `exportSelectorContinue` and will drive Photoshop through the process of allocating and loading the requested buffer. Termination is reported as `userCanceledErr` in the result from the `advanceState` call. Calling `advanceState` when `theRect or theRect32` is empty will result in nothing.

Error return values

The plug–in module may return standard operating system error codes, or report its own errors, in which case it can return any positive integer. These errors and more detail is available in `Examples/CIncludes/PIExport.h`.

```
#define exportBadParameters -30200    // an error with the parameters
#define exportBadMode       -30201    // module does not support <mode> images
```

The Export parameter block

The `pluginParamBlock` parameter passed to your plug–in module's entry point contains a pointer to an `ExportRecord` structure with the following fields. This structure is declared in `PIExport.h`.

Table 5–2: ExportRecord structure

Type	Field	Description
int32	serialNumber	This field contains Adobe Photoshop's serial number. Plug–in modules can use this value for copy protection, if desired.
TestAbortProc	abortProc	This field contains a pointer to the `TestAbort` callback.
ProgressProc	progressProc	This field contains a pointer to the `UpdateProgress` callback. This procedure should only be called during the actual main operation of the plug–in, not during long operations during the preliminary user interface.
int32	maxData	Photoshop initializes this field to the maximum of number of bytes it can free up. You may reduce this value during in your `exportSelectorPrepare` handler. The `exportSelectorContinue` handler should process the image in pieces no larger than `maxData`, less the size of any large tables or scratch areas it has allocated.
int16	imageMode	The mode of the image being exported (grayscale, RGB Color, etc.). See `PIExport.h` for values. Your `exportSelectorStart` handler should return an `exportBadMode` error if it is unable to process this mode of image.
Point	imageSize	The image's width, `imageSize.h`, and height, `imageSize.v`, in pixels.See imageSize32 below for large document support.
int16	depth	The image's resolution in bits per pixel per plane. The only possible settings are 1 for bitmap mode images, and 8 for all other modes.
int16	planes	The number of channels in the image. For example, if an RGB image without alpha channels is being processed, this field will be set to 3.
Fixed	imageHRes	The image's horizontal and vertical resolution in terms of pixels per inch. These are fixed point 16-binary digit numbers.
Fixed	imageVRes	
LookUpTable	redLUT	If an indexed color or duotone mode image is being processed, these fields will contain its color table.
LookUpTable	greenLUT	
LookUpTable	blueLUT	
Rect	theRect	Your `exportSelectorStart` and `exportSelectorContinue` handlers should set this field to request a piece of the image for processing. It should be set to an empty rectangle when complete.See theRect32 below for large document support.
int16	loPlane	Your `exportSelectorStart` and `exportSelectorContinue` handlers should set these fields to the first and last planes to process next.
int16	hiPlane	

Table 5–2: ExportRecord structure (Continued)

Type	Field	Description
void *	data	This field contains a pointer to the requested image data. If more than one plane has been requested (`loPlane`|`hiPlane`), the data is interleaved.
int32	rowBytes	The offset between rows for the requested image data.
Str255	fileName	The name of the file the image was read from. File-exporting modules should use this field as the default name for saving.
int16	vRefNum	The volume reference number of the file the image was read from.
Boolean	dirty	If your plug–in is used to save an image into a file, you should set this field to `TRUE` to prompt the user to save any unsaved changes when the image is eventually closed. If your module outputs to a printer or other hardware device, you should set this to `FALSE`. This is initialized as `TRUE`. It does *not* reflect whether other unsaved changes have been made.
Rect	selectBBox	The bounding box of the current selection. If there is no current selection, this is an empty rectangle.See selectBB32 below for large document support.
OSType	hostSig	The plug–in host provides its signature to your plug–in module in this field. Photoshop's signature is `8BIM`.
HostProc	hostProc	If not `NULL`, this field contains a pointer to a host–defined callback procedure that can do anything the host wishes. Plug–ins should verify `hostSig` before calling this procedure. This provides a mechanism for hosts to extend the plug–in interface to support application specific features.
Handle	duotoneInfo	When exporting a duotone mode image, the host allocates a handle and fills it with the duotone information. The format of the information is the same as that required by Import modules, and should be treated as a black box by plug–ins.
int16	thePlane	Either: Currently selected channel; –1 if a composite color channel; –2 if some other combination of channels.
PlugInMonitor	monitor	This field contains the monitor setup information for the host. See Appendix A.
void *	platformData	This field contains a pointer to platform specific data. Not used under the Mac OS.
BufferProcs *	bufferProcs	Buffer callback suite. See chapter 3.
ResourceProcs *	resourceProcs	Pseudo–Resource callback suite. See chapter 3.
ProcessEventProc	processEvent	`ProcessEvent` callback. See chapter 3.
DisplayPixelsProc	displayPixels	`DisplayPixels` callback. See chapter 3.
HandleProcs *	handleProcs	Handle callback suite. See chapter 3.
ColorServicesProc	colorServices	`ColorServices` callback suite. See chapter 3..
GetPropertyProc	getProperty	Obsolete Property suite. This direct callback has been replaced by `PropertyProcs` (see below), but is maintained here for backwards compatibility.

Table 5–2: ExportRecord structure (Continued)

Type	Field	Description
AdvanceStateProc	advanceState	The `advanceState` callback allows you to drive the interaction through the inner `exportSelectorContinue` loop without actually returning from the plug–in. If it returns an error, then the plug–in generally should treat this as a continue error and pass it on when it returns.
For documents with transparency, the Export module is passed the merged data together with the layer mask for the current target layer. This information is contained in the following fields:		
int16	layerPlanes	This field contains the number of planes of data possibly governed by a transparency mask.
int16	transparencyMask	This field contains 1 or 0 indicating whether the data is governed by a transparency mask.
int16	layerMasks	This field contains the number of layers masks (currently 1 or 0) for which 255 = fully opaque. In Photoshop 3.0.4+, layer masks are not visible to Export modules since they are layer properties rather than document properties.
int16	invertedLayerMasks	This field contains the number of layers masks (currently 1 or 0) for which 255 = fully transparent. In Photoshop 3.0.4+, layer masks are not visible to Export modules since they are layer properties rather than document properties.
int16	nonLayerPlanes	This field contains the number of planes of non–layer data, e.g., flat data or alpha channels. The planes are arranged in that order. Thus, an RGB image with an alpha channel and a layer mask on the current target layer would appear as: red, green, blue, transparency, layer mask, alpha channel
These fields are new since version 3.0.4 of Adobe Photoshop.		
ImageServicesProcs *	imageServicesProcs	Image Services callback suite. See chapter 3.
int16	tileWidth	The host sets the width and height of the tiles. Best size for you to work in, if possible.
int16	tileHeight	
Point	tileOrigin	The origin point of the tiling system.
PropertyProcs *	propertyProcs	Property callback suite. See chapter 3.
These fields are new since version 4.0 of Adobe Photoshop.		
PIDescriptorParameters *	descriptorParameters	Descriptor suite. See chapter 3.
Str255 *	errorString	If you return with result=`errReportString` then whatever string you store here will be displayed as: "Cannot complete operation because *string*".
ChannelPortProcs *	channelPortProcs	Channel Ports callback suite. See chapter 3.
ReadImageDocumentDesc	documentInfo	The Channel Ports document information.
These fields are new since version 5.0 of Adobe Photoshop.		
SPBasicSuite *	sSPBasic	PICA basic suite.
void *	plugInRef	Plugin reference used by PICA.
int32	transparentIndex	If IndexedColor, and < 256, this is the index of the transparent color (for GIF).

Table 5–2: ExportRecord structure (Continued)

Type	Field	Description
Handle	iCCprofileData	Handle containing the ICC profile for the image, (NULL if none). Photoshop allocates the handle using Photoshop's handle suite. The handle is unlocked while calling the plug-in. The handle will be valid from Start to Finish. Photoshop will free the handle after Finish.
int32	iCCprofileSize	Size of profile.
int32	canUseICCProfiles	Non-zero if the host can accept/export ICC profiles. If this is zero, you'd better not set or dereference iCCprofileData.
These fields are new since version 5.5 of Adobe Photoshop.		
int32	lutCount	Number of entries in the indexed color table. This should include the transparent index if any. Plug-ins should pad out the color table with black for backword compatibility.
These fields are new since version CS (8.0) of Adobe Photoshop.		
int32	HostSupports32BitCoordinates	set by host if the host supports 32 bit plugin API
int32	PluginUsing32BitCoordinates	set to nonzero by the plugin if it is using the 32 bit fields
VPoint	imageSize32	Size of image in 32 bit coordinates replaces imageSize
VRect	theRect32	Rectangle being returned in 32 bit coordinates replaces theRect
VRect	selectBBox32	Rectangle being returned in 32 bit coordinates replaces selectBBox
char[106]	reserved	Reserved for future use. Set to zero.

8. Filter Modules

Filter plug-in modules modify a selected area of an image, and are accessed under the **Filter** menu. Filter actions range from subtle shifts of hue or brightness, to wild changes that create stunning visual effects.

Table 6–1: Filter file types

OS	Filetype/extension
Mac OS	8BFM
Windows	.8BF

SampleCode/Filter/Dissolve

Dissolve is a sample filter plug–in which demonstrates the use of accessing pixel data using a tilling scheme. In previous versions of the SDK there was the combination of Dissolve-Sans and Dissolve-With plug ins. This combines those two plug ins into one. Using the tilling scheme has increased the speed of this plug in by 2.

SampleCode/Filter/Propetizer

Propetizer is a utility filter that demonstrates different properties.

SampleCode/Filter/ColorMunger

ColorMunger is a utility filter that exercises the Color Services callback suite.

SampleCode/Filter/Hidden

Hidden is a utility filter for the AutomationFilter project found in the automation folder. Automation plug ins cannot access pixel memory directly but they can call Filter plug ins. The Hidden plug in allows the AutomationFilter plug in to read and write pixel data via the scripting system. See the AutomationFilter project for more details.

SampleCode/Filter/MFCPlugin

MFCPlugin is an example of using MFC in a Photoshop plug in. This is not a recommended way to implement your plug in. You only get the Microsoft operating system, memory management in MFC goes against using Photoshop call backs for acquiring memory, etc.

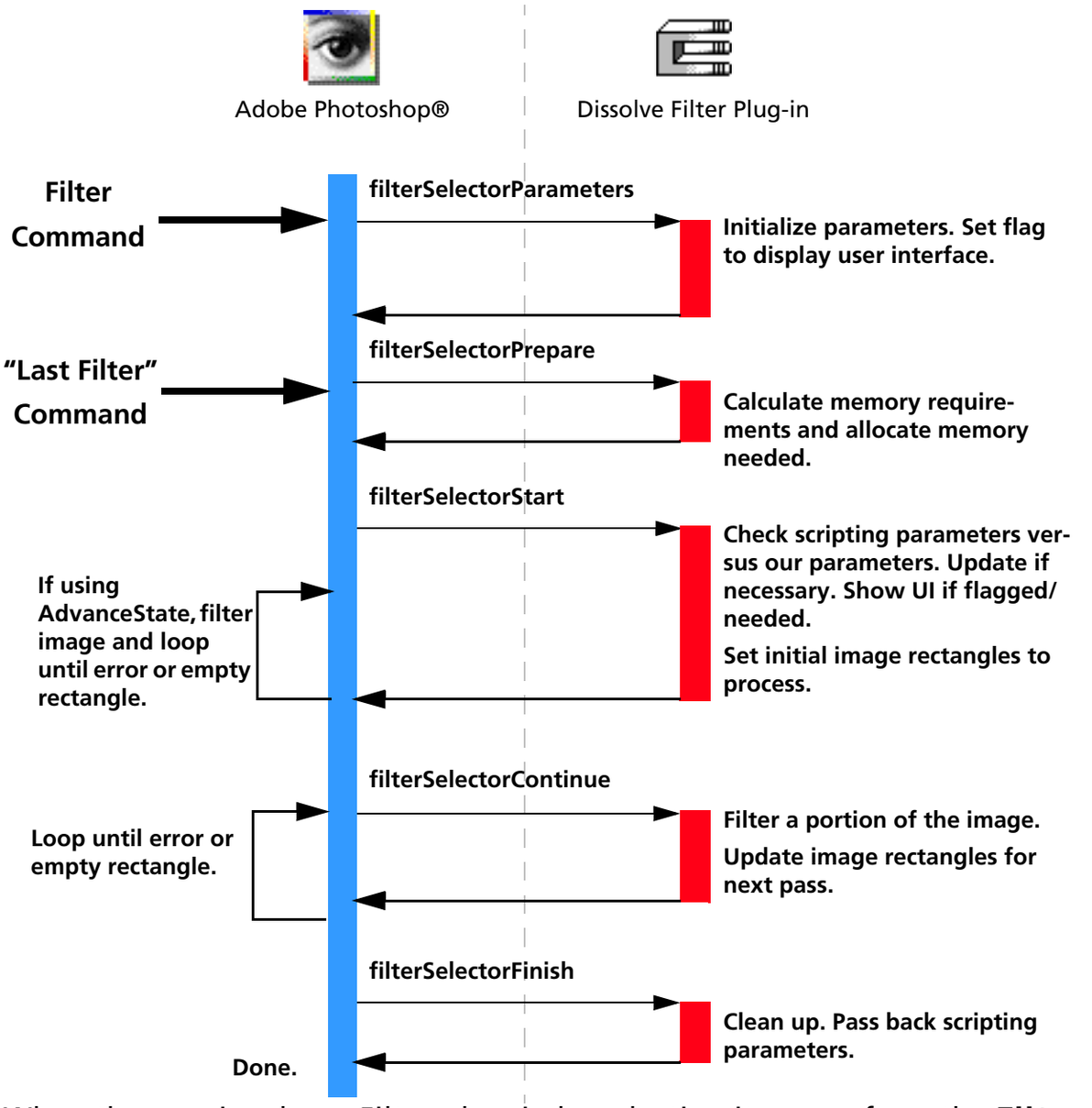
SampleCode/Filter/PoorMansTypeTool

PoorMansTypeTool is an example of using the Channel Ports Suite to read and write pixel data. This interface is much easier to understand than the AdvanceState mechanism that the other filter plug ins use. NOTE: The Channel Ports Suite is not implemented on other host adapters. Notably the ImageReady interface does not support the Channel Ports Suite.

SampleCode/Filter/Shell

Shell is a starting place for developing a plug in. There is a PiPL and an entry point and nothing else.

Calling sequence



When the user invokes a Filter plug–in by selecting its name from the **Filter** menu, Adobe Photoshop calls it with the sequence of selector values shown in the figure above. The actions for these selectors is discussed next.

filterSelectorParameters

If the plug–in filter has any parameters that the user can set, it should prompt the user and save the parameters in a relocatable memory block whose handle is stored in the parameters field. Photoshop initializes the parameters field to `NULL` when starting up.

This routine may or may not be called depending on how the user invokes the filter. After a filter has been invoked once, the user may re–apply that same filter with the same parameters. This is the "Last Filter" command in the **Filter** menu. When Last Filter is selected, the plug–in host does not call `filterSelectorParameters`, and the user will not be shown any dialogs to enter new parameters. Due to this, always check, validate, and initialize if necessary, your parameters handle in `filterSelectorStart` before using it.

Note: Your plug–in should validate the contents of its parameter handle whenever it starts processing if there is a danger of it crashing from bad parameters.

Since the same parameters can be used on different size images, the parameters should not depend on the size or mode of the image, or the size of the filtered area (these fields are not even defined at this point).

The parameter block should contain the following information:

1. A *signature* so that the plug–in can do a quick confirmation that this is, in fact, one of its parameter blocks.

2. A *version number* so that the plug–in can evolve without requiring a new signature.

3. A convention regarding byte–order for cross–platform support (or a flag to indicate what byte order is being used).

You may wish to design your filter so that you store default values or the last used set of values in the filter plug-in's Mac OS resource fork or a Windows external file. This way, you can save preference settings for your plug–in that will be consistent every time the host application runs.

Parameter block and scripting
The scripting system passes its parameters at every selector call. While it is possible to use the scripting system to store all your parameters, for backwards compatibility, it is recommended you track your parameters with your own parameter block. Once your parameter structure is validated, you should read your scripting-passed parameters and override your structure with them.

The most effective way to do this is:

1. First call a *ValidateMyParameters* routine to validate (and initialize if necessary) your global parameters.

2. Then call a *ReadScriptingParameters* routine to read the scripting parameters and then write them to your global parameters structure.

This way, the scripting system overrides your parameters, but you can use the initial values if the scripting system is unavailable or has parameter errors, and you can use your global parameters to pass between your functions.

filterSelectorPrepare

The *filterSelectorPrepare* selector calls allow your plug–in module to adjust Photoshop's memory allocation algorithm. The "Last Filter" command initially executes this selector call first.

Photoshop sets `maxSpace` to the maximum number of bytes it can allocate to your plug–in. You may want to reduce `maxSpace` for increased efficiency. Refer to chapter 3 for details on memory management strategies.

imageSize or imageSize32, planes & filterRect or filterRect32
The fields such as *imageSize or imageSize32*, *planes*, and *filterRect or filterRect32*, have now been defined, and can be used in computing your buffer size requirements. Refer to table 8-1 for more detail.

bufferSpace
If your plug–in filter module is planning on allocating any large buffers or tables over 32k, you should set the *bufferSpace* field to the number of bytes you are planning to allocate. Photoshop will then try to free up that amount of memory before calling the plug–in's `filterSelectorStart` handler.

Alternatively, you can set this field to zero and use the buffer and handle suites if they are available. See chapter 2 and 3 for a discussion on `maxSpace` and `bufferSpace`.

filterSelectorStart

At filterSelectorStart you should validate your parameters block, update your parameters based on the passed scripting parameters, and show your user interface, if requested. Then drop into your processing routine.

advanceState and filterSelectorStart

If you're using `AdvanceState`, the core of your filter may occur in this routine. Once done processing, set `inRect=outRect=maskRect=NULL`.

If you are not using `AdvanceState`, then you should initialize your processing and set-up the first chunk of image to be manipulated in `filterSelectorContinue`.

inRect, outRect & maskRect see also BigDocumentStruct

Your plug–in should set *inRect* and *outRect* (and *maskRect*, if it is using the selection mask) to request the first areas of the image to work on.

If at all possible, you should process the image in pieces to minimize memory requirements. Unless there is a lot of startup/shutdown overhead on each call (for example, communicating with an external DSP), tiling the image with rectangles measuring 64x64 to 128x128 seems to work fairly well.

Tiling, as opposed to row-oriented or column-oriented processing, also seems to be more operable for multi-processors. Multi-processors take well to spawning multiple separate threads, each processing a tile, but have a hard time (if at all) with rows or columns.

filterSelectorContinue

Your *filterSelectorContinue* handler is called repeatedly as long as at least one of the `inRect`, `outRect`, or `maskRect` fields is not empty.

inData, outData & maskData

Your handler should process the data pointed by *inData* and *outData* (and possibly *maskData*) and then update `inRect` and `outRect` (and `maskRect`, if using the selection mask) to request the next area of the image to process.

filterSelectorFinish

This call allows the plug–in to clean up after a filtering operation. This call is made if and only if the `filterSelectorStart` handler returns without error, even if the `filterSelectorContinue` routine returns an error.

If Photoshop detects *Command–period* in the Mac OS or *Escape* in Windows between calls to the `filterSelectorContinue` routine, it will call the `filterSelectorFinish` routine.

Note: Be careful processing user-cancel events during `filterSelectorContinue`. Normally your plug-in would be expecting another `filterSelectorContinue` call. If the user cancels, the next call will be `filterSelectorFinish`, *not* `filterSelectorContinue`!

Scripting at filterSelectorFinish

If your plug-in is scripting-aware and you've changed any initial parameters, you should pass a complete descriptor back to the scripting system in the `PIDescriptorParameters` structure.

Behavior and caveats

If `filterSelectorStart` succeeds, then Photoshop guarantees that `filterSelectorFinish` will be called.

Photoshop may call `filterSelectorFinish` instead of `filterSelectorContinue` if it detects a need to terminate while fulfilling a request.

`advanceState` may be called from either `filterSelectorStart` or `filterSelectorContinue` and will drive Photoshop through the buffer set up code. If the rectangles are empty, the buffers will simply be cleared. Termination is reported as `userCanceledErr` in the result from the `advanceState` call.

Error return values

The plug–in module may return standard operating system error codes, or report its own errors, in which case it can return any positive integer. These errors and more detail is available in `Examples/CIncludes/PIFilter.h`.

```
#define filterBadParameters     -30100     // a problem with the interface

#define filterBadMode           -30101     // a module doesn't support <mode> images
```

Big Document support

Photoshop CS (8.0) allows editing of documents beyond the 30,000 by 30,000 pixel limit in earlier versions. A new structure BigDocumentStruct was added to the FilterRecord to support this. This structure deprecates `imageSize, filterRect, inRect, outRect, maskRect, floatCoord and wholeSize`.

Table 6–2: BigDocumentStruct

Type	Field	Description
int32	PluginUsing32BitCoordinates	set to nonzero by the plugin if it si using the 32 bit fileds
VPoint	imageSize32	Size of the image in 32 bit coordinates, repleces imageSize
VRect	filterRect32	Rectangle to filter in 32 bit coordinates replaces filterRect
VRect	inRect32	Requested input rectangle in 32 bit coordinates replaces inRect
VRect	outRect32	Requested output rectangle in 32 bit coordinates replaces outRect
VRect	maskRect32	Requested mask rectangle in 32 bit coordinates replaces maskRect
VPoint	floatCoord32	Top left coordinate of selection in 32 bit coordinates replaces floatCoord
VPoint	wholeSize32	Size of image the selection is over in 32 bit coordinates replaces wholeSize

The Filter parameter block

The `pluginParamBlock` parameter passed to your plug–in module's entry point contains a pointer to a `FilterRecord` structure with the following fields. This structure is declared in `PIFilter.h`.

Table 6–3: FilterRecord structure

Type	Field	Description
int32	serialNumber	This field contains Adobe Photoshop's serial number. Your plug–in module can use this value for copy protection, if desired.
TestAbortProc	abortProc	This field contains a pointer to the `TestAbort` callback documented in chapter 3.
ProgressProc	progressProc	This field contains a pointer to the `UpdateProgress` callback documented in chapter 3. This procedure should only be called during the actual main operation of the plug–in, not during long operations during the preliminary user interface such as building a preview.
Handle	parameters	Photoshop initializes this handle to `NULL` at startup. If your plug–in filter has any parameters that the user can set, you should allocate a relocatable block in your `filterSelectorParameters` handler, store the parameters in the block, and store the block's handle in this field.
Point	imageSize	The image's width, `imageSize.h`, and height, `imageSize.v`, in pixels. If the selection is floating, this field instead holds the size of the floating selection. See imageSize32 below for large document support.
int16	planes	For version 4+ filters, this field contains the total number of active planes in the image, including alpha channels. The image mode should be determined by looking at `imageMode`. For version 0-3 filters, this field will be equal to 3 if filtering the RGB channel of an RGB color image, or 4 if filtering the CMYK channel of a CMYK color image. Otherwise it will be equal to 1.
Rect	filterRect	The area of the image to be filtered. This is the bounding box of the selection, or if there is no selection, the bounding box of the image. If the selection is not a perfect rectangle, Photoshop automatically masks the changes to the area actually selected (unless the plug–in turns off this feature using autoMask). This allows most filters to ignore the selection mask, and still operate correctly. See filterRect32 below for large document support.
RGBColor	background	The current background and foreground colors. If planes is equal to 1, these will have already been converted to monochrome. (Obsolete: Use `backColor` and `foreColor`.)
RGBColor	foreground	

Table 6–3: FilterRecord structure (Continued)

Type	Field	Description
int32	maxSpace	This lets the plug–in know the maximum number of bytes of information it can expect to be able to access at once (input area size + output area size + mask area size + `bufferSpace`).
int32	bufferSpace	If the plug–in is planning on allocating any large internal buffers or tables, it should set this field during the `filterSelectorPrepare` call to the number of bytes it is planning to allocate. Photoshop will then try to free up the requested amount of space before calling the `filterSelectorStart` routine.
Rect	inRect	Set this field in your `filterSelectorStart` and `filterSelectorContinue` handlers to request access to an area of the input image. The area requested must be a subset of the image's bounding rectangle. After the entire `filterRect` has been filtered, this field should be set to an empty rectangle.See inRect32 below for large document support.
int16	inLoPlane	Your `filterSelectorStart` and `filterSelectorContinue` handlers should set these fields to the first and last input planes to process next.
int16	inHiPlane	
Rect	outRect	Your plug–in should set this field in its `filterSelectorStart` and `filterSelectorContinue` handlers to request access to an area of the output image. The area requested must be a subset of `filterRect`. After the entire `filterRect` has been filtered, this field should be set to an empty rectangle.
int16	outLoPlane	Your `filterSelectorStart` and `filterSelectorContinue` handlers should set these fields to the first and last output planes to process next.
int16	outHiPlane	
void *	inData	This field contains a pointer to the requested input image data. If more than one plane has been requested (`inLoPlane`¦`inHiPlane`), the data is interleaved.
int32	inRowBytes	The offset between rows of the input image data. There may or may not be pad bytes at the end of each row.
void *	outData	This field contains a pointer to the requested output image data. If more than one plane has been requested (`outLoPlane`¦`outHiPlane`), the data is interleaved.
int32	outRowBytes	The offset between rows of the output image data. There may or may not be pad bytes at the end of each row.
Boolean	isFloating	This field is set `TRUE` if and only if the selection is floating.
Boolean	haveMask	This field is set `TRUE` if and only if a non–rectangular area has been selected.

Table 6–3: FilterRecord structure (Continued)

Type	Field	Description
Boolean	autoMask	By default, Photoshop automatically masks any changes to the area actually selected. If `isFloating=FALSE`, and `haveMask=TRUE`, your plug–in can turn off this feature by setting this field to `FALSE`. It can then perform its own masking. If you have set the PiPL bit `writesOutsideSelection`, this will always be `FALSE` and you must supply your own mask, if you want one.
Rect	maskRect	If `haveMask=TRUE`, and your plug–in needs access to the selection mask, your should set this field in your `filterSelectorStart` and `filterSelectorContinue` handlers to request access to an area of the selection mask. The requested area must be a subset of `filterRect`. This field is ignored if there is no selection mask.
void *	maskData	A pointer to the requested mask data. The data is in the form of an array of bytes, one byte per pixel of the selected area. The bytes range from (0...255), where 0=no mask (selected) and 255=masked (not selected). Use `maskRowBytes` to iterate over the scan lines of the mask.
int32	maskRowBytes	The offset between rows of the mask data.
FilterColor	backColor	The current background and foreground colors, in the color space native to the image.
FilterColor	foreColor	
OSType	hostSig	The plug–in host provides its signature to your plug–in module in this field. Photoshop's signature is `8BIM`.
HostProc	hostProc	If not `NULL`, this field contains a pointer to a host–defined callback procedure that can do anything the host wishes. Plug–ins should verify `hostSig` before calling this procedure. This provides a mechanism for hosts to extend the plug–in interface to support application specific features.
int16	imageMode	The mode of the image being filtered (Gray Scale, RGB Color, etc.). See `PIFilter.h` for values. Your `filterSelectorStart` handler should return `filterBadMode` if it is unable to process this mode of image.
Fixed	imageHRes	The image's horizontal and vertical resolution in terms of pixels per inch. These are fixed point numbers (16.16).
Fixed	imageVRes	
Point	floatCoord	The coordinate of the top–left corner of the selection in the main image's coordinate space.
Point	wholeSize	The size in pixels of the entire main image.
PlugInMonitor	monitor	This field contains the monitor setup information for the host. See Appendix A.

Table 6–3: FilterRecord structure (Continued)

Type	Field	Description
void *	platformData	This field contains a pointer to platform specific data. Not used under Mac OS.
BufferProcs *	bufferProcs	This field contains a pointer to the Buffer suite if it is supported by the host, otherwise `NULL`. See chapter 3.
ResourceProcs	*resourceProcs	This field contains a pointer to the Pseudo–Resource suite if it is supported by the host, otherwise `NULL`. See chapter 3.
ProcessEventProc	processEvent	This field contains a pointer to the `ProcessEvent` callback. It contains `NULL` if not supported. See chapter 3.
DisplayPixelsProc	displayPixels	This field contains a pointer to the `DisplayPixels` callback. It contains `NULL` not supported. See chapter 3.
HandleProcs *	handleProcs	This field contains a pointer to the Handle callback suite if it is supported by the host, otherwise `NULL`. See chapter 3.
These fields are new since version 3.0 of Adobe Photoshop.		
Boolean	supportsDummyPlanes	Does the host support the plug–in requesting non–existent planes? (see `dummyplane` fields, below) This field is set by the host to indicate whether it respects the dummy planes fields.
Boolean	supportsAlternateLayouts	Does the host support data layouts other than rows of columns of planes? This field is set by the plug–in host to indicate whether it respects the `wantLayout` field.
int16	wantLayout	The desired layout for the data. See `PIGeneral.h`. The plug–in host only looks at this field if it has also set `supportsAlternateLayouts`.
int16	filterCase	The type of data being filtered. Flat, floating, layer with editable transparency, layer with preserved transparency, with and without a selection. A zero indicates that the host did not set this field, and the plug-in should look at `haveMask` and `isFloating`.
int16	dummyPlaneValue	The value to store into any dummy planes. 0..255 = specific value. –1 = leave undefined.
void *	premiereHook	At one time was used for Adobe Premiere plug-in accessibility. Obsolete.
AdvanceStateProc	advanceState	The `AdvanceState` callback. See chapter 3.
Boolean	supportsAbsolute	Does the host support absolute channel indexing? Absolute channel indexing ignores visiblity concerns and numbers the channels from zero starting with the first composite channel. If existing, transparency follows, followed by any layer masks, then alpha channels.

Table 6–3: FilterRecord structure (Continued)

Type	Field	Description
Boolean	wantsAbsolute	Enable absolute channel indexing for the input. This is only useful if `supportsAbsolute=TRUE`. Absolute indexing is useful for things like accessing alpha channels.
GetPropertyProc	getProperty	The `GetProperty` callback. This direct callback pointer has been superceded by the Property callback suite, but is maintained here for backwards compatibility. See chapter 3.
Boolean	cannotUndo	If the filter makes a non–undoable change, then setting this field will prevent Photoshop from offering undo for the filter. This is rarely needed and usually frustrates users.
Boolean	supportsPadding	Does the host support requests outside the image area? If so, see padding fields below.
int16	inputPadding	The input, output, and mask can be padded when loaded. The options for padding include specifying a specific value (0...255), specifying `plugInWantsEdgeReplication`, specifying that the data be left random (`plugInDoesNotWantPadding`), or requesting that an error be signaled for an out of bounds request (`plugInWantsErrorOnBoundsException`). The error case is the default since previous versions would have errored out in this event.
int16	outputPadding	
int16	maskPadding	
char	samplingSupport	Does the host support non–1:1 sampling of the input and mask? Photoshop 3.0.1+ supports integral sampling steps (it will round up to get there). This is indicated by the value `hostSupportsIntegralSampling`. Future versions may support non–integral sampling steps. This will be indicated with `hostSupportsFractionalSampling`.
char	reservedByte	(for alignment)
Fixed	inputRate	The sampling rate for the input. The effective input rectangle in normal sampling coordinates is `inRect * inputRate`. For example, (`inRect.top * inputRate`, `inRect.left * inputRate`, `inRect.bottom * inputRate`, `inRect.right * inputRate`). `inputRate` is rounded to the nearest integer in Photoshop 3.0.1+. Since the scaled rectangle may exceed the real source data, it is a good idea to set some sort of padding for the input as well.
Fixed	maskRate	Like `inputRate`, but as applied to the mask data.
ColorServicesProc	colorServices	Function pointer to access color services routines. See chapter 3.

Table 6–3: FilterRecord structure (Continued)

Type	Field	Description
int16	inLayerPlanes	The number of planes (channels) in each category for the input data. This is the order in which the planes are presented to the plug–in and as such gives the structure of the input data. The inverted layer masks are ones where 0 = fully visible and 255 = completely hidden. If these are all zero, then the plug–in should assume the host has not set them.
int16	inTransparencyMask	
int16	inLayerMasks	
int16	inInvertedLayerMasks	
int16	inNonLayerPlanes	
int16	outLayerPlanes	The structure of the output data. This will be a prefix of the input planes. For example, in the protected transparency case, the input can contain a transparency mask and a layer mask while the output will contain just the layerPlanes.
int16	outTransparencyMask	
int16	outLayerMasks	
int16	outInvertedLayerMasks	
int16	outNonLayerPlanes	
int16	absLayerPlanes	The host sets these as the structure of the input data when `wantsAbsolute=TRUE`.
int16	absTransparencyMask	
int16	absLayerMasks	
int16	absInvertedLayerMasks	
int16	absNonLayerPlanes	
int16	inPreDummyPlanes	The number of extra planes before and after the input data. This is only available if `supportsDummyChannels=TRUE`. This is used for things like forcing RGB data to appear as RGBA.
int16	inPostDummyPlanes	
int16	outPreDummyPlanes	Like `inPreDummyPlanes` and `inPostDummyPlanes`, except it applies to the output data.
int16	outPostDummyPlanes	
int32	inColumnBytes	The step from column to column in the input. If using the layout options, this value may change from being equal to the number of planes. If zero, assume the host has not set it.
int32	inPlaneBytes	The step from plane to plane in the input. Normally 1, but this changes if the plug–in uses the layout options. If zero, assume the host has not set it.
int32	outColumnBytes	The output equivalent of `inColumnBytes` and `inPlaneBytes`.
int32	outPlaneBytes	
These fields are new since version 3.0.4 of Adobe Photoshop.		
ImageServicesProcs *	imageServicesProcs	This is a pointer to the Image Services callback suite. See chapter 3.
PropertyProcs *	propertyProcs	This is a pointer to the Property callback suite. See chapter 3.
int16	inTileHeight	The host will set the tiling for the input. Best to work at this size, if possible.
int16	inTileWidth	
Point	inTileOrigin	
int16	absTileHeight	The host will set the tiling for the absolute data. Best to work at this size, if possible.
int16	absTileWidth	
Point	absTileOrigin	

Table 6–3: FilterRecord structure (Continued)

Type	Field	Description
int16	outTileHeight	The host will set the tiling for the output. Best to work at this size, if possible.
int16	outTileWidth	
Point	outTileOrigin	
int16	maskTileHeight	The host will set the tiling for the mask. Best to work at this size, if possible.
int16	maskTileWidth	
Point	maskTileOrigin	
These fields are new since version 4.0 of Adobe Photoshop.		
PIDescriptorParameters *	descriptorParameters	Descriptor callback suite. See chapter 3.
Str 255 *	errorString	If you return with result=`errReportString` then whatever string you store here will be displayed as: "Cannot complete operation because *string*".
ChannelPortProcs *	channelPortProcs	Channel Ports callback suite. See chapter 3.
ReadImageDocumentDesc *	documentInfo	Suite for passing pixels through channel ports.
These fields are new since version 5.0 of Adobe Photoshop.		
SPBasicSuite *	sSPBasic	PICA basic suite.
void *	plugInRef	Plugin reference used by PICA.
int32	depth	Bit depth per channel (1,8,16).
These fields are new since version 6.0 of Adobe Photoshop.		
Handle	iCCprofileData	Handle containing the ICC profile for the image. (NULL if none) Photoshop allocates the handle using Photoshop's handle suite. The handle is unlocked while calling the plug-in. The handle will be valid from Start to Finish. Photoshop will free the handle after Finish.
int32	iCCprofileSize	size of profile
int32	canUseICCProfiles	non-zero if the host can export ICC profiles. If this is zero, you'd better not set or dereference iCCprofileData.
These fields are new since version 7.0 of Adobe Photoshop.		
int32	hasImageScrap	No-zero if Photoshop has image scrap. Plug-in can ask for the exporting of image scrap by setting the PiPL resource, WantsScrap. The document info for the image scrap will be chained right behind the targeted document pointed by the documentInfo field. hasScrap will be set to indicate if the image scrap is available. A plug-in can use it to tell whether Photoshop failed to export the scrap because some unknown reasons or there is no scrap at all.
These fields are new since version CS (8.0) of Adobe Photoshop.		
BigDocumentStruct *	bigDocumentData	support for documents larger than 30,000 pixels NULL if host does not support big documents see definition of BigDocumentStruct
char[46]	reserved	Reserved for future use. Set to zero.

10. Selection Modules

Selection plug-in modules modify the pixels and paths selected, and are accessed under the **Selection** menu.

Table 7–1: Selection file types

OS	Filetype/extension
Mac OS	`8BSM`
Windows	`.8BS`

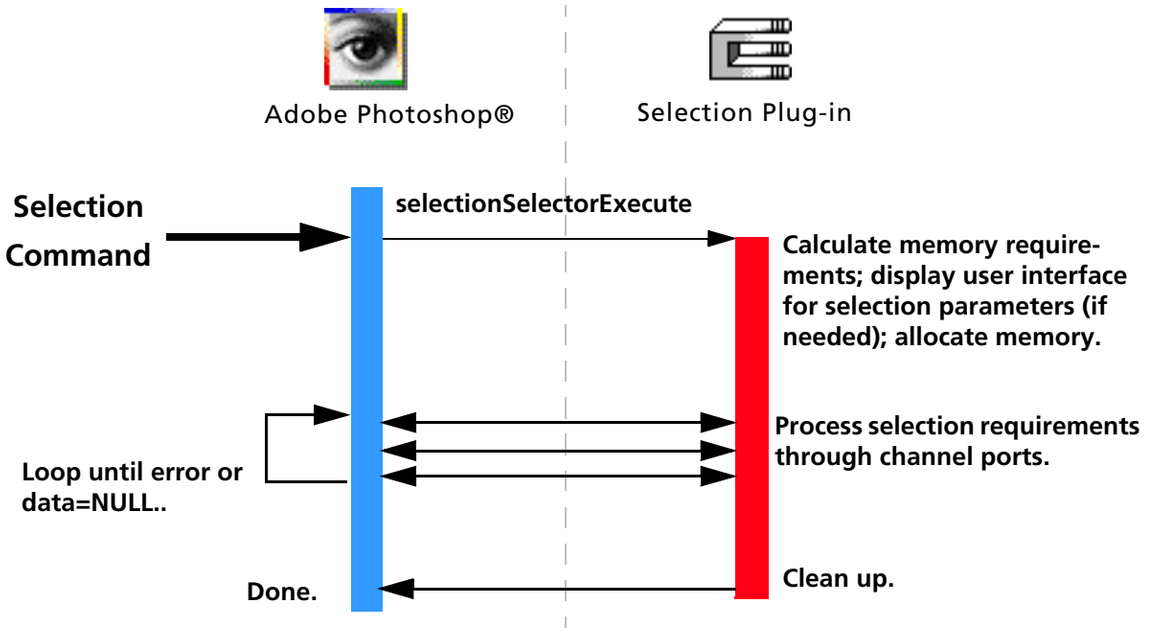
SampleCode/Selection/Selectorama

Selectorama is a sample selection plug–in which demonstrates pixel selection based on certain criteria.

SampleCode/Selection/Shape

Shape is a sample selection plug–in which demonstrates creating paths.

Calling sequence



When the user invokes a Selection plug–in by selecting its name from the **Plug-ins** sub-menu of the **Selection** menu, Adobe Photoshop calls it once with `selectionSelectorExecute`. The recommended sequence of actions for your plug-in to take is described next.

selectionSelectorExecute

Unlike other plug-ins, a Selection Module only gets one execute call, and is expected to do all the work during that call. However, it's recommended you use a similar process model:

1. Prompt for parameters

If the plug–in has any parameters that the user can set, it should prompt the user and save the values.

Your plug–in should validate the contents of its playback parameters when it starts processing if there is a danger of it crashing from bad parameters.

You may wish to design your plug-in so that you store default values or the last used set of values in the plug-in's Mac OS resource fork or a Windows external file. This way, you can save preference settings for your plug–in that will be consistent every time the host application runs. You may also use the scripting system as a way to store your parameters. They will be passed to you at `selectionSelectorExecute`, whether recording, playing back, or neither.

2. Allocate memory

Use the Buffer and Handle Suites to allocate any extra memory needed for your computations. See chapter 2 and 3 for a discussion on `maxData` and `bufferSpace`.

3. Begin your main loop

Your plug–in should call *readPixels* to request the first areas of the image to work on.

If at all possible, you should process the image in pieces to minimize memory requirements. Unless there is a lot of startup/shutdown overhead on each call (for example, communicating with an external DSP), tiling the image with rectangles measuring 64x64 to 128x128 seems to work fairly well.

4. Modify, write the results, continue until done.
Make your adjustments then call *WriteBasePixels*. Then continue looping until you've implemented your entire selection or path.

5. Finish and clean up
Clean up after your operation. Dispose any handles you created, etc.

Behavior and caveats

No behavior or caveats to note as of suite version 1.

Channel Ports structures

These structures are used by the Channel Ports callback suite and Selection modules.

Table 7–2: ReadImageDocumentDesc structure

Type	Field	Description
int32	minVersion	Minimum and maximum version required to interpret this record. Current `min=max=0`.
int32	maxVersion	
int32	imageMode	Color mode. See appendix A for valid image modes.
int32	depth	Default bit depth.
VRect	bounds	Document bounds.
Fixed	hResolution	Horizontal and vertical resolution.
Fixed	vResolution	
LookUpTable	redLUT	Color table for indexed color and duotone.
LookUpTable	greenLUT	
LookUpTable	blueLUT	
ReadChannelDesc *	targetCompositeChannels	Composite channels in the target layer. See table 7–3.
ReadChannelDesc *	targetTransparency	Transparency channel for the target layer.
ReadChannelDesc *	targetLayerMask	Layer mask for the target layer.
ReadChannelDesc *	mergedCompositeChannels	Composite channels in the merged data. Merged data is either merged layer data or merged document data.
ReadChannelDesc *	mergedTransparency	Transparency channel for the merged data.
ReadChannelDesc *	alphaChannels	Alpha channels for masks.
ReadChannelDesc *	selection	Selection mask, if any.

Table 7–3: ReadChannelDesc structure

Type	Field	Description
int32	minVersion	Minimum and maximum version required to interpret this record. Current `min=max=0`.
int32	maxVersion	
ReadChannelDesc *	next	Next descriptor in the list.
ChannelReadPort	port	Port to use for reading.
VRect	bounds	Channel data bounds.
int32	depth	Horizontal and vertical resolution.
VPoint	tileSize	Size of the tiles set by the host. Use this if you can to optimize to match the host's memory scheme.
VPoint	tileOrigin	Origin of the tiles, set by the host.

Table 7–3: ReadChannelDesc structure (Continued)

Type	Field	Description
Boolean	target	=TRUE if this is a target channel.
Boolean	shown	=TRUE if this channel is visible.
int16	channelType	The channel type. See table 7–5.
void *	contextInfo	Pointer to additional info dependent on context.
const char *	name	Name of the channel.

Table 7–4: WriteChannelDesc structure

Type	Field	Description
int32	minVersion	Minimum and maximum version required to interpret this record. Current min=max=0.
int32	maxVersion	
WriteChannelDesc *	next	Next descriptor in the list.
ChannelWritePort	port	Port to write to.
VRect	bounds	Channel data bounds.
int32	depth	Horizontal and vertical resolution.
VPoint	tileSize	Size of the tiles.
VPoint	tileOrigin	Origin of the tiles.
int16	channelType	The channel type. See table 7–5.
int16	padding	Reserved. Defaults to zero.
void *	contextInfo	Pointer to additional info dependent on context.
const char *	name	Name of the channel.

Table 7–5: Channel types

Field	Description
0=ctUnspecified	Unspecified channel.
1=ctRed	Red of RGB.
2=ctGreen	Green of RGB.
3=ctBlue	Blue of RGB.
4=ctCyan	Cyan of CMYK.
5=ctMagenta	Magenta of CMYK.
6=ctYellow	Yellow of CMYK.
7=ctBlack	Black of CMYK.
8=ctL	L of LAB.
9=ctA	A of LAB.
10=ctB	B of LAB.
11=ctDuotone	Duotone.
12=ctIndex	Index.
13=ctBitmap	Bitmap.
14=ctColorSelected	Selected color.
15=ctColorProtected	Protected color.
16=ctTransparency	Transparent color.

Table 7–5: Channel types (Continued)

Field	Description
17=ctLayerMask	Layer mask (alpha channel). White = transparent, Black = mask.
18=ctInvertedLayerMask	Inverted layer mask (inverted alpha channel).
19=ctSelectionMask	Mask/alpha for selection.

Treatments and SupportedTreatments

The `treatment` field indicates what a selection module is returnning. The supportedTreatments field is a mask indicating what the host supports.

Table 7–6: Treatments and SupportedTreatments

Name	Value
piSelMakeMask	0
piSelMakeWorkPath	1
piSelMakeLayer	2

Error return values

The plug–in module may return standard operating system error codes, or report its own errors, in which case it can return any positive integer. These errors and more detail is available in `PISelection.h`.

```
#define selectionBadParameters  -30700  // a problem with the interface
#define selectionBadMode        -30701  // module doesn't support <mode> images
```

The Selection parameter block

The `pluginParamBlock` parameter passed to your plug–in module's entry point contains a pointer to a `PISelectionParams` structure with the following fields. This structure is declared in `PISelections.h`.

Table 7–7: PISelectionParams structure

Type	Field	Description
int32	serialNumber	This field contains Adobe Photoshop's serial number. Your plug–in module can use this value for copy protection, if desired.
TestAbortProc	abortProc	This field contains a pointer to the `TestAbort` callback in chapter 3.
ProgressProc	progressProc	This field contains a pointer to the `UpdateProgress` callback documented in chapter 3. This procedure should only be called during the actual main operation of the plug–in, not during long operations during the preliminary user interface such as building a preview.
OSType	hostSig	The plug–in host provides its signature to your plug–in module in this field. Photoshop's signature is `8BIM`.
HostProc	hostProc	If not `NULL`, this field contains a pointer to a host–defined callback procedure that can do anything the host wishes. Plug–ins should verify `hostSig` before calling this procedure. This provides a mechanism for hosts to extend the plug–in interface to support application specific features.
BufferProcs *	bufferProcs	This field contains a pointer to the Buffer suite if it is supported by the host, otherwise `NULL`. See chapter 3.
ResourceProcs *	resourceProcs	This field contains a pointer to the Pseudo–Resource suite if it is supported by the host, otherwise `NULL`. See chapter 3.
ProcessEventProc	processEvent	This field contains a pointer to the `ProcessEvent` callback. It contains `NULL` if not supported. See chapter 3.
DisplayPixelsProc	displayPixels	This field contains a pointer to the `DisplayPixels` callback. It contains `NULL` not supported. See chapter 3.
HandleProcs *	handleProcs	This field contains a pointer to the Handle callback suite if it is supported by the host, otherwise `NULL`. See chapter 3.
ColorServicesProc	colorServices	Color services suite. See chapter 3.
ImageServicesProcs *	imageServicesProcs	Image Services suite. See chapter 3.
PropertyProcs *	propertyProcs	Property suite. See chapter 3.
ChannelPortProcs *	channelPortProcs	Channel ports suite. See chapter 3.
PIDescriptorParameters *	descriptorParameters	Descriptor suite. See chapter 3.
Str255	errorString	If you return with result=`errReportString` then whatever string you store here will be displayed as: "Cannot complete operation because *string*".
PlugInMonitor	monitor	Monitor setup info. See appendix A.
void *	platformData	Pointer to platform specific data. Not used in Mac OS.

Table 7–7: PISelectionParams structure (Continued)

Type	Field	Description
Boolean	hostSupportsPaths	Check this flag before returning a path. All host will clean up `newPath`.
char[3]	reserved	Reserved for future use. Set to zero.
ReadImageDocumentDesc *	documentInfo	The document for the selection. See table 7–2.
WriteChannelDesc *	newSelection	Output for new selection. See table 7–4.
Handle	newPath	If non-NULL then `newSelection` is ignored and the path described by this handle becomes the work path. Handle is disposed by host.
int32	treatment	Treatment for returned pixels/mask. See table 7–6.
int32	supportedTreatments	Mask indicating host supported treatments. See table 7–6.
These fields are new since version 5.0 of Adobe Photoshop.		
SPBasicSuite *	sSPBasic	PICA basic suite.
void *	plugInRef	Plugin reference used by PICA.
char[248]	reservedBlock	Reserved for future use. Set to zero.

11. Scripting Plug-ins

Adobe Photoshop 4.0 introduced a new palette and subsequent set of commands and callbacks: the *Actions* palette, and the *Descriptor* callback suite. The Actions palette is the user-interface and hub for the scripting system for Adobe Photoshop. Adobe Photoshop 5.0 extends the Actions structure to include automation plug-ins that can access all scriptable Photoshop commands.

Actions allow commands in Photoshop to be recorded in a form that is easy for an end user to read and edit. Actions are similar to AppleScript and AppleEvents but are cross platform and designed to support both AppleScript on the Macintosh and OLE Automation on Windows.

Actions extend the plug-in API to allow Import, Export, Filter, Format and Selection plug-in modules to be fully recordable and automated.

Scripting on Windows with OLE
While the scripting system operates consistently across both Windows and Macintosh platforms, in Photoshop 4.0 and 5.0, additional OLE automation has been added. Refer to the OLE Automation pdf file.

AppleScript and AppleEvents recommended reading
Since Actions are based on AppleScript and AppleEvents, we recommend the following materials for preliminary reading:

Inside Macintosh: Interapplication Communication (Addison-Wesley, 1993); "Apple Event Objects and You" (Richard Clark, *develop*, issue 10); "Better Apple Event Coding Through Objects" (Eric M. Berdahl, *develop*, issue 11); "Designing Scriptability" (Cal Simone, *develop*, issue 21); Series: "According to Script" (Cal Simone, *develop*, issues 22-25).

Issues of *develop* can be found online at:
`http://dev.info.apple.com/develop/developtoc.html`.

All the plug-in module examples that support scripting have been updated. Detailed code-related information is available in each separate module example and in `PIActions.h`.

Photoshop 5.0 Automation plug-ins

Refer to the document *Actions.pdf* for implementation details on Adobe Photoshop 5.0 Automation plug-in modules.

Scripting Basics

For a plug-in to be *scripting-aware*, or able to record scripting parameters and be automated by them, it requires the addition of two basic mechanisms:

1. **Terminology resource**. A *terminology resource* maps the keys to human readible text, providing additional type information for values. For instance, key `keyLuminance` ('`Lmnc`') and its value `typeInteger` may be mapped to the human readible text "luminance value". This is accompanied by the `HasTerminology` PiPL property, which points the scripting system to the terminology resource.

2. **Descriptors**. A *descriptor* is a pair of data in the form of [<*key*> <*value*>] that describes the property of an object or the parameter of an event.

Implementation order

We recommend you convert existing plug-ins to scripting-aware plug-ins by following this scripting implementation order:

1. Look at your user interfaces and describe the parameters as human-readable text;

2. Create a terminology resource for your plug-in;

3. Add the `HasTerminology` PiPL property;

4. Update your plug-in code to record scripting events and objects;

5. Update your plug-in code to be automated by (play back) scripting events and objects.

Scripting caveats

The scripting system has been designed specifically to drive plug-ins in a way that is transparent to the existing operation of the host. This means that there is no way to know whether your plug-in is being driven by the scripting system or an end-user. You should treat all operations as consistently as possible.

The scripting system always hands you a descriptor at every selector call.

If you use a descriptor that was handed to you by the host, and you hand back a new descriptor, you are responsible for deleting the old descriptor. All the examples do this through the set of utility routines in PIUtilities.

If you don't use the descriptor handed to you by the host, you may hand it back and it will be deleted automatically.

if you don't use the descriptor handed to you by the host, but you hand back `NULL`, then you are responsible for deleting the descriptor the host handed you.

Creating a terminology resource

A terminology resource is used to specify the mapping from a descriptor to human readable text. The format of the terminology resource is identical to an AppleEvent terminology resource. `CNVTPIPL.EXE` on Windows understands this resource and converts it accordingly. All the example plug-ins have a rez entry in the *plugInName*`.r` file for an '`aete`' resource.

To let Photoshop know that the terminology resource is present, a `PiPL` property is added, `HasTerminology` ('`hstm`'), which contains the class ID, event ID, and terminology resource ID for your plug-in. Refer to the *Plug-in Resource Guide.pdf* for information on Scripting-specific properties.

The terminology resource is a complex structure designed by Apple to cover numerous scripting situations that are not required by Photoshop. By that nature, the structure is more complicated than it needs to be to describe plug-ins. However, it was chosen because Apple plans to support it both now and in the future, and it allows you to increase the scope of your plug-in by being AppleEvent- and AppleScript-savvy. See the last section of this chapter for information on AppleScript.

Basic terminology resource

```
resource 'aete' (0)
{                                                   // aete version and language specifiers
    { /* suite descriptor */
        { /* filter/selection/color picker descriptor */
            { /* any parameters */
                / * additional parameters */
            }
        },
        { /* import/export/format descriptors */
            { /* properties. First property defines inheritance. */
                /* any properties */
            },
            { /* elements. Not supported for plug-ins. */
            },
            /* class descriptions for other classes used as parameters or properties */
        },
        { /* comparison ops. Not currently supported. */
        },
        { /* any enumerations */
            {
                /* additional values for enumeration */
            },
            /* any additional enumerations */
            /* variant types are a special enumeration: */
            {
                /* additional types for variant */
            },
            /* any additional variants */
            /* class and reference types are a special enumeration: */
            {
            },
            /* any additional class or reference types */
        }
    }
}
```

Whether your plug-in is a filter, or one of the others, each section of the terminology resource must be present (even if it's blank "{ } , ").

Detailed terminology resource

```
resource 'aete' (0)
{
    1, 0, english, roman,                   // aete version and language specifiers
    { /* suite descriptor below */
        "suite name",                       // name of suite
        "description",                      // optional description of suite
        'stID',                             // suite ID, must be unique 4-char code
        1,                                  // suite code, must be 1
        1,                                  // suite level, must be 1
        { /* filter/selection/color picker descriptor below */
            "plug-in name",                 // name of plug-in, must be unique
            "description",                  // optional description of filter
            'clID',                         // class ID, must be unique 4-char code or suite ID
            'evID',                         // event ID, must be unique 4-char code within class
                                               (may be suite ID)
            NO_REPLY,                       // never a reply
            IMAGE_DIRECT_PARAMETER          // direct parameter
            { /* any parameters below */
                "parameter name",           // name of parameter
                'kyID',                     // parameter key ID. See table 8–14.
                'tyID',                     // parameter type ID. See table 8–4.
                flagsTypeParameter,         // parameter flags. See table .
                / * additional parameters here */
            }
        },
        { /* import/export/format descriptors below */
            "plug-in name",                 // name of plug-in, must be unique
            'clID',                         // class ID, must be unique 4-char code or suite ID
            "description",                  // optional description of plug-in
            { /* properties below. First property defines inheritance. */
                "<Inheritance>",            // required
                keyInherits,                // required
                classInherited,             // parent class: Format, Import, or Export
                "",
                flagsSingleParameter,
                /* any properties below */
                "property name",            // name of property
                'kyID',                     // property key ID. See table 8–14.
                'tyID',                     // property type ID. See table 8–4.
                "description",              // optional description
                flagsTypeParameter,         // property flags. See table .
            },
            { /* elements. Not supported for plug-ins. */
            },
            /* class descriptions for other classes used as parameters or properties */
        },
        { /* comparison ops. Not currently supported. */
        },
        { /* any enumerations below */
            'enID',                         // enumeration ID
            {
                "enumerated name",          // first value name
```

```
            'e1ID',                 // first value ID
            "description",          // optional description of first value
            /* additional values for this enumeration */
        },
        /* any additional enumerations */
        /* variant types are a special enumeration: */
        '#vID',                     // variant ID (must begin with "#")
        {
            "type name",            // first type name
            'v1ID',                 // first type ID
            "",
            /* additional types for variant */
        },
        /* any additional variants */
        /* class and reference types are a special enumeration: */
        '#tID',                     // enumeration ID (must begin with "#")
        {
            "type name",            // name of type
            't1ID',                 // type ID. Either typeClass or typeObjectReference.
            ""
        },
        /* any additional class or reference types */
    }
  }
}
```

Nomenclature

The user terms in the terminology resource should be all lower case with the exception of proper names and acronyms. Photoshop will capitalize terms appropriately.

Parameter and property flags

The AppleScript Editor doesn't display parameter type `list` correctly. In order for the dictionary to read correctly, the `description` field for the type should begin with the word "`list`".

The flags for properties are the same for parameters, except there is no flag for `optional`. Properties can be optional by putting "`optional`" at the beginning of the `description` field.

Table 8–1: Valid terminology resource parameter and property flags

Name	Alternates	Description
flagsSingleParameter flagsSingleProperty	flagsOptionalSingleParameter	Key and value type is single value.
flagsListParameter flagsListProperty	flagsOptionalListParameter flagsEnumeratedListParameter flagsEnumeratedListProperty flagsOptionalEnumeratedListParameter	Key and value type is for a list.
flagsEnumeratedParameter flagsEnumeratedProperty	flagsOptionalEnumeratedParameter	Key and value type is for enumerated list.

Classes and the terminology resource

For Import, Export, and Format plug-ins, one of the classes must be the base class for the plug-in. Additional classes may be defined as templates for parameters or properties.

Table 8–2: Predefined classes

Name	Code	Description/keys
classImport	'Impr'	Class for Import modules.
classExport	'Expr'	Class for Export modules.
classFormat	'Fmt '	Class for Format modules.
classColor	'Clr '	Class for color classes.
classRGBColor	'RGBC'	keyRed, keyGreen, keyBlue
classCMYKColor	'CMYC'	keyCyan, keyMagenta, keyYellow, keyBlack.
classUnspecifiedColor	'UnsC'	Unspecified.
classGrayscale	'Grsc'	keyGray
classBookColor	'BkCl'	Book color.
classLabColor	'LbCl'	keyLuminance, keyA, keyB.
classHSBColor	'HSBC'	keyHue, keySaturation, keyBrightness.
classPoint	'Pnt '	keyHorizontal, keyVertical.

Inheritance

Inheritance can also be used to specify a hierarchy of types. Inheritance is used by defining a base class with the first property configured with:

1. Name = the name of the class;
2. Type = class type.

Class types are defined by creating a special enumeration. If the class *color* is specified as a parameter or property type, then any of its sub-classes are acceptable. The class `color` is defined:

```
{ /* suite descriptor below */
   "color",                    // class name
   classColor,                 // class ID for Color 'Clr '
   "",                         // no description
   {
      "color",                 // color property (special for base class)
      keyColor,                // property ID for Color 'Clr '
      typeClassColor,          // type this class
      "",                      // no description
      flagsEnumeratedParameter // “type” is special enumeration
   },
   { /* no elements */
   }
```

The class *RGB color* is defined:

```
   "RGB color",                // class name
   classRGBColor,              // class ID 'RGBC'
   "",                         // no description
   {
      "<Inheritance>",         // define inheritance
      keyInherits,             // property ID for inheritance 'Clr '
      classColor,              // from parent class “color”
      "",                      // no description
      flagsSingleParameter     // single parameter

      "red",                   // red property
```

```
        keyRed,                     // property ID for Red 'Rd  '
        typeFloat,                  // value type “float”
        "",                         // no description
        flagsSingleParameter        // single parameter

        "green",                    // green property
        keyGreen,                   // property ID for Green 'Grn '
        typeFloat,                  // value type “float”
        "",                         // no description
        flagsSingleParameter        // single parameter

        "blue",                     // blue property
        keyBlue,                    // property ID for Blue 'Bl  '
        typeFloat,                  // value type “float”
        "",                         // no description
        flagsSingleParameter        // single parameter
    },
    {  /* no elements */
    }
```

Enumerated types

Enumerated types are used in the standard fashion to create a type that can have one or a set of values.

Note:
For the enumerated value IDs, as tempting as it may be, don’t use simple indexes, use four-character types.

An enumerated type for *quality* with the values of *low*, *medium*, *high*, and *maximum* is defined:

```
    typeQuality,                    // type ID for Quality 'Qlty'
    {
        "low",                      // “low” value
        enumLow,
        "",

        "medium",                   // “medium” value
        enumMedium,
        "",

        "high",                     // “high” value
        enumHigh,
        "",

        "maximum",                  // “maximum” value
        enumMaximum,
        "",
    }
```

Variant types

Enumerated types are also used to specify variant types for parameters and properties.

Note:
The first character of a variant type ID must be "#".

If you have a parameter that can take text or an integer, it is defined:

```
"specifier",                    // parameter name
keySpecifier,                   // parameter ID
typeTextInteger,                // text or integer
"index or name",                // short description
flagsEnumeratedParameter
```

The type `typeTextInteger` is an enumeration:

```
typeTextInteger,                // type ID (variant types must begin with "#")
{
   "string",                    // name of first type (AppleScript name)
   typeText,
   "",

   "integer",                   // name of second type
   typeInteger,
   ""
}
```

Enumerations and object reference types

Enumeration variants can also be used to specify object reference types and class types. From the example of the class `color`, `typeClassColor` is defined:

```
typeClassColor,                 // type ID (variant types must begin with "#")
{
   "type color",                // name of type
   typeClass,                   // generic type reference
   "",
}
```

Lists and the terminology resource

All types can be used as lists for parameters and properties. All items in a list must be of the same type. To specify a list in the terminology resource use the `flagsListParameter` or `flagsListProperty` from table .

Descriptors

Because the Actions palette provides an alternate, text-based user interface to Adobe Photoshop, textual script commands need to map intuitively to the graphical user interface. The way to start developing a scripting interface for your plug-in is to look carefully at the options provided in your dialogs, and then describe them in writing.

For some options, such as checkboxes and popup menus, this is fairly straight-forward. For others, such as showing placement of an object graphically, this is more difficult.

All scripting commands are described with the following form:

event [*target*] [<*key*> <*value*>]

Table 8–3: Scripting command syntax

Name	Description
event	Command being executed.
target	Item being acted upon.
key	Parameter key. See table 8–14.
value	Parameter value type. See table 8–4.

Table 8–4: Basic value types

Name	Code	Description
typeInteger	`'long'`	int32.
typeFloat	`'doub'`	IEEE 64 bit double
typeBoolean	`'bool'`	`TRUE` or `FALSE`.
typeText	`'TEXT'`	Block of any number of readable characters.
typeAlias	`'alis'`	Macintosh file system path.
typePaths	`'Pth '`	Windows file system path.
typePlatformFilePath	`'alis'` or `'Pth '`	`typeAlias` for Macintosh, `typePath` for Windows.

Table 8–5: Special value types

Name	Code	Description
type*Enumeration*		Enumeration declared in the terminology resource.
typeClass	`'char'`	Used in terminology resource for class type specifier.
typeObjectReference	`'indx'`	Refers to Photoshop object, such as channel or layer. See below.
class*Class*		Enumerated class. See table 8–2.

Filter, Selection, and Color Picker events

Filter, Selection, and Color Picker scripting is described as “scripting events”:

filter [*target*] [<*key*> <*value*>]

Such as:

```
gaussian blur layer 1 radius 5
```

Table 8–6: Scripting event command syntax

Name	Description
filter	Menu name of the filter plug-in, or similar. (“`gaussian blur`”)
target	Portion of the document to apply filter. (“`layer 1`”)
key	Parameter key. (“`radius`”)
value	Parameter value. (“`5`”)

Import, Export, and Format objects

Import, Export, and Format scripting is described as “scripting objects”:

command [*target*] [as *type* | *object* | *string*] [in *file* | *folder*]

Such as:

```
save document “bright banana” as “Shareware”
save document 1 as Photoshop EPS
save as Photoshop EPS { preview 8 bit TIFF }
```

Table 8–7: Script object command syntax

Name	Description
command	Operation: “`save`” or “`open`”. (“`save`”)
target	Document. (“`document 1`”)
as	type = type of class (“`TIFF`”) object = object of class (“`Photoshop EPS`”) string = name of non-scriptable format (“`Shareware`”)
in	Location to save/load the file.

In the example “`save as Photoshop EPS { preview 8 bit TIFF }`” the target document is being saved with the “`Photoshop EPS`” format with the parameter: key “`preview`”, value “`8 bit TIFF`”.

Note: This form is a little different then AppleScript. In AppleScript, the “`save as Photoshop EPS`” example would appear as:

```
save as {format: Photoshop EPS, preview: 8 bit TIFF}
```

In Photoshop the object’s class is implied by the object passed since the scripting mechanism has a stronger type system than AppleScript.

Save as object

To save as an object, the nomenclature is `save as` *classFormat*, where *classFormat*, is generally a class type with parameters, such as `JPEG`, `PDF`, or `EPS`:

```
save as { class: JPEG, quality: 3 }
save as JPEG with properties { quality: 5 }
```

Save as type

Saving as a type takes the form of `save as` *typeClassFormat*, which is always a specific type, with no parameters:

```
save as JPEG
save as EPS
```

typeObjectReference

The type `typeObjectReference` is used to refer to an external object, such as a channel or layer. Plug-ins cannot access these objects directly but can use object references to refer to elements that are accessible through other means.

An object reference can refer to an object which is either an element or a property of another object. Elements may be referred to by name or index. Plug-ins can only refer to elements or properties of the immediate target, due to the one-dimensional nature of `PIDescriptorSimpleReference`. For example, your plug-in may specify:

```
channel 1
```

But it cannot specify:

```
channel 1 of layer 2
```

PIDescriptorSimpleReference

```
typedef struct PIDescriptorSimpleReference
    {
    DescriptorTypeID           desiredType;
    DescriptorFormID           keyForm;
    struct _keyData
       {
          Str255               name;
          uint32               index;
          DescriptorTypeID     type;
          DescriptorEnumID     value;
       } keyData;
    } PIDescriptorSimpleReference;
```

Table 8–8: PIDescriptorSimpleReference structure

Type	Field	Description
DescriptorTypeID	desiredType	`typeInteger`, `typeFloat`, etc. See table 8–4.
DescriptorFormID	keyForm	Item type=`formIndex`, `formName`, or `formProperty`.
struct	_keyData	`keyForm` specific info. See table 8–9.

Table 8–9: keyData structure

Type	Field	Description
Str255	name	if `keyForm=formName`, the name of the key.
uint32	index	if `keyForm=formIndex`, the index of the key.
DescriptorTypeID	type	if `keyForm=formEnumerated` the type and enumeration of the key.
DescriptorEnumID	value	

If a plug-in attempts to read a complex object reference (for instance, one containing other references) the host will attempt to simplify the reference; if it can't, it will return an error.

Scripting Parameters

Once you've added a terminology resource and you've edited the `HasResource` PiPL property (see the *Plug-in Resource Guide.pdf* file), your plug-in is considered *scripting-aware*. At every selector a structure is passed in the Descriptor Suite portion of the parameter block: `PIDescriptorParameters`. The suite of routines for Getting and Putting values from and to this structure is described in the chapter 3. You will access this data structure for Recording and Playback.

PIDescriptorParameters

```
typedef struct PIDescriptorParameters
{
    int16                       descriptorParametersVersion;
    int16                       playInfo;
    int16                       recordInfo;

    PIDescriptorHandle          descriptor;

    WriteDescriptorProcs*       writeDescriptorProcs;
    ReadDescriptorProcs*        readDescriptorProcs;
} PIDescriptorParameters;
```

Table 8–10: PIDescriptorParameters structure

Type	Field	Description
int16	descriptorParametersVersion	Minimum version required to process.
int16	playInfo	Flags for playback: 0=plugInDialogDontDisplay 1=plugInDialogDisplay 2=plugInDialogSilent
int16	recordInfo	Flags for recording: 0=plugInDialogOptional 1=plugInDialogRequired 2=plugInDialogNone
PIDescriptorHandle	descriptor	Handle to actual descriptor key/value pairs.
WriteDescriptorProcs*	writeDescriptorProcs	WriteDescriptorProcs sub-suite.
ReadDescriptorProcs*	readDescriptorProcs	ReadDescriptorProcs sub-suite.

Recording

Building a descriptor

If your plug-in has no options, `descriptor` may be set to `NULL`.

To build a descriptor:

1. Call `openWriteDescriptorProc` which will return a `PIWriteDescriptor` token, such as *writeToken*.
2. Call various Put routines such as `PutIntegerProc`, `PutFloatProc`, etc., to add key/value pairs to `writeToken`. The keys and value types must correspond to those in your terminology resource.
3. Call `CloseWriteDescriptorProc` with `writeToken`, which will create a `PIDescriptorHandle`.
4. Place the `PIDescriptorHandle` into the `descriptor` field. The host will dispose of it when finished.
5. Store your `recordInfo`. See table 8–11.

Table 8–11: recordInfo

Name	Description
plugInDialogOptional	Display dialog only if necessary or requested by user.
plugInDialogRequired	Always display dialog.
plugInDialogNone	No dialog.

Recording error handling

If an error occurs during or after `PIWriteDescriptor`, then `writeToken` and the new `PIDescriptorHandle` should be disposed of using `DisposePIHandleProc` from the Handle Suite.

Recording classes

Classes may be declared by plug-ins to be used as templates for structures. Classes delcared by plug-ins may not contain elements, but may use inheritance. Objects of a particular class are created by defining a descriptor and adding the key/value pairs for the properties. The root property of the base class is not added to the descriptor.

Playback

Reading a descriptor

If a plug-in has no options, or is not scripting-aware, `descriptor` will be `NULL`.

To read a descriptor:

1. Call `openReadDescriptorProc` with two parameters: the `PIDescriptorHandle` in `descriptor`, and a `NULL`-terminated array of expected key IDs. It will return a `PIReadDescriptor` token, such as *readToken*.

Note: `descriptorKeyIDArray` must be `NULL`-terminated, or the automatic array processor will start to read and write past the array, tromping on your other data and likely crashing the host.

2. Make repeated calls to `GetKeyProc`, which will return information about the current key in the `readToken`. `GetKeyProc` will return `FALSE` when there are no more keys.

3. Make the appropriate call to the `Get` routine responding to the key and type, such as `GetIntegerProc`, `GetFloatProc`, etc.

4. Call `closeReadDescriptorProc` with `readToken`, which will dispose of `readToken` and return any errors that occurred during `GetKeyProc`. (See “Sticky errors”, below.)

5. Dispose of the `PIDescriptorHandle` pointed to in `descriptor` by calling `DisposePIHandleProc` with it. You may keep the descriptor for use, such as a parameter handle, but the `descriptor` field should still be set to `NULL`.

6. Set the descriptor field to `NULL`.

7. Check for and manage any errors (see “Playback error handling” below.)

8. Update your parameters and show your dialog, depending on `playInfo`. See table 8–12.

Table 8–12: playInfo

Name	Description
plugInDialogDontDisplay	Display dialog only if necessary due to missing parameters or error.
plugInDialogDisplay	Present your dialog using the descriptor information.
plugInSilent	Never present a dialog. Use only descriptor information. If the information in insufficient then you should return the error in the `errorString` field. See below.

Playback error handling

Because a descriptor can be built by other software, don’t assume that your keys will come in order, be of the proper type, or all be present.

Coerced parameters

If a `Get` call is made for the wrong type, `paramErr` will be returned unless the type could be coerced, in which case the value will be returned with the `coercedParam` error.

If an error occurs and not `plugInDialogSilent`, then the plug-in may:

1. Present a dialog and return a positive error value, or
2. Return a negative error value and the host will display a standard message.

DescriptorKeyIDArray
During the repeated calls to `GetKeyProc`, `DescriptorKeyIDArray`, passed to `openReadDescriptorProc`, is updated automatically. As each key is found in `GetKeyProc`, the corresponding key in `descriptorKeyIDArray` is set to `typeNull='null'`. Keys still in the array after you're done reading all the data indicate keys that were not passed in the descriptor and you will need to coerce them or request them from the user (if not `plugInDialogSilent`).

Sticky errors
Errors that occur in `Get` routines and GetKeyProc are sticky, meaning an error will keep getting returned until another more drastic error supercedes it. This way you can check for any major errors after you've read all your data.

Table 8–13: Playback errors returned

Name	Description
NULL	No error.
coercedParam	Coerced data to requested type.
paramErr	Error with parameters passed or data does not match requesting proc.
errWrongPlatformFilePath	Mismatch between `typeAlias` (Macintosh) and `typePath` (Windows) request.

Common keys and parameters

Table 8–14: Predefined common keys and parameters

Name	Code	Title/description
keyA	'A '	Alpha channel, A in LAB.
keyAmount	'Amnt'	Amount.
keyAngle	'Angl'	Angle.
keyAs	'As '	Angles, alphas.
keyB	'B '	B in LAB.
keyBlack	'Blck'	Black, K in CMYK.
keyBlue	'Bl '	B in RGB.
keyBook	'Bk '	Book.
keyBrightness	'Brgh'	Brightness, B in HSB.
keyCenter	'Cntr'	Center.
keyColor	'Clr '	Color.
keyCyan	'Cyn '	Cyan, C in CMYK.
keyDatum	'Dt '	Data. Opaque data. Displays as "...".
keyDepth	'Dpth'	Depth, bitdepth.
keyDistance	'Dstn'	Distance.
keyDistribution	'Dstr'	Distribution.
keyDither	'Dthr'	Dithering.
keyEdge	'Edg '	Edge.
keyEncoding	'Encd'	Encoding.
keyFill	'Fl '	Fill.
keyFlatness	'Fltn'	Flatness.
keyFrequency	'Frqn'	Frequency.
keyGray	'Gry '	Gray, grayscale.
keyGreen	'Grn '	Green, G in RGB.
keyHalftoneScreen	'HlfS'	Halftone screen.
keyHeight	'Hght'	Height.
keyHorizontal	'Hrzn'	Horizontal, pixels.
keyHue	'H '	Hue, H in HSB.
keyIn	'In '	In, inData.
keyKind	'Knd '	Kind, type kind.
keyLevel	'Lvl '	Level, level height.
keyLocation	'Lctn'	Location.
keyLuminance	'Lmnc'	Luminance.
keyMagenta	'Mgnt'	Magenta, M in CMYK.
keyMatrix	'Mtrx'	Matrix.
keyMethod	'Mthd'	Method.
keyMode	'Md '	Mode, color mode.
keyMonochromatic	'Mnch'	Monochrome, bitmap, grayscale.
keyName	'Name'	Name, channel name, filename.

Table 8–14: Predefined common keys and parameters (Continued)

Name	Code	Title/description
keyNew	'Nw '	New.
keyOffset	'Ofst'	Offset.
keyPalette	'Plt '	Palette, palette name/number.
keyPosition	'Pstn'	Position.
keyPreview	'Prvw'	Preview.
keyOrientation	'Ornt'	Orientation, landscape, portrait.
keyQuality	'Qlty'	Quality, low, medium, high, max.
keyRadius	'Rds '	Radius.
keyRatio	'Rt '	Ratio.
keyRed	'Rd '	Red, R in RGB.
keyResolution	'Rslt'	Resolution, pixel depth.
keyResponse	'Rspn'	Response.
keySaturation	'Strt'	Saturation, S in HSB.
keyScale	'Scl '	Scale, enlarge/reduce value.
keyShape	'Shp '	Shape.
keyThreshold	'Thsh'	Threshold.
keyTitle	'Ttl '	Title.
keyTo	'To '	To, from-to destination.
keyUsing	'Usng'	Using.
keyValue	'Vl '	Value, generic.
keyVertical	'Vrtc'	Vertical.
keyWidth	'Wdth'	Width.
keyYellow	'Ylw '	Yellow, Y in CMYK.

AppleScript compatibility

The Photoshop scripting system was made with AppleScript compatibility as one of its primary goals. This explains the reliance on some of the more (seemlingly needless) complex structures such as the dictionary resource. The reliance on the dictionary resource, and the structure of the key name and ID pairs, maps directly to AppleScript. There are some important details to watch out for.

AppleScript maintains a global name space, which means if your plug-in is going to be AppleScript compatible, your key name and ID pairs must be completely unique. For example, if you defined:

```
"red",                  // red property
myRed,                  // my unique property ID for Red
typeFloat,              // value type “float”
"",                     // no description
flagsSingleParameter    // single parameter

"green",                // green property
myGreen,                // my unique property ID for Green
typeFloat,              // value type “float”
"",                     // no description
flagsSingleParameter    // single parameter

"blue",                 // blue property
myBlue,                 // my unique property ID for Blue
typeFloat,              // value type “float”
"",                     // no description
flagsSingleParameter    // single parameter
```

You would ruin “red”, “green”, and “blue” for anyone else who attempted to use it, as it would now map to your unique keys (or whoever got their dictionary registered before yours.)

In this case, you must use unique textual names as well, such as:

```
"AdobeSDK red",         // unique red property name
myRed,                  // my unique property ID for Red
typeFloat,              // value type “float”
"",                     // no description
flagsSingleParameter    // single parameter
```

In that case, future requests would take the form of:

```
tell “Adobe SDK dissolve”
    set AdobeSDK red of AdobeSDK Dissolve to 65535, 0, 0
end tell
```

This way is safe and makes sure you don’t conflict with anything else. When in doubt, make the name and ID unique, or use the predefined values. Those are always available and will be mapped to your plug-in through your dictionary resource automatically.

Registration and unique name spaces

When trying to determine unique key name and ID spaces, you must follow these rules:

1. All ID’s starting with an uppercase letter are reserved by Adobe.

2. All ID's that are all uppercase are reserved by Apple.

3. All ID's that are all lowercase are reserved by Apple.

4. This leaves all ID's that begin with a lowercase letter and have at least one uppercase letter for you and other plug-in developers.

Since the scripting system is based on unique IDs and AppleScript, and works the same between Macintosh and Windows, this means that if you wish to register a unique ID you must still use the Apple filename ID registration web page, whether Windows or Macintosh based. The web page is at:

`http://dev.info.apple.com/cftype/main.html`

Common Adobe plug-in ID good will format

For all plug-in developers wishing to allocate a block of IDs (as often is want to do, for sets of plug-ins needing unique variables, etc.) register your plug-in type as a varient, with the first three characters following the basic rules for ID creation, and a last character of "#". This will register all 255 permutations of your ID. Such as:

'`sdK#`' will reserve `sdK`*x*, where *x* is any character, allowing keys such as '`sdK*`', '`sdK0`', '`sdKA`', or '`sdKz`'.

'`gAr#`' will reserve `gAr`*x*, where *x* is any character, allowing keys such as '`gAr$`', '`gAr8`', '`gArG`', or '`gArr`'.

Remember, if you're registering a block of name space, that the first three characters must follow the ID rules: they must start with a lowercase character, and at least one character must be uppercase.

Note: This registration method is not supported by Apple. As a matter of fact, Apple explicitly states that one cannot reserve more than one ID at a time. If we all follow the same rule, however, it will work just fine until another solution becomes apparent.

When you log onto the registration web page to check your unique ID, you must check for '*nam*#' where *nam* is your three-digit ID. If you check and register any four digit ID, without searching for your three-digit ID + "#", then you will probably stomp someone elses name space.

This name space registration method is only useful if we all agree do follow it.

Ignoring AppleScript

If you've decided that forward compatibility with future AppleScript features is not a major concern, you can disable any AppleScript-savvy features and make your plug-in only Photoshop-specific. By doing this, you may ignore any requirements for unique key name and ID pairs. To do this, add a unique ID string to your `HasTerminology` resource. Information on doing this is in the *Plug-in Resource Guide.pdf* in the Photoshop PiPL section under "Scripting-specific properties."

AppleEvents

In Photoshop 4.0 and 5.0, besides the standard AppleEvents, there is one additional AppleEvent call that is supported by the host: *do script*.

You may call into Photoshop with the `do script` command to have it play any currently loaded script in the Actions palette.

```
tell application "Photoshop 4.0"
    do script “MyAction”
end tell
```

A. Data Structures

This appendix provides information about various data structures used by plug–in modules.

Information about the `PiPL` and `PiMI` data structures is contained in the document *Plug-in Resource Guide.pdf*. The different plug–in parameter blocks are described in their respective chapters.

PSPixelMap

```
typedef struct PSPixelMap
{
    int32        version;
    VRect        bounds;
    int32        imageMode;
    int32        rowBytes;
    int32        colBytes;
    int32        planeBytes;
    void         *baseAddr;
    /* Fields new in version 1. */
    PSPixelMask  *mat;
    PSPixelMask  *masks;
    int32        maskPhaseRow;
    int32        maskPhaseCol;
} PSPixelMap;
```

Table A-1: PSPixelMap structure

Type	Field	Description
int32	version	=1. Future versions of Photoshop may support additional parameters and will support higher version numbers for `PSPixelMap`'s.
VRect	bounds	The bounds for the pixel map.
int32	imageMode	The mode for the image data. The supported modes are `grayscale`, `RGB`, `CMYK`, and `Lab`. Additionally, if the mode of the document being processed is `DuotoneMode` or `IndexedMode`, you can pass `plugInModeDuotone` or `plugInModeIndexedColor`.
int32	rowBytes	The offset from one row to the next of pixels.
int32	colBytes	The offset from one column to the next of pixels.
int32	planeBytes	The offset from one plane of pixels to the next. In RGB, the planes are ordered red, green, blue; in CMYK, the planes are ordered cyan, magenta, yellow, black; in Lab, the planes are ordered L, a, b.
void *	baseAddr	The address of the byte value for the first plane of the top left pixel.
PSPixelMask *	mat	For all modes except indexed color, you can specify a mask to be used for matting correction. For example, if you have white matted data to display, you can specify a mask in this field which will be used to remove the white fringe. This field points to a `PSPixelMask` structure (see below) with a `maskDescription` indicating what type of matting needs to be compensated for. If this field is `NULL`, Photoshop performs no matting compensation. If the masks are chained, only the first mask in the chain is used.
PSPixelMask *	masks	This points to a chain of `PSPixelMasks` which are multiplied together (with the possibility of inversion) to establish which areas of the image are transparent and should have the checkerboard displayed. `kSimplePSMask`, `kBlackMatPSMask`, `kWhiteMatPSMask`, and `kGrayMatPSMask` all operate such that 255=opaque and 0=transparent. `kInvertPSMask` has 255=transparent and 0=opaque.
int32	maskPhaseRow	The phase of the checkerboard with respect to the top left corner of the `PSPixelMap`.
int32	maskPhaseCol	

PSPixelMask

```
typedef struct PSPixelMask
{
    struct PSPixelMask   * next
    void                 * maskData;
    int32                rowBytes;
    int32                colBytes;
    int32                maskDescription;
} PSPixelMask;
```

Table A-2: PSPixelMask structure

Type	Field	Description
`PSPixelMask *`	next	A pointer to the next mask in the chain
void *	maskData	A pointer to the mask data.
int32	rowBytes	The row step for the mask.
int32	colBytes	The column step for the mask.

int32	maskDescription	The mask description value, which is one of the following:
		
		`0=kSimplePSMask`
		`1=kBlackMatPSMask`
		`2=kGrayMatPSMask`
		`3=kWhiteMatPSMask`
		`4=kInvertPSMask`

ColorServicesInfo

This data structure is used in the `ColorServices` callback function. See chapter 3 and the notes following table A-3 for more details.

```
typedef struct ColorServicesInfo
    {
    int32 infoSize;
    int16 selector;
    int16 sourceSpace;
    int16 resultSpace;
    Boolean resultGamutInfoValid;
    Boolean resultInGamut;
    void *reservedSourceSpaceInfo;
    void *reservedResultSpaceInfo;
    int16 colorComponents[4];
    void *reserved;
    union
        {
        Str255 *pickerPrompt;
        Point *globalSamplePoint;
        int32 specialColorID;
        } selectorParameter;
    }
ColorServicesInfo;
```

Table A-3: ColorServicesInfo structure

Type	Field	Description

int32	infoSize	Size of the `ColorServicesInfo` record in bytes. The value is used as a version identifier in case this record is expanded in the future. It can be filled in like so:
		
		`ColorServicesInfo requestInfo;`
		`requestInfo.infoSize = sizeof(requestInfo);`

int16	selector	Operation performed by the `ColorServices` callback.
		
		`0=plugIncolorServicesChooseColor`
		`1=plugIncolorServicesConvertColor`
		`2=plugIncolorServicesSamplePoint`
		`3=plugIncolorServicesGetSpecialColor`

int16	sourceSpace	Indicates the color space of the input color contained in `colorComponents`.
		
		For `plugIncolorServicesChooseColor` the input color is used as an initial value for the picker.
		
		For `plugIncolorServicesConvertColor` the input color will be converted from the color space indicated by `sourceSpace` to the one indicated by `resultSpace`.
		
		`0=plugIncolorServicesRGBSpace`
		`1=plugIncolorServicesHSBSpace`
		`2=plugIncolorServicesCMYKSpace`
		`3=plugIncolorServicesLabSpace`
		`4=plugIncolorServicesGraySpace`
		`5=plugIncolorServicesHSLSpace`
		`6=plugIncolorServicesXYZSpace`

Table A-3: ColorServicesInfo structure (Continued)

Type	Field	Description
int16	resultSpace	Desired color space of the result color. The result will be contained in the `colorComponents` field. For the `plugIncolorServicesChooseColor` selector, `resultSpace` can be set to `-1=plugIncolorServicesChosenSpace` to return the color in whichever color space the user chose. In that case, `resultSpace` will contain the chosen color space on output.
Boolean	resultGamutInfoValid	This output only field indicates whether the `resultInGamut` field has been set. In Photoshop 3.0 and later, this will only be true for colors returned in the `plugIncolorServicesCMYKSpace` color space.
Boolean	resultInGamut	Boolean. Indicates whether the returned color is in gamut for the currently selected printing setup. Only meaningful if `resultGamutInfoValid=TRUE`.
void *	reservedSourceSpaceInfo	`=NULL`, otherwise returns parameter error .
void *	reservedResultSpaceInfo	`=NULL`, otherwise returns parameter error .
int16	colorComponents[4]	Actual color components of the input or output color. See table A-4.
void *	reserved	`=NULL`, otherwise returns parameter error .
union	selectorParameter	This union is used for providing different information based on the selector field: `pickerPrompt`, `globalSamplePoint`, or `specialColorID`. The `pickerPrompt` variant contains a pointer to a Pascal string which will be used as a prompt in the color picker for the `plugIncolorServicesChooseColor` call. `NULL` can be passed to indicate no prompt. `globalSamplePoint` points to a `Point` record that is the current sample point. `specialColorID` should be either: `0=plugIncolorServicesForegroundColor` or `1=plugIncolorServicesBackgroundColor`.

Table A-4: 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

Table A-4: colorComponents array structure (Continued)

Color space	color Components[0]	color Components[1]	color Components[2]	color Components[3]
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
Gray scale	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

PlugInMonitor

A number of the plug–in module types get passed monitor descriptions via the `PlugInMonitor` structure. These descriptions basically detail the information recorded in Photoshop's "Monitor Setup" dialog and are passed in a structure of the following type:

```
typedef struct PlugInMonitor
{
      Fixed gamma;
      Fixed redX;
      Fixed redY;
      Fixed greenX;
      Fixed greenY;
      Fixed blueX;
      Fixed blueY;
      Fixed whiteX;
      Fixed whiteY;
      Fixed ambient;
} PlugInMonitor;
```

Table A-5: PlugInMonitor structure

Type	Field	Description
Fixed	gamma	This field contains the monitor's gamma value or zero if the whole record is invalid.
Fixed	redX	These fields specify the chromaticity coordinates of the monitor's phosphors.
Fixed	redY	
Fixed	greenX	
Fixed	greenY	
Fixed	blueX	
Fixed	blueY	
Fixed	whiteX	These fields specify the chromaticity coordinates of the monitor's white point.
Fixed	whiteY	
Fixed	ambient	This field specifies the relative amount of ambient light in the room. Zero means a relatively dark room, 0.5 means an average room, and 1.0 means a bright room.

ResolutionInfo

This structure contains information about the resolution of an image. It is written as an image resource. See the *Document file formats* chapter for more details.

```
struct ResolutionInfo
{
    Fixed               hRes;
    int16               hResUnit;
    int16               widthUnit;
    Fixed               vRes;
    int16               vResUnit;
    int16               heightUnit;
};
```

Table A-6: ResolutionInfo structure

Type	Field	Description
Fixed	hRes	Horizontal resolution in pixels per inch.
int16	hResUnit	1=display horitzontal resolution in pixels per inch; 2=display horitzontal resolution in pixels per cm.
int16	widthUnit	Display width as 1=inches; 2=cm; 3=points; 4=picas; 5=columns.
Fixed	vRes	Vertial resolution in pixels per inch.
int16	vResUnit	1=display vertical resolution in pixels per inch; 2=display vertical resolution in pixels per cm.
int16	heightUnit	Display height as 1=inches; 2=cm; 3=points; 4=picas; 5=columns.

DisplayInfo

This structure contains display information about each channel. It is written as an image resource. See the *Document file formats* chapter for more details.

```
struct DisplayInfo
    {
    int16               colorSpace;
    int16               color[4];
    int16               opacity;    // 0..100
    char                kind;       // selected = 0, protected = 1
    char                padding;    // should be zero
    };
```

Table A-7: DisplayInfo Color spaces

Color ID	Name	Description
0	RGB	The first three values in the color data are *red*, *green*, and *blue*. They are full unsigned 16–bit values as in Apple's `RGBColor` data structure. Pure red=65535,0,0.
1	HSB	The first three values in the color data are *hue*, *saturation*, and *brightness*. They are full unsigned 16–bit values as in Apple's `HSVColor` data structure. Pure red=0,65535, 65535.
2	CMYK	The four values in the color data are *cyan*, *magenta*, *yellow*, and *black*. They are full unsigned 16–bit values. 0=100% ink. Pure cyan=0,65535,65535,65535.
7	Lab	The first three values in the color data are *lightness*, *a chrominance*, and *b chrominance*. Lightness is a 16–bit value from 0...10000. The chromanance components are each 16–bit values from –12800...12700. Gray values are represented by chrominance components of 0. Pure white=10000,0,0.
8	grayscale	The first value in the color data is the gray value, from 0...10000.

The table is identical to the Colors load file format. See the *Photoshop File Formats.pdf*.

Symbols

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z