Adobe

Adobe Photoshop® 4.0

Software Development Kit

Version 4.0 Release 2
November 1996

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.

Version History

Date	Author	Status
7 November 1994	David J. Wise	First draft
15 January 1995	David J. Wise	First release
8 February 1995	Seetharaman Narayanan	MS-Windows modifications
16 July 1995	Paul D. Ferguson	Reformatted and updated for Photoshop 3.0.4
6 February 1996	Andrew Coven	Updated for Photoshop 3.0.5, Cross-application development
20 November 1996	Andrew Coven	Information, modules and callbacks updated for Photoshop 4.0.

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 non-module specific details of Scripting.

Chapters 11 and 12 cover all the different load, save, and document file formats built into Photoshop.

The appendicies contain specific information about different parameter structures and scripting with OLE Automation.

A new document has been added, *Plug-in Resources Guide.pdf*, with valuable tips and tricks for creating Photoshop Plug-ins that function in every major Adobe graphics application.

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.

GAP SDK tech notes mailing list

The Adobe Developers Association maintains an area on Adobe's world-wide-web site: `http://www.adobe.com`, which includes the latest SDK public releases and technical notes. You can also have the technical notes e–mailed to you directly by joining the Graphics and Publishing SDK tech notes mailing list. The GAP SDK Tech Notes e-mail list is for Adobe After Effects, Adobe Illustrator, Adobe PageMaker, Adobe Photoshop, Adobe PhotoDeluxe, and Adobe Premiere. Send an e-mail to

```
gap-dse@adobe.com
```

with the subject:

```
SUBSCRIBE GAP-SDK-TECH-NOTES
```

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.
10. CC:
    Any other e-mail addresses you want CC:'ed.

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 4.0 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`.

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 eight types of plug–in modules:

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 *4. Color Picker Modules*, on page 58.

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 *5. Import Modules*, on page 63.

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 *6. Export Modules*, on page 74.

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 *7. Filter Modules*, on page 81.

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 *8. Format Modules*, on page 93.

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 *9. Selection Modules*, on page 111.

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

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`
Color Picker	`8BCM`	`.8BC`
Import	`8BAM`	`.8BA`
Export	`8BEM`	`.8BE`
Extension	`8BXM`	`.8BX`
Filter	`8BFM`	`.8BF`
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, unsigned8, unsigned16, unsigned32	These are 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 maxSpace 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.

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

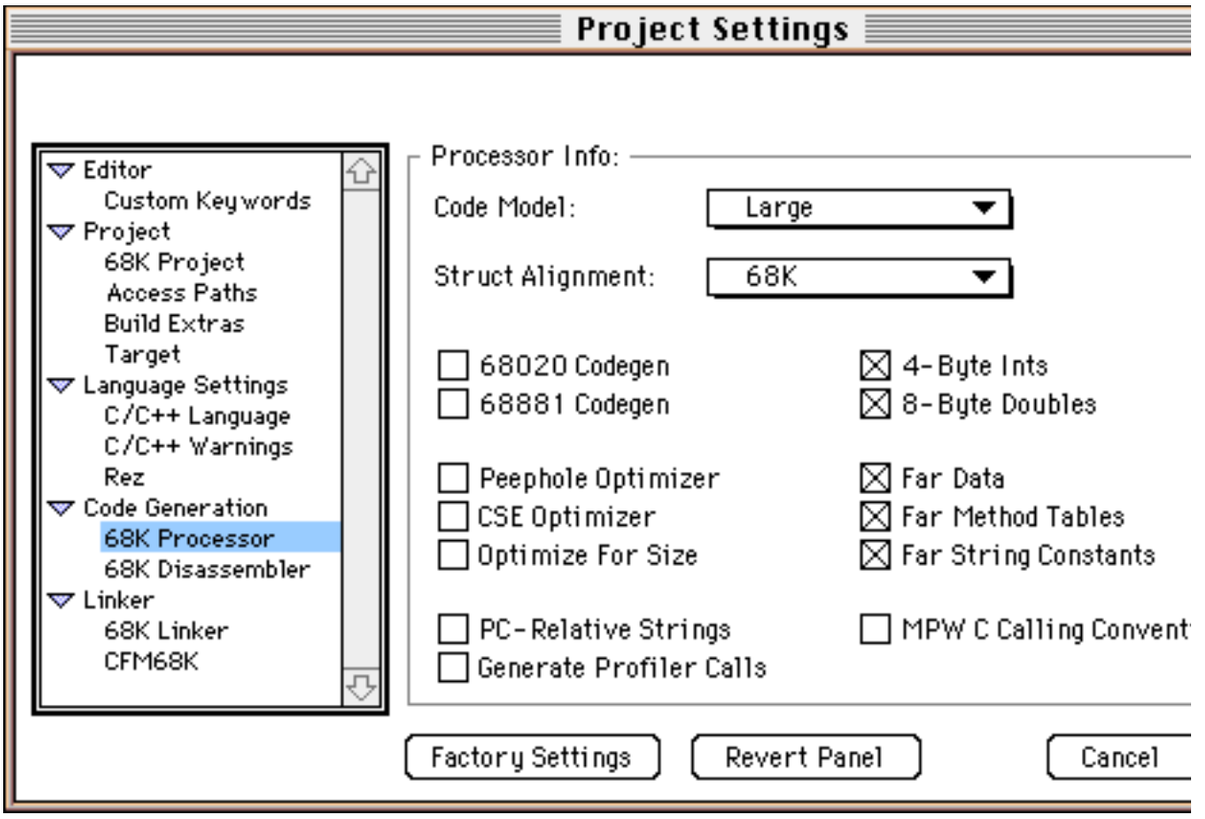
Macintosh 680x0 applications have a special code segment called the jump table. When a routine in one segment calls a routine in another segment, it actually calls a small glue routine in the jump table segment. This glue routine loads the routine's segment into memory if needed, and jumps to its actual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

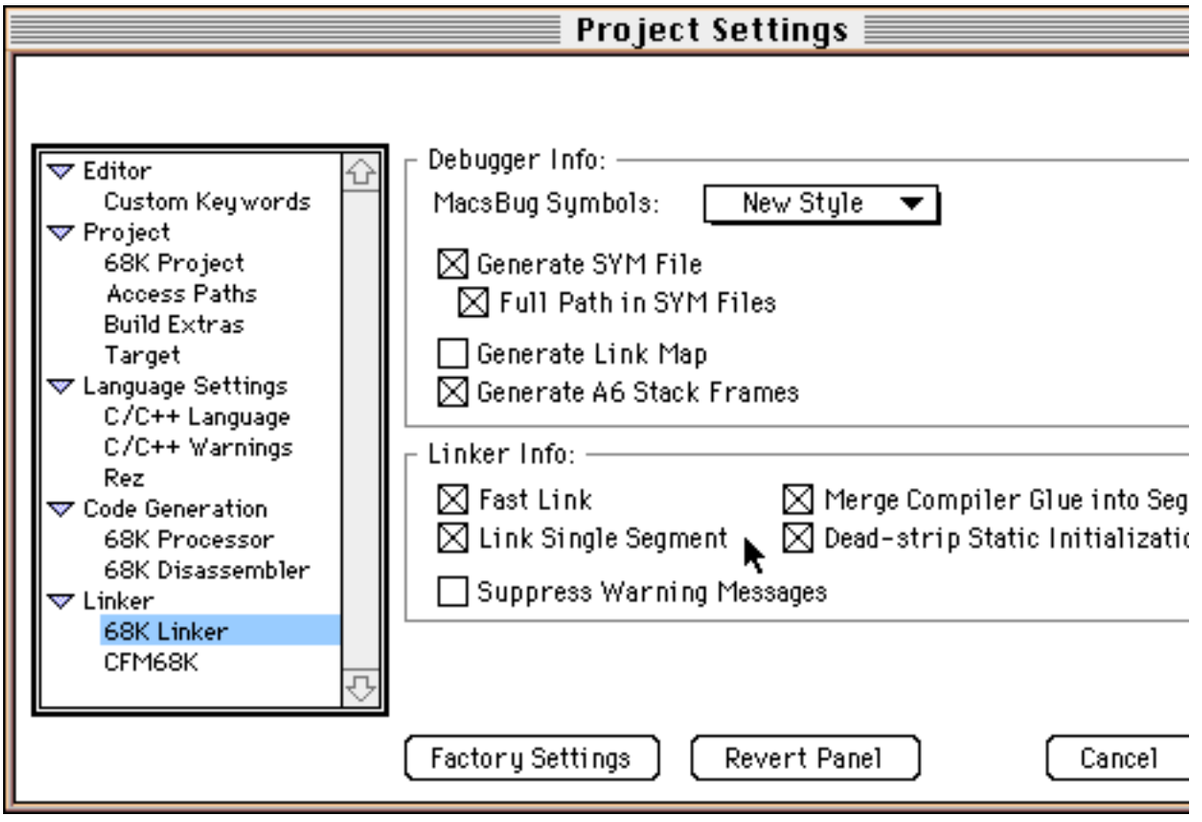
The simplest way to solve this is to link all the plug–in's code into a single segment. This usually requires setting optional compilation/link flags in your development environment if the resultant segment exceeds 32k.

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.

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.

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 uses 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

Building your plug-ins with CodeWarrior 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++ 4.1 and Windows 95.

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.

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:

1. Do a registry search for the Photoshp.exe key.
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.

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

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 by Adobe Photoshop 4.0 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: 4.0; 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: 4.0; 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 *9. Selection Modules*, on page 32.

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: 1; Adobe Photoshop 4.0; 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 *10. Scripting Plug-ins*, on page 118.

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: 1; Adobe Photoshop: 4.0; 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: 1; Adobe Photoshop 4.0; 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: 4.0; 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: 4.0; 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 4.0

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,
                                    PIImagePlane *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: 4.0; 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`).

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( )

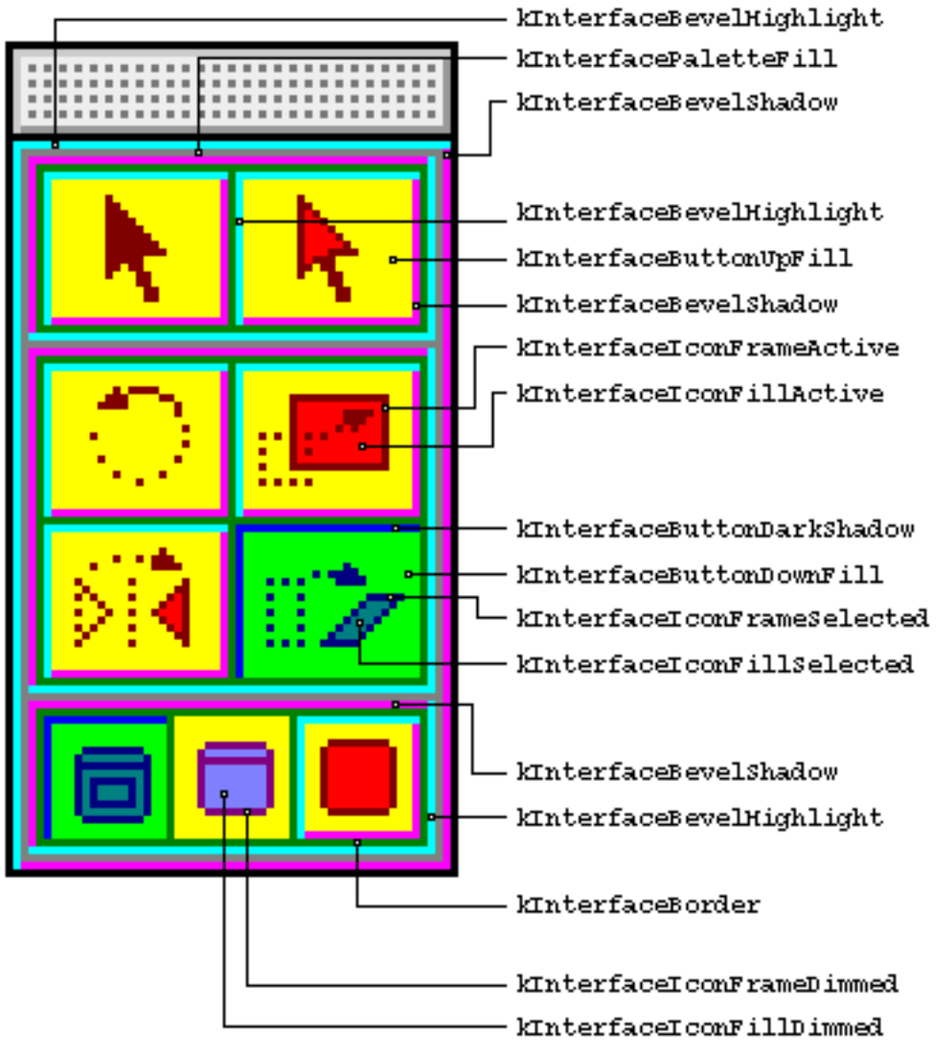
```
MACPASCAL OSErr (*SetPropertyProc) (OSType signature, OSType key, int32 index,
int32 simpleProperty, Handle complexProperty);
```

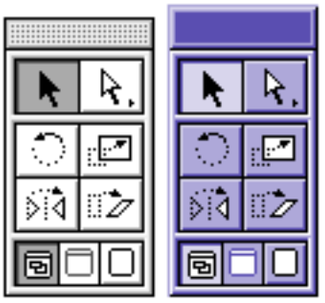
This routine allows you to update information in the plug–in host about the document currently being processed.

propInterfaceColor

Adobe Photoshop 4.0 includes 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
kInterfacePaletteFill	5
kInterfaceIconFrameDimmed	6
kInterfaceIconFrameActive	7
kInterfaceBevelHighlight	8
kInterfaceButtonDownFill	9

Table 3–8: propInterfaceColor index (Continued)

Name	Value
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 *11. Document File Formats.*
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.
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.

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

Property Name	ID	Type	Description
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	simple *mod*	Whether the current image is considered copy-written.
propURL	URL	complex *mod*	The URL for the current image.
propTitle	titl	complex	The title of the current image.

Pseudo–Resource suite

Current version: 3; Adobe Photoshop: 4.0; 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. 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.

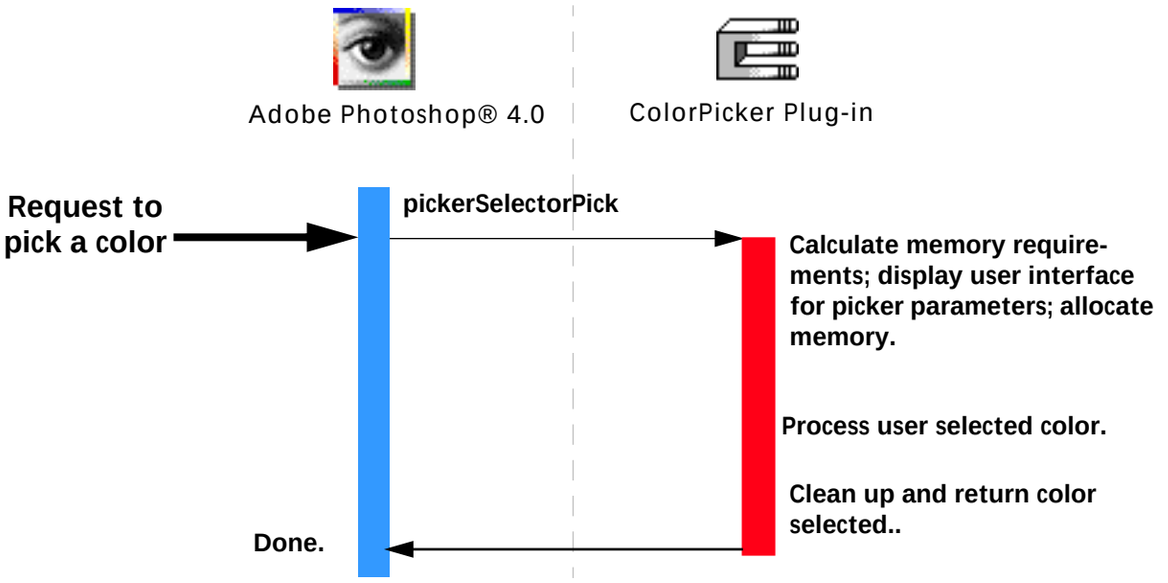
Table 4–1: Color Picker file types

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

Examples/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.
ChannelPortProcs *	channelPortProcs	Channel Ports callback suite. See chapter 3.
PropertyProcs *	propertyProcs	Property 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.
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.
PickParms	pickParms	Picker incoming and outgoing parameters. See table 4–2.
char[260]	reservedBlock	Reserved for future use. Set to zero.

5. Import Modules

Import plug–in modules are used to capture images from add–on hardware, such as scanners or video cameras, and put these images into new Photoshop document windows.

Import modules can also be used to read images from unsupported file formats, although file format modules often are better suited for this purpose. File format modules are accessed directly from the **Open...** command, while Import modules use the **Import** sub–menu.

Prior to Photoshop version 4.0 these modules were called Acquire modules. Most of the internal nomenclature and function calls reflect the old naming conventions, to stay compatible with previous versions.

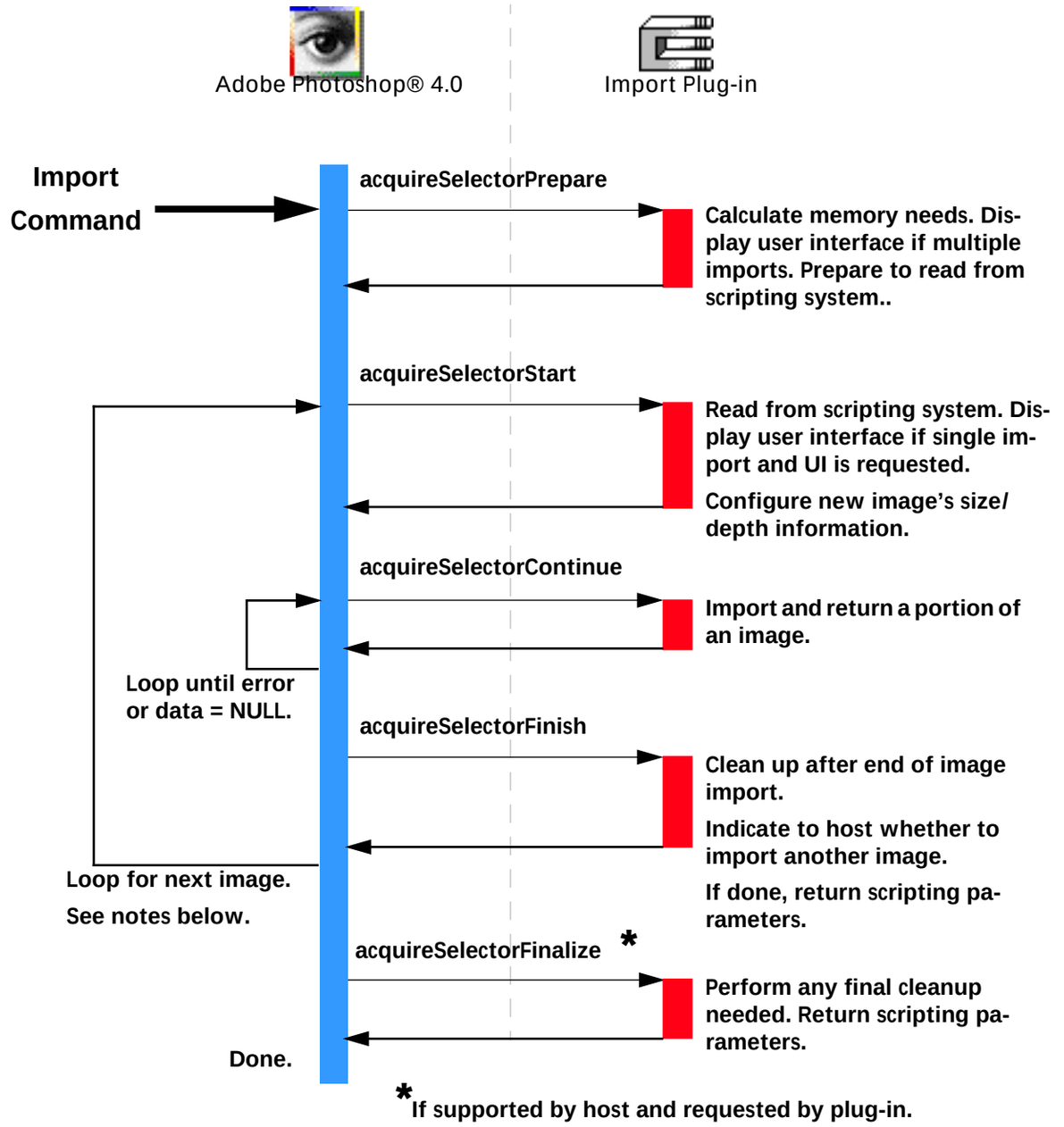
Table 5–1: Import file types

OS	Filetype/extension
Mac OS	8BAM
Windows	.8BA

Examples/Import/GradientImport

GradientImport is a sample import module. This version of GradientImport requires at least Photoshop 3.0, since it uses the `advanceState` callback and the improved multiple import design. It is also updated for scripting and has a detailed example of exporting scripting information for multiple import structures.

Calling sequence



The calling sequence for Import modules is a little more complex than other types of plug–in modules. In a single invocation, Import modules may be capable of capturing multiple images and creating multiple new Photoshop document windows. Because captured images may be large, each capture may require multiple exchanges between the host and the module.

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

acquireSelectorPrepare

The *acquireSelectorPrepare* calls allow your plug–in to adjust Photoshop's memory allocation algorithm. Photoshop sets `maxData` to the maximum number of bytes it can allocate to your plug–in. For Import modules to perform efficiently, you should reduce `maxData` to permit Photoshop to process the imported image in RAM. 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.

acquireSelectorStart

This call lets you indicate to the plug–in host the mode, size and resolution of the image being returned, so it can allocate and initialize its data structures. Here you can update your parameters based on the passed scripting parameters, and show your user interface, if requested.

During this call, your plug–in module should set `imageMode`, `imageSize`, `depth`, `planes`, `imageHRes` and `imageVRes`. If an indexed color image is being returned, you should also set `redLUT`, `greenLUT` and `blueLUT`. If a duotone mode image is being returned, you should also set `duotoneInfo`. See the descriptions of these fields later in this chapter.

acquireSelectorContinue

This call returns an area of the image to the plug–in host. Photoshop will continue to call this routine until it either returns an error, or your plug–in module sets the `data` field to `NULL`.

theRect, loPlane & hiPlane

The area of the image being returned is specified by *theRect*, *loPlane* and *hiPlane*.

The portion of the image being returned is specified by `theRect`. If the resolution of the imported image is always going to be very small (for example, NTSC frame grabbers), your plug–in can simply set `theRect` to the entire image area. However, if you are working with large images, your plug–in must use the `theRect` field to return the image in several pieces.

There are no restrictions on how the pieces tile the image; horizontal and vertical strips are allowed as are a grid of tiles. Each piece should contain no more than `maxData` bytes, less the size of any large tables or scratch areas allocated by the plug–in. These restrictions don't apply if the buffer for the image data was allocated using the Buffer or Handle suites.

data, colBytes, rowBytes & planeBytes

The *data* field should point to the actual data being returned. The fields *colBytes*, *rowBytes* and *planeBytes* specify the organization of the data.

The data field contains a pointer to the data being returned. You can allocate a buffer for the data via:

1. the Mac OS NewPtr trap,
2. the Windows GlobalAlloc function, or
3. via the Buffer suite.

Your plug–in module is responsible for freeing this buffer in its acquireSelectorFinish handler.

Photoshop is very flexible in the format in which image data can be returned. For example, to return just the red plane of an RGB color image, use the parameter values in Table 5–2.

Table 5–2: Return red plane of RGB

Parameter	Value
loPlane	0
hiPlane	0
colBytes	1
rowBytes	width of the area being returned
planeBytes	ignored, since loPlane=hiPlane.

If you wish to return the RGB data in interleaved form (RGBRGB...), use the values shown in Table 5–3.

Table 5–3: Return RGB data in interleaved form

Parameter	Value
loPlane	0
hiPlane	2
colBytes	3
rowBytes	3 * width of the area being returned
planeBytes	1

acquireSelectorFinish

This call allows your plug–in to clean up after an image import. This call is made if and only if the acquireSelectorStart routine returns without error, even if the acquireSelectorContinue routine returns an error.

Most plug–ins will at least need to free the buffer used to return the image data.

If Photoshop detects *Command–period* in the Mac OS or *Escape* in Windows while processing the results of an acquireSelectorContinue call, it will call acquireSelectorFinish.

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

Scripting at acquireSelectorFinish

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.

acquireSelectorFinalize

If your plug–in is using finalization—the host set `canFinalize` and your plug–in set `wantFinalize`—then this call will be made after all possible looping is complete. This can be used to do any final clean–up, and is typically used in the case where a plug–in module is acquiring multiple images during a single invocation.

Scripting at acquireSelectorFinalize

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 `acquireSelectorPrepare` succeeds—the result value is zero—and `wantFinalize=TRUE`, then Photoshop guarantees that `acquireSelectorFinalize` will be called.

If `acquireSelectorStart` succeeds then Photoshop guarantees that `acquireSelectorFinish` will be called.

In the event of any error during import, the document being imported is discarded.

Plug–in hosts may choose to treat `acquireAgain` as `FALSE`.

Your plug–in module can tell whether the host understands finalization by checking the `canFinalize` flag.

The `advanceState` callback allows your plug–in module to drive the interaction through the inner `acquireSelectorContinue` loop without actually returning to the plug–in host. If the host returns an error, then you should treat this as an error condition and return the error code when returning from your `acquireSelectorContinue` handler.

Multiple Acquire

The plug-in host can loop back to `acquireSelectorStart` to begin importing another image for multi-image importing if the following conditions are true:

1. the plug–in host supports multiple imports (Photoshop version 3.0 and later)

2. your plug–in module has set `acquireAgain=TRUE`, and

3. the `acquireSelectorContinue` loop finished normally, meaning no error was returned and the loop ended with `data=NULL`.

The plug-in host can also loop back to acquireSelectorstart to begin acquiring another image if these *alternate* conditions are true:

1. the plug–in host supports multiple imports,
2. the plug–in host set canFinalize=TRUE,
3. your plug–in module set wantFinalize=TRUE and acquireAgain=TRUE, and
4. the acquireSelectorContinue loop finished with a result code >= 0 or a result code of userCanceledErr.

Batch Import

Batch Importing is a feature of the scripting system that automatically processes multiple files through your scripting-aware Import module. If your Import module is scriptable, Batch Importing is handled completely by the host, which passes parameters and control to your Import plug-in as part of a script. Batch Import is available from the Actions palette.

Batch Import versus Multiple Import

While Multiple Import is an internal feature available in the Import module, Batch Import is based on the host scripting mechanism. Here are some issues that should help you determine whether to implement Multiple Import or Batch Import:

1. Batch Import is transparent if your plug-in is scripting-aware. This means you need only export your parameters to the scripting system and read them in at call-time to be able to be controlled by Batch Import. Batch Import of single imports would be the most appropriate for single-sheet scanners, for instance. Batch Import triggers when your user interface is closed. This means if you do a multiple acquire and leave your user interface up, the scripting system will not take control until after your multiple acquire is done.
2. Multiple Acquire is always controlled by the plug-in. It allows you to return multiple images with one invocation, versus Batch Import which calls your plug-in new for every image. The scripting system will only take control after your user interface is closed. You may hide your user interface during your Multiple Acquire loop, between the acquireSelectorFinish and next acquireSelectorStart call, for the scripting system to process that image and return to your control.

If you decide to implement Multiple Import but still want your plug-in to be scripting aware, then we recommend you follow the GradientImport example from the SDK, and export your Multiple Import commands as a single scripting event. Batch Import of a module that does Multiple Import would be the most appropriate for digital cameras, for instance, where the user wants to grab every other image in the cameras' buffer.

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 PIAcquire.h.

```
#define acquireBadParameters   -30000    // an error with the interface
#define acquireNoScanner       –30001    // no scanner installed
#define acquireScannerProblem  –30002    // a problem with the scanner
```

The Import parameter block

The pluginParameterBlock parameter passed to your plug–in module's entry point contains a pointer to an AcquireRecord structure with the following fields. This structure is declared in PIAcquire.h.

Table 5–4: AcquireRecord fields

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. You should only call this during the actual main operation of the plug–in, not during long operations during the preliminary user interface. For example, it should not be used during a preview operation that computes a low resolution proxy image. It should be used during the main, high–resolution scan.
int32	maxData	Photoshop initializes this field to the maximum number of bytes it can free up. Your plug–in may reduce this value during the acquireSelectorPrepare routine. The acquireSelectorContinue routine should return the image in strips no larger than maxData, less the size of any large tables or scratch areas it has allocated unless it uses the Buffer or Handle suites to allocate the memory.
int16	imageMode	Your acquireSelectorStart handler should set this field to inform the plug–in host what mode image is being imported (grayscale, RGB Color, etc.). See PIGeneral.h for valid image mode constants.
Point	imageSize	Your acquireSelectorStart handler should set this field to inform the plug–in host of the image's width, imageSize.h, and height, imageSize.v in pixels.
int16	depth	Your acquireSelectorStart handler should set this field to inform the plug–in host of the image's resolution in bits per pixel per plane. The only valid values are 1 for bitmap mode images, 8 for all other modes; grayscale and RGB also allow 16.
int16	planes	Your acquireSelectorStart handler should set this field to inform the plug–in host of the number of channels in the image. For example, if an RGB image without alpha channels is being returned, this field should be set to 3. Even though Import modules can create documents with up to 24 channels, because of the implementation of the plane map, Import and Format modules should never try to work with more than 16 planes at a time.

Table 5–4: AcquireRecord fields (Continued)

Type	Field	Description
Fixed	imageHRes	Your `acquireSelectorStart` handler should set these fields to inform the plug–in host of the image's horizontal and vertical resolution in pixels per inch. This is a fixed point, 16 binary digit-number. Photoshop initializes these fields to 72 pixels per inch. The current version of Photoshop only supports square pixels, so it ignores the `imageVRes` field. Plug–ins should set both fields anyway in case future versions of Photoshop support non–square pixels.
Fixed	imageVRes	
LookUpTable	redLUT	If an indexed color mode image is being returned, your `acquireSelectorStart` handler should return the image's color table in these fields.
LookUpTable	greenLUT	
LookUpTable	blueLUT	
void *	data	Your `acquireSelectorContinue` handler should return a pointer to the image's data in this field. After all of the image has been returned, set this pointer to `NULL`. Note that your plug–in is responsible for freeing any memory pointed to by this field. This is a change from previous versions of Photoshop's Import plug–in interface.
Rect	theRect	Your `acquireSelectorContinue` handler should set this field to the area being returned.
int16	loPlane	Your `acquireSelectorContinue` handler should set these fields to the first and last planes being returned. For example, if interleaved RGB data is being returned, they should be set to 0 and 2, respectively.
int16	hiPlane	
int16	colBytes	Your `acquireSelectorContinue` handler should set this field to the offset in bytes between columns of returned data. This is usually 1 for non–interleaved data, or `hiPlane–loPlane+1` for interleaved data.
int32	rowBytes	Your `acquireSelectorContinue` handler should set this field to the offset in bytes between rows of returned data.
int32	planeBytes	Your `acquireSelectorContinue` handler should set this field to the offset in bytes between planes of returned data. This field is ignored if `loPlane=hiPlane`. It should be set to 1 for interleaved data.
Str255	fileName	By default, Photoshop opens newly imported images as "Untitled–..." . For file-importing Import modules, set this field to the filename in the `acquireSelectorStart` routine. Photoshop will display the correct window title. Scanning modules should ignore this field.
int16	vRefNum	If your plug–in module sets `fileName`, and you're in the Mac OS, set `vRefNum` to the file's volume reference number. This is ignored in Windows.
Boolean	dirty	By default, newly imported images are marked as *dirty*, meaning that the user will be prompted to save the image when closing the window. Set this field to FALSE to prevent this. This does *not* reflect whether there are unsaved changes in the current document.

Table 5–4: AcquireRecord fields (Continued)

Type	Field	Description
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.
int32	hostModes	This field is used by the host to inform your plug–in module which imageMode values it supports. If the corresponding bit is 1 (LSB = bit 0), the mode is supported. This field can be used by plug–ins to disable features such as color scanning if not supported by the host.
PlaneMap	planeMap	This is initialized by the plug–in host to a linear map, planeMap[i]=i. This is used to map plane (channel) numbers between the plug–in and the host. For example, Photoshop stores RGB images with an alpha channel in the order RGBA, whereas most frame buffers store the data in ARGB order. To return the data in this order, set planeMap[0]=3, planeMap[1]=0, planeMap[2]=1, and planeMap[3]=2. Attempts to index past the end of a planeMap will result in the identity map being used for the indexing.
Boolean	canTranspose	If the host supports transposing images during or after scanning, it sets this field to TRUE. Photoshop always sets this field to TRUE.
Boolean	needTranspose	This field is initialized by the host to FALSE. If your plug–in wishes to have the image transposed, and canTranspose=TRUE, you should set this field to TRUE in your acquireSelectorStart handler. The logical effect is to transpose the image after scanning is complete, although some hosts may find it more efficient to transpose the data during scanning. This feature was added to the plug–in specification because versions of Photoshop prior to Photoshop 2.5 had a strong bias toward horizontal strips. Using this routine, a plug–in could import an image in vertical strips by passing Photoshop horizontal strips and then having Photoshop transpose the data when it was done.
Handle	duotoneInfo	If your plug–in module is acquiring a duotone mode image, you should allocate a handle and return the duotone information here. The format of the information is the same as that provided by export modules. Your plug–in is responsible for freeing the handle in its acquireSelectorFinish handler.
int32	diskSpace	This field contains the number of free bytes on the plug–in host's scratch disk or disks. If the plug–in host does not use a scratch disk, this will be –1.
SpaceProc	spaceProc	If not NULL, this field contains a pointer to the SpaceProc callback. See chapter 3.

Table 5–4: AcquireRecord fields (Continued)

Type	Field	Description
PlugInMonitor	monitor	This field contains the monitor setup information for the host. Refer to Appendix A.
void *	platformData	This field contains a pointer to platform specific data. Not used under the Mac OS.
BufferProcs *	bufferProcs	This field contains a pointer to the Buffer suite if it is supported by the plug–in host, otherwise NULL. See chapter 3.
ResourceProcs *	resourceProcs	This field contains a pointer to the Pseudo–Resource suite if it is supported by the plug–in host, otherwise NULL. See chapter 3.
ProcessEventProc	processEvent	This field contains a pointer to the ProcessEvent callback documented in chapter 3. It contains NULL if the callback is not supported. This function is not useful on Windows.
Boolean	canReadBack	If the plug–in host supports reading back image data for further processing, it should set this field to TRUE. Photoshop always sets this field to TRUE.
Boolean	wantReadBack	If your plug–in module sets this flag and the host supports image read back, then the host will ignore the contents of the buffer it is passed and will instead fill the buffer with the image data. It will store the data in the format described by loPlane, hiPlane, colBytes, rowBytes, planeBytes, and planeMap. If theRect exceeds the bounds of the image, those portions of the buffer will be left untouched.
Boolean	acquireAgain	If you want your plug–in to be called again to import another image, set this flag in the acquireSelectorFinish handler. Plug–in hosts that support multiple image imports should start the import process again with a call to acquireSelectorStart. If you do not want to put up a user interface for each import, you should display your interface during the acquireSelectorPrepare call. With the addition acquireSelectorFinalize, import plug–in modules can now put up an interface that remains active across multiple imports. Your plug–in module should not count on being called again just because it sets this flag; acquireSelectorFinish should still do all of the necessary clean–up.
Boolean	canFinalize	If the host can make the finalize call, it should set this field to TRUE.
DisplayPixelsProc	displayPixels	This field contains a pointer to the DisplayPixels callback. It contains NULL if the callback is not supported. See chapter 3.
HandleProcs *	handleProcs	This field contains a pointer to the Handle 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	wantFinalize	This flag requests an acquireSelectorFinalize call if the host provides the newer protocol. See also canFinalize.

Table 5–4: AcquireRecord fields (Continued)

Type	Field	Description
char[3]	reserved	This 3 byte field is used for alignment to a four–byte boundary.
ColorServicesProc	colorServices	This field contains a pointer to the `ColorServices` callback. It contains `NULL` if the callback is not supported.
AdvanceStateProc	advanceState	The `advanceState` callback allows your plug–in module to drive the interaction through the inner `acquireSelectorContinue` loop without actually returning to the plug–in host. If the `advanceState` call returns an error, you should treat this as a continue error and return the error code back to the plug–in host.
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.
int16	tileWidth	The host reports the width and height of a tile, which would be the best unit to work in, if possible.
int16	tileHeight	
Point	tileOrigin	The origin of the tiling system.
PropertyProcs *	propertyProcs	A pointer to the Property callback suite. See chapter 3.
These fields are new since version 4.0 of Adobe Photoshop.		
PIDescriptorParameters *	descriptorParameters	Descriptor callback 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*” .
char[192]	reserved	Reserved for future use. Set to zero.

6. 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 6–1: Export file types

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

Examples/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.

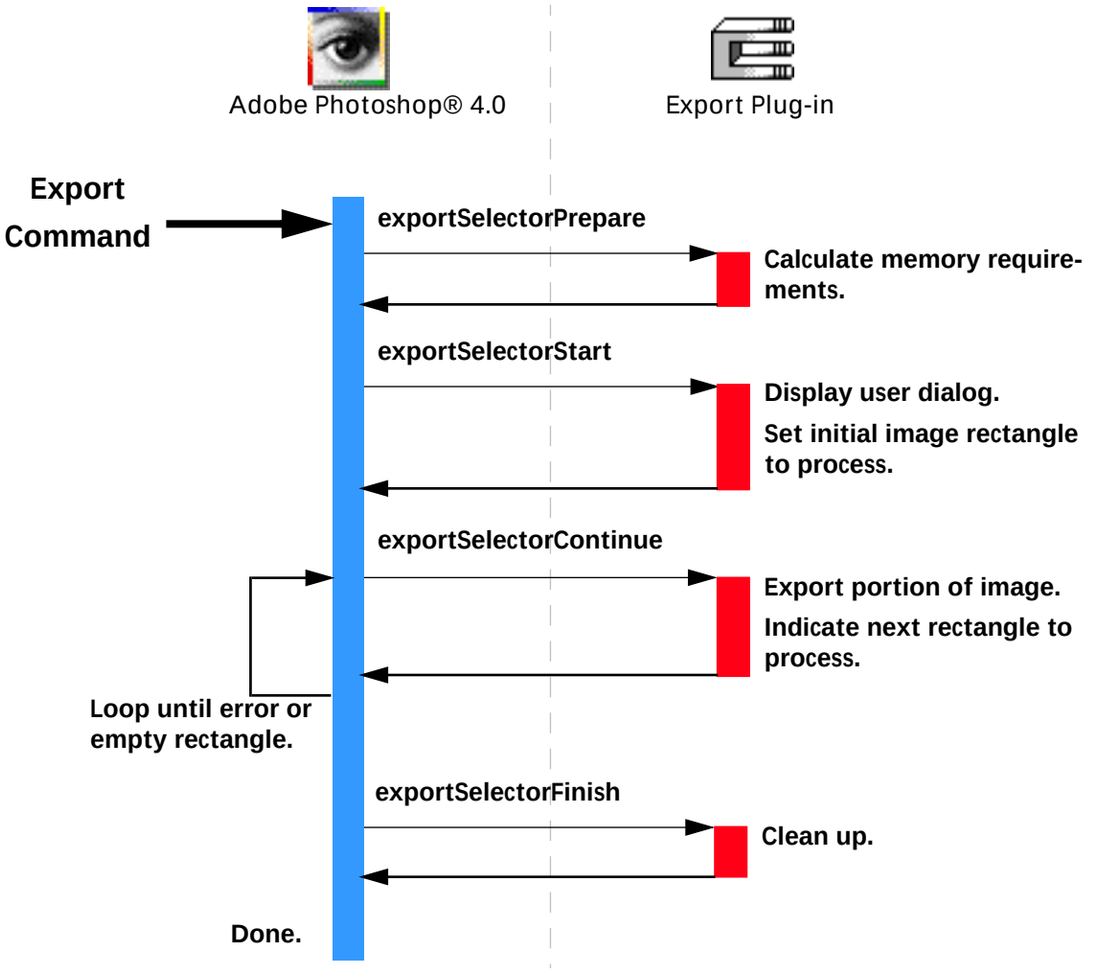
Examples/IllustratorExport

IllustratorExport 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.

Examples/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, loPlane & hiPlane

During this call, your plug–in module should set *theRect*, *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`, `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` 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`; 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` 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 6–2: 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.
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.
int16	loPlane	Your exportSelectorStart and exportSelectorContinue handlers should set these fields to the first and last planes to process next.
int16	hiPlane	

Table 6–2: 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.
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 6–2: 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.
char[178]	reserved	Reserved for future use. Set to zero.

7. 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 7–1: Filter file types

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

Examples/Filter/Dissolve–with–AppleScript

Dissolve–with–AppleScript is a sample filter plug–in which also demonstrates how to manipulate layers. It's terminology and scripting is built for AppleScript compatibility.

Examples/Filter/Dissolve–sans–AppleScript

Dissolve–sans–AppleScript is exactly the same filter as *Dissolve– with–AppleScript*, except its terminology and scripting is built for host scripting only and is not AppleScript compliant. It is an example of how to quickly build a Filter plug-in module for host-scripting compliance without having to delve into the caveats for AppleScript compatibility issues. An 'aete' resource is still required so that parameters are displayed correctly in the Actions palette.

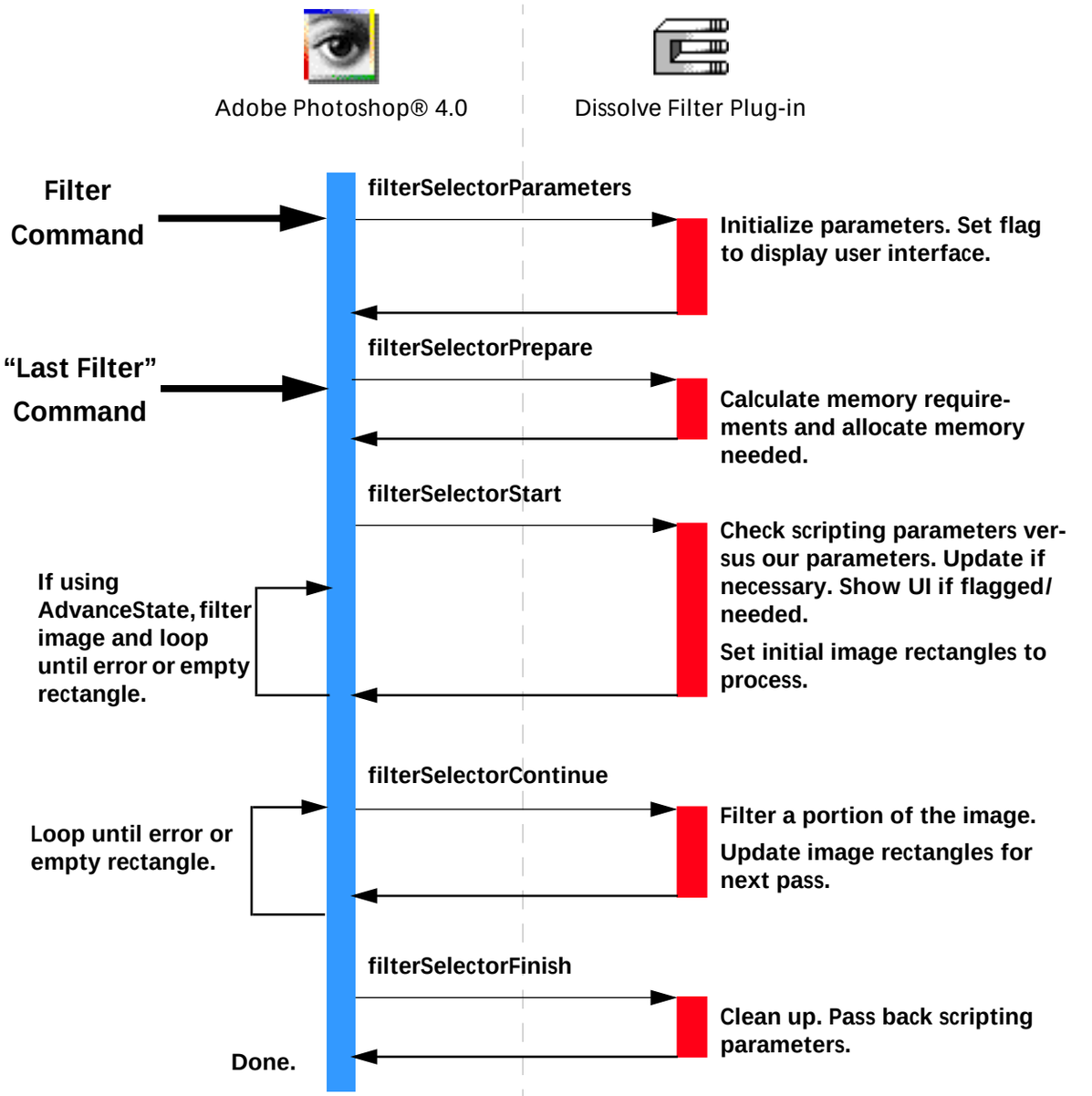
Examples/Filter/Propetizer

Propetizer is a utility filter that demonstrates different properties.

Examples/Filter/ColorMunger

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

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, planes & filterRect
The fields such as *imageSize*, *planes*, and *filterRect*, 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

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    // module doesn't support <mode> images
```

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 7–2: 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.
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.
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	
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).

Table 7–2: FilterRecord structure (Continued)

Type	Field	Description
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.
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.
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.

Table 7–2: FilterRecord structure (Continued)

Type	Field	Description
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.
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.

Table 7–2: FilterRecord structure (Continued)

Type	Field	Description
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.
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.

Table 7–2: FilterRecord structure (Continued)

Type	Field	Description
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 (`plugInWantsErrorOnBoundsExceptio n`). 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`. Futu re 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.
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	

Table 7–2: FilterRecord structure (Continued)

Type	Field	Description
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.
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	
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.

Table 7–2: FilterRecord structure (Continued)

Type	Field	Description
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.
char[78]	reserved	Reserved for future use. Set to zero.

8. Format Modules

Format plug–in modules, sometimes referred to as Image Format, or File Format modules, are used to add new file types to the **Open...**, **Save**, and **Save As...** commands. Adobe Photoshop ships with several file format modules including GIF, MacPaint, and BMP.

Import and Export modules may also be used to read and write files. You should create a Format module if you want your users to treat your files in the same fashion as native Photoshop files. Use a Format module if:

1. You want users to be able to create, modify, save, and re–open files in your format. If your format uses a lossy compression algorithm, you may want to consider image degradation issues for this situation.

2. You want users to be able to double–click a document to launch Photoshop or associate your file extension with the Photoshop application.

You may *not* want to use a Format module if:

1. With respect to Photoshop, your file format is read–only or write–only.

2. The image compression and/or color space conversion on multiple reading and writing would result in unacceptable image degradation.

Table 8–1: Format file types

OS	Filetype/extension
Mac OS	8BIF
Windows	.8BI

Examples/Format/SimpleFormat

SimpleFormat is a sample Format module. This module is written to use the `AdvanceStateProc` callback, introduced in Photoshop 3.0.

Format module operations

File Format plug–in modules have two main functions: reading an image from a file, and writing an image to a file.

Reading a file is a two step process:

1. *formatSelectorFilterFile* is used to determine whether a Format module can read a particular file. This selector is called when the user performs an **Open...** command, and is described in more detail on the next page.

2. The *read* sequence is used to read image files.

Writing a file consists of three sequences:

1. The *options* sequence is used to request save options from the user. It will only be used when first saving a document in a particular format.

2. The *estimate* sequence estimates the file size so that the host can decide whether there is enough disk space available.

3. The *write* sequence actually writes the file.

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.

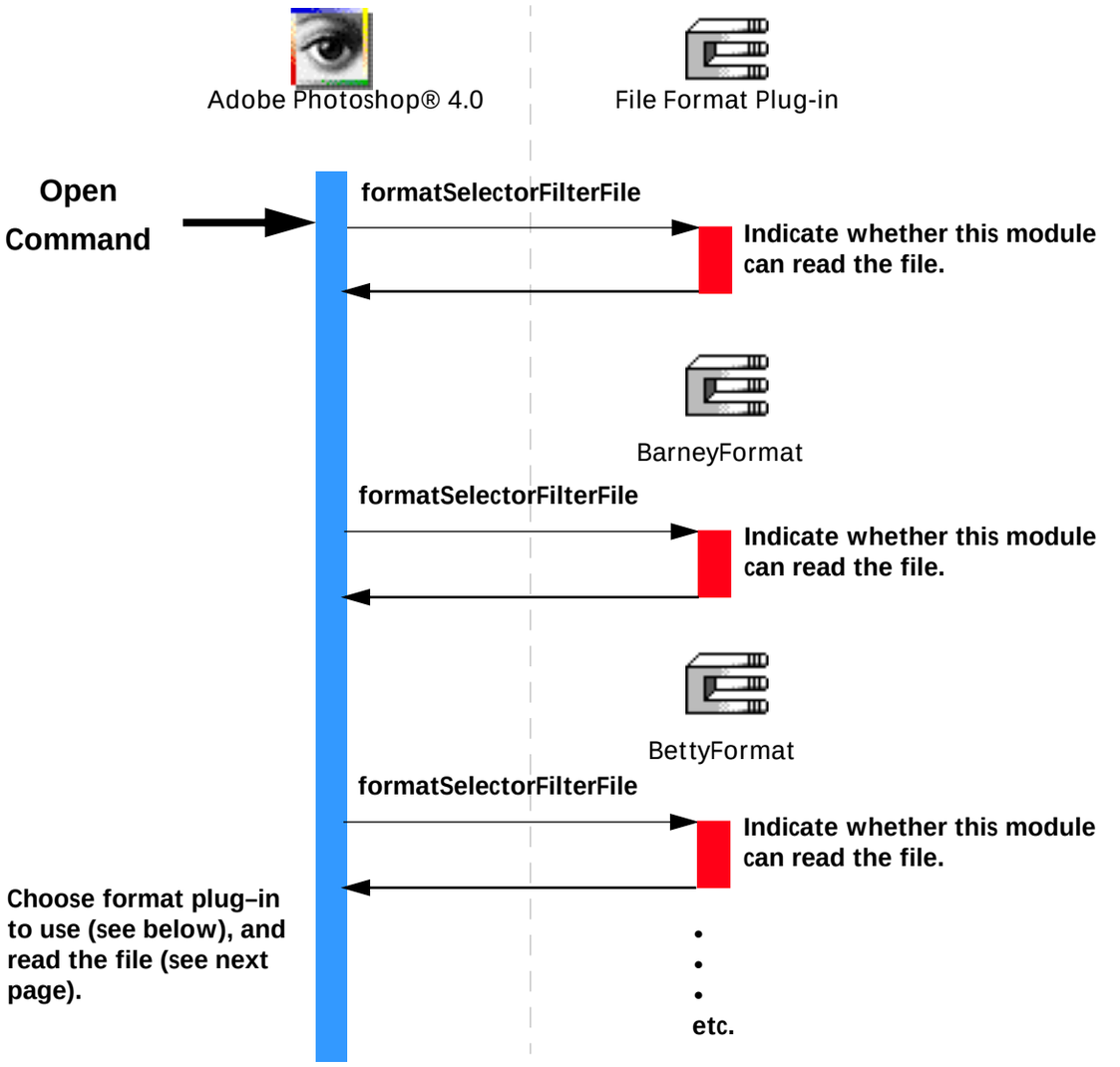
The most effective way to do this is:

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.

Reading a file (file filtering)



When the user selects a file with the **Open...** command from the file menu, there may be one or more Format modules that list the Mac OS file type or Windows file extension as a supported format. For each such plug–in module, Photoshop will call the plug–in with `formatSelectorFilterFile`. The plug–in module should then examine the file to determine whether the file is one that it can process, and indicate this in its `result` parameter:

```
if (module can read this file)
    *result = noErr;
else
    *result = formatCannotRead;
```

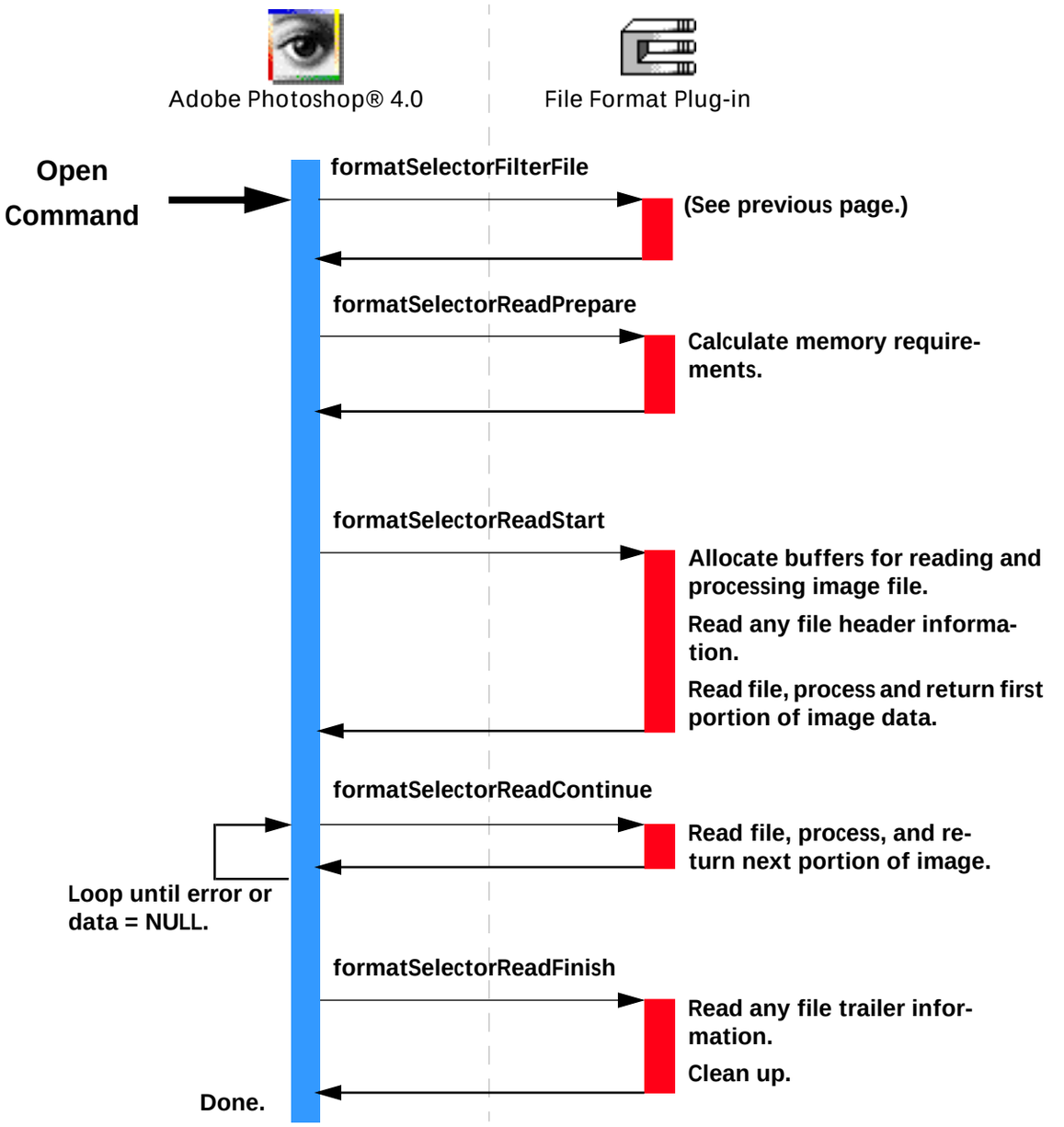
If more than one format module can read the file, Photoshop uses the following priority scheme to determine which plug–in module to use:

1. The module with the first `PICategoryProperty` string (sorted alphabetically) will be used. Modules with no `PICategoryProperty` will default to their `PINameProperty` for this comparison.

2. If two or more modules have matching category names, the module with the highest `PIPriorityProperty` value will be used.

3. If two or more modules have matching category and priority, which module will be selected is undefined.

Reading a file (read sequence)



formatSelectorFilterFile

This selector is discussed in more detail on the previous page. The rest of this sequence will be called only if your plug–in module returns `noErr` from this call, and your module is selected by the plug–in host to process the file.

formatSelectorReadPrepare

This selector allows 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.

formatSelectorReadStart

This selector allows the plug–in module to begin its interaction with the host.

Scripting at formatSelectorReadStart

If you are supporting scripting, read any scripting parameters here to override any default parameters. The scripting system will also return whether to show your dialog or not.

imageMode & imageSize
You should initialize *imageMode*, *imageSize*, *depth*, *planes*, *imageHRes* and *imageVRes*.

Reading an indexed color image (redLut, greenLut & blueLUT)
If an indexed color image is being opened, you should also set *redLUT*, *greenLUT* and *blueLUT*.

imageRsrcData
If your plug–in has a block of image resources you wish to have processed, you should read it in from the file and set *imageRsrcData* to be a handle to the resource data. For more information about Photoshop image resources, see chapter *11. Document File Formats*.

theRect, loPlane & hiPlane
Your plug–in should allocate and read the first pixel image data buffer as appropriate. The area of the image being returned to the plug–in host is specified by *theRect*, *loPlane*, and *hiPlane*.

data, colBytes, rowBytes, planeBytes & planeMap
The actual pixel data is pointed by *data*. The *colBytes*, *rowBytes*, *planeBytes*, and *planeMap* fields must specify the organization of the data.

Photoshop is very flexible in the format in which image data can be read. For example, to read just the red plane of an RGB color image, use the parameter values in Table 8–2.

Table 8–2: Read red plane of RGB

Parameter	Value
loPlane	0
hiPlane	0
colBytes	1
rowBytes	width of the area being read
planeBytes	ignored, since `loPlane=hiPlane`.

If you wish to read the RGB data in interleaved form (RGBRGB...), use the values shown in Table 8–3.

Table 8–3: Read RGB data in interleaved form

Parameter	Value
loPlane	0
hiPlane	2
colBytes	3
rowBytes	3 * width of the area being read
planeBytes	1

formatSelectorReadContinue

This selector may be used to process a sequence of areas within the image. Your handler should process any incoming data and then, just as with the start call, set up `theRect`, `loPlane`, `hiPlane`, `planeMap`, `data`, `colBytes`, `rowBytes`, and `planeBytes` to describe the next chunk of the image being returned. The host will keep calling with `formatSelectorReadContinue` until you set `data=NULL`.

formatSelectorReadFinish

The `formatSelectorReadFinish` selector allows you to clean–up from the read operation just performed. This call is made by the plug–in host if and only if `formatSelectorReadStart` returned without error, even if one of the `formatSelectorReadContinue` calls results in an error.

Most plug–ins will at least need to free the buffer used to return pixel data if this has not been done previously.

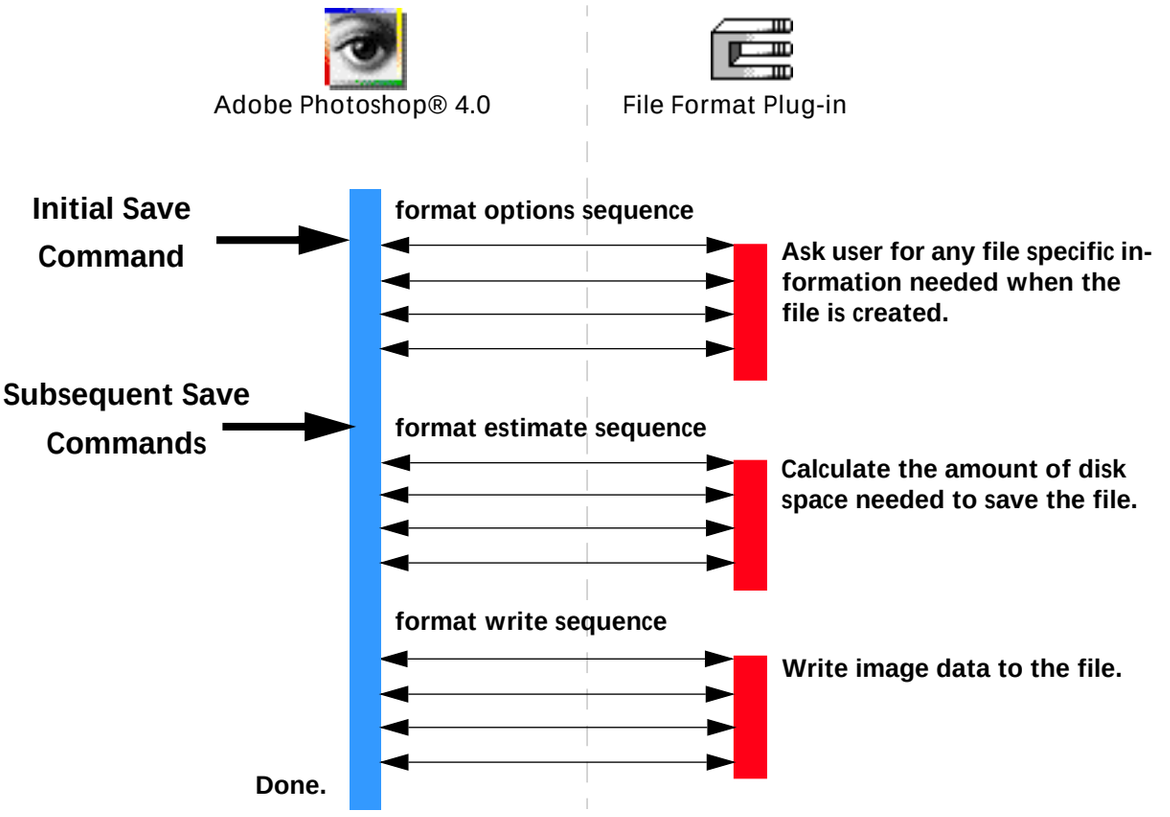
If Photoshop detects *Command–period* in the Mac OS or *Escape* in Windows while processing the results of a `formatSelectorReadContinue` call, it will call `formatSelectorReadFinish`.

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

Scripting at formatSelectorReadFinish

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.

Writing a file



Writing a file involves either two or three distinct sequences, each similar in structure. The details of these sequences are described on the following pages.

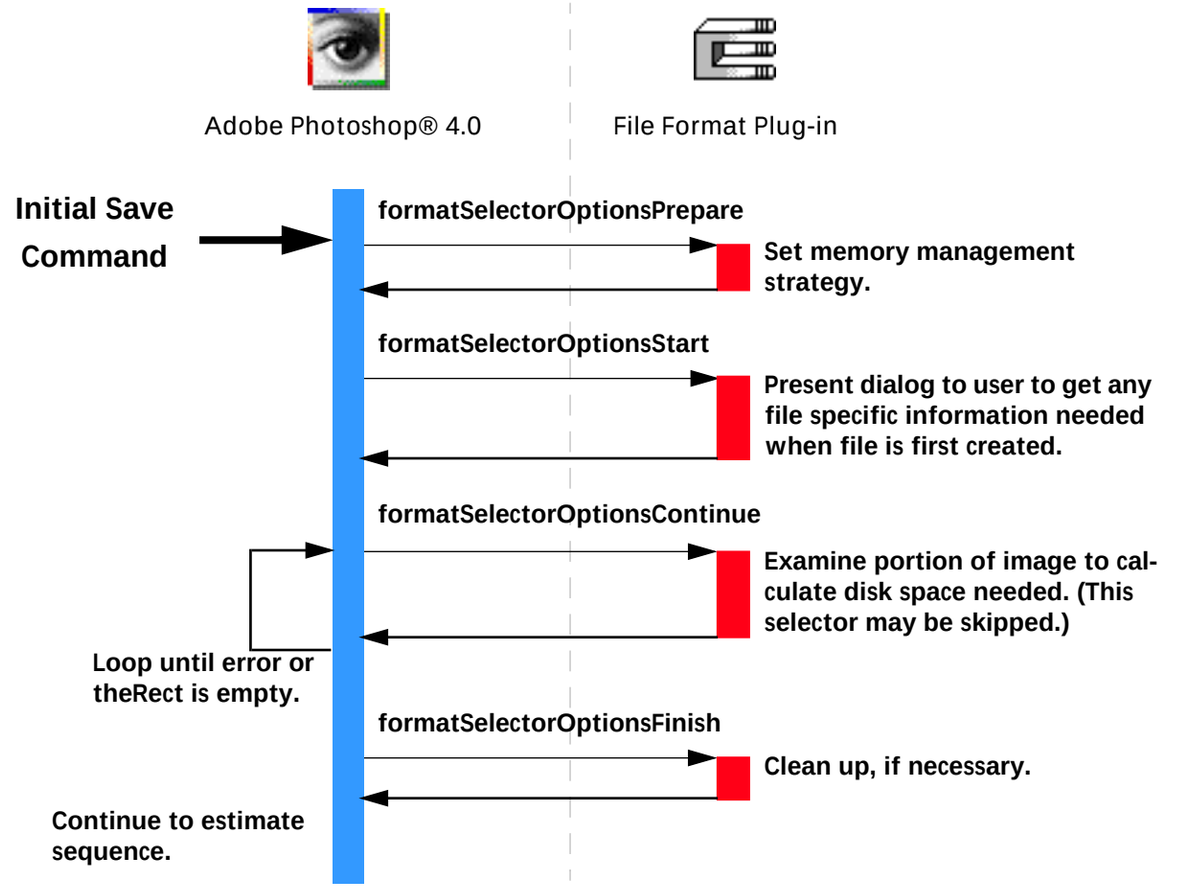
When a document is first saved, Photoshop calls your Format plug–in in this order:

1. the *options* sequence,
2. the *estimate* sequence,
3. the *write* sequence.

After a document has been saved once, each time the user saves the file again, the plug-in is called without the options sequence:

1. the *estimate* sequence,
2. the *write* sequence.

Writing a file (options sequence)



formatSelectorOptionsPrepare

formatSelectorOptionsPrepare allows 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.

formatSelectorOptionsStart

formatSelectorOptionsStart allows you to determine whether the current document can be saved in your file format, and if necessary, get any file options from the user.

Scripting at formatSelectorOptionsStart

If you are supporting scripting, read any scripting parameters here to override any default parameters. The scripting system will also return whether to show your dialog or not.

If you need to examine the image to compute the file size, you can iterate through the image data in *formatSelectorOptionsContinue* in the same fashion as is done when writing the file to request sections of the image.

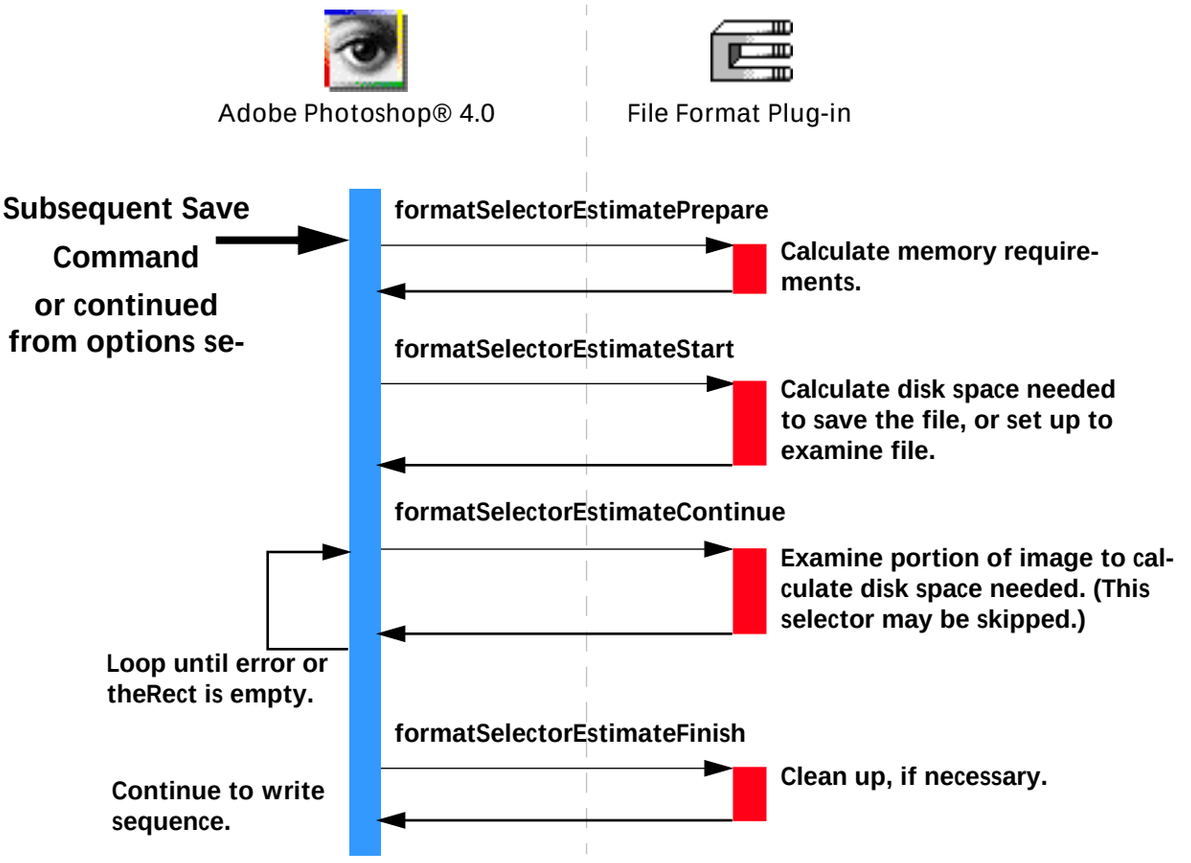
formatSelectorOptionsContinue

If the `data` field in the `FormatRecord` is set to `NULL` in the `formatSelectorOptionsStart` call, this selector will not be called at all. Otherwise, your plug–in can request parts of the image from which you determine whether your plug–in module can store the file. Refer to `formatSelectorWriteStart` and `formatSelectorWriteContinue` on the following page for details.

You may also use the `AdvanceStateProc` to iterate through the image.

formatSelectorOptionsFinish
Perform any clean up, if necessary.

Writing a file (estimate sequence)



formatSelectorEstimatePrepare

formatSelectorWritePrepare allows 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.

formatSelectorEstimateStart

Calculate the disk space needed to save the file. If you can calculate the file size without examining the image data, you can set the *minDataBytes* and *maxDataBytes* fields in the `FormatRecord` to the approximate size of your file (due to compression, you may not be able to exactly calculate the final size), and set `data=NULL`.

If you need to examine the image to compute the file size, you can iterate through the image data in *formatSelectorEstimateContinue* in the same fashion as is done when writing the file to request sections of the image.

formatSelectorEstimateContinue

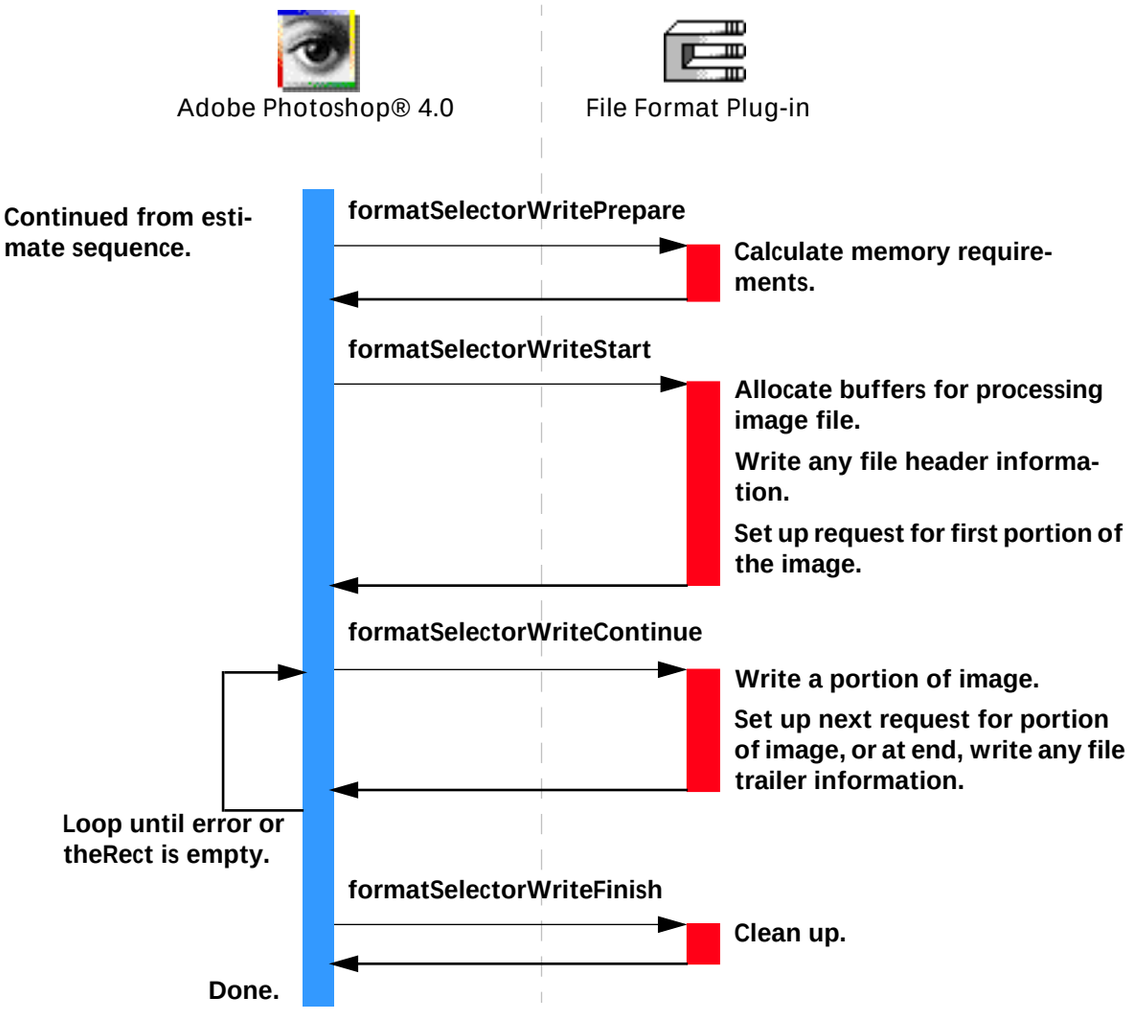
If the `data` field in the `FormatRecord` is set to `NULL` in the `formatSelectorEstimateStart` call, this selector will not be called at all. Otherwise, your plug–in can request parts of the image from which you can compute the minimum and maximum bytes to store the file. Refer to `formatSelectorWriteStart` and `formatSelectorWriteContinue` on the following page for details.

You may also use the `AdvanceStateProc` to iterate through the image.

formatSelectorEstimateFinish

Perform any clean up, if necessary.

Writing a file (write sequence)



formatSelectorWritePrepare

formatSelectorWritePrepare allows 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.

formatSelectorWriteStart

The formatSelectorReadStart selector call allows your plug–in module to begin writing the file. On entry, the file to be written is open, and the file pointer is positioned at the start of the file. You should write any file header information, such as image resources, to the file.

Your plug–in should then indicate which portion of the image data to provide for the first *formatSelectorWriteContinue* call.

theRect, loPlane & hiPlane

The area of the image being requested from the plug–in host is specified by *theRect*, *loPlane*, and *hiPlane*.

data

The actual pixel data is pointed by *data*.

colBytes, rowBytes, planeBytes & planeMap

You must specify the organization of the data to be returned by the plug–in host in the *colBytes*, *rowBytes*, *planeBytes*, and *planeMap* fields.

Photoshop is very flexible in the format in which image data can be delivered to the plug–in. For example, to return just the red plane of an RGB color image, use the parameter values in Table 7-3.

Table 8–4: Return red plane of RGB

Parameter	Value
loPlane	0
hiPlane	0
colBytes	1
rowBytes	width of the area being returned
planeBytes	ignored, since `loPlane=hiPlane`.

If you wish to return the RGB data in interleaved form (RGBRGB...), use the values shown in Table 7-4.

Table 8–5: Return RGB data in interleaved form

Parameter	Value
loPlane	0
hiPlane	2
colBytes	3
rowBytes	3 * width of the area being read
planeBytes	1

formatSelectorWriteContinue

This selector is call repeatedly by the plug–in host to provide your plug–in module some or all of the image data; your plug–in module should write this data to file. If successful, set up `theRect`, `loPlane`, `hiPlane`, `planeMap`, `data`, `colBytes`, `rowBytes`, and `planeBytes` to describe the next chunk of the image being requested.

The host will keep calling your `formatSelectorReadContinue` handler until you set `theRect` to an empty rectangle. Before returning after the last image data has been written, write any file trailer information to the file.

formatSelectorWriteFinish

formatSelectorWriteFinish allows you to clean–up from the write operation just performed. This call is made by the plug–in host if and only if `formatSelectorWriteStart` returned without error, even if one of the `formatSelectorWriteContinue` calls results in an error.

Most plug–ins will at least need to free the buffer used to hold pixel data if this has not been done previously.

If Photoshop detects *Command–period* in the Mac OS or *Escape* in Windows while processing the results of a `formatSelectorWriteContinue` call, it will call the `formatSelectorWriteFinish` routine.

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

Scripting at formatSelectorWriteFinish

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.

Image Resources

Photoshop documents can have other properties associated with them besides pixel data. For example, documents typically contain page setup information and pen tool paths.

Photoshop supports the concept of a block of data known as the image resources for a file. Format plug–in modules can store and retrieve this information if the file format definition allows for a place to put such an arbitrary block of data (e.g., a `TIFF` tag or a `PicComment`).

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 `PIFormat.h`.

```
#define formatBadParameters     -30500    // an error with the interface
#define formatCannotRead        –30501    // no scanner installed
```

Note: When writing a file, if your plug–in module sets `result` to any non–zero value, then no subsequent selector calls will be made by Photoshop. For example, if in your `formatSelectorOptionsStart` handler, you determine that the file cannot be saved, then none of the remaining selectors will be called: options, estimate, nor write.

The Format parameter block

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

Table 8–6: FormatRecord 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. See chapter 3.
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. See chapter 3.
int32	maxData	Photoshop initializes this field to the maximum of number of bytes it can free up. The plug–in may reduce this value during the prepare routines. The continue routines should process the image in pieces no larger than maxData less the size of any large tables or scratch areas it has allocated.
int32	minDataBytes	These fields give the limits on the data fork space needed to write the file. The plug–in should set these during the estimate sequence of selector calls.
int32	maxDataBytes	
int32	minRsrcBytes	These fields give the limits on the resource fork space needed to write the file. The plug–in should set these during the estimate sequence of selector calls.
int32	maxRsrcBytes	
int32	dataFork	The reference number for the data fork of the file to be read during the read sequence or written during the write sequence. During the options and estimate sequences, this field is undefined. In Windows, this is the file handle of the file returned by OpenFile( ).
int32	rsrcFork	The reference number for the resource fork of the file to be read during the read sequence or written during the write sequence. During the options and estimate sequences, this field is undefined. In Windows, this field is undefined.
int16	imageMode	The formatSelectorReadStart routine should set this field to inform Photoshop what mode image is being imported (grayscale, RGB Color, etc.). See the header file for possible values. Photoshop will set this field before it calls formatSelectorOptionsStart, formatSelectorEstimateStart, or formatSelectorWriteStart.
Point	imageSize	The formatSelectorReadStart routine should set this field to inform Photoshop of the image's width, imageSize.h and height, imageSize.v in pixels. Photoshop will set this field before it calls formatSelectorOptionsStart, formatSelectorEstimateStart, or formatSelectorWriteStart.

Table 8–6: FormatRecord structure (Continued)

Type	Field	Description
int16	depth	The `formatSelectorReadStart` routine should set this field to inform Photoshop of the image's resolution in bits per pixel per plane. The only valid settings are 1 for bitmap mode images, and 8 for all other modes. Photoshop will set this field before it calls `formatSelectorOptionsStart`, `formatSelectorEstimateStart`, or `formatSelectorWriteStart`.
int16	planes	The `formatSelectorReadStart` routine should set this field to inform Photoshop of the number of channels in the image. For example, if an RGB image without alpha channels is being returned, this field should be set to 3. Photoshop will set this field before it calls `formatSelectorOptionsStart`, `formatSelectorEstimateStart`, or `formatSelectorWriteStart`. Because of the implementation of the plane map, Format and Import modules should never try to work with more than 16 planes at a time. The results would be unpredictable.
Fixed	imageHRes	The `formatSelectorReadStart` routine should set these fields to inform Photoshop of the image's horizontal and vertical resolution in terms of pixels per inch. This is a fixed point number (16 binary digits). Photoshop initializes these fields to 72 pixels per inch. Photoshop will set these fields before it calls `formatSelectorOptionsStart`, `formatSelectorEstimateStart`, or `formatSelectorWriteStart`. The current version of Photoshop only supports square pixels, so it ignores the `imageVRes` field. Plug–ins should set both fields anyway in case future versions of Photoshop support non–square pixels.
Fixed	imageVRes	
LookUpTable	redLUT	If an indexed color mode image is being returned, the `formatSelectorReadStart` routine should return the image's color table in these fields. If an indexed color document is being written, Photoshop will set these fields before it calls `formatSelectorOptionsStart`, `formatSelectorEstimateStart`, or `formatSelectorWriteStart`.
LookUpTable	greenLUT	
LookUpTable	blueLUT	
void *	data	The start and continue routines should return a pointer to the buffer where image data is or is to be stored in this field. After the entire image has been processed, the continue selectors should set this field to `NULL`. Note that the plug–in is responsible for freeing any memory pointed to by this field.
Rect	theRect	The plug–in should set this to the area of the image covered by the buffer specified in data.
int16	loPlane	The start and continue routines should set this to the first and last planes covered by the buffer specified in `data`. For example, if interleaved RGB data is being used, they should be set to 0 and 2.
int16	hiPlane	

Table 8–6: FormatRecord structure (Continued)

Type	Field	Description
int16	colBytes	The start and continue routines should set this field to the offset in bytes between columns of data in the buffer. This is usually 1 for non–interleaved data, or `hiPlane–loPlane+1` for interleaved data.
int32	rowBytes	The start and continue routines should set this field to the offset in bytes between rows of data in the buffer.
int32	planeBytes	The start and continue routines should set this field to the offset in bytes between planes of data in the buffers. This field is ignored if `loPlane=hiPlane`. It should be set to 1 for interleaved data.
PlaneMap (array of 16 int16's)	planeMap	This is initialized by the host to a linear map `planeMap[i]=i`. This is used to map plane (channel) numbers between the plug–in and the host. For example, Photoshop stores RGB images with an alpha channel in the order RGBA, whereas most frame buffers store the data in ARGB order. To work with the data in this order, the plug–in should set `planeMap[0]=3`, `planeMap[1]=0`, `planeMap[2]=1`, and `planeMap[3]=2`.
Boolean	canTranspose	If the host supports transposing images during or after reading or before or during writing, it should set this field to `TRUE`. Photoshop always sets this field to `TRUE`.
Boolean	needTranspose	Initialized by the host to `FALSE`. If the plug–in wishes to have the image transposed, and `canTranspose=TRUE`, it should set this field to `TRUE` during the start call.
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	hostModes	This field is used by the host to inform the plug–in which imageMode values it supports. If the corresponding bit is 1, LSB = bit 0, the mode is supported. This field can be used by plug–ins to disable reading unsupported file formats.

Table 8–6: FormatRecord structure (Continued)

Type	Field	Description
Handle	revertInfo	This field is set to NULL by Photoshop when a format for a file is first created. If this field is defined on a formatSelectorReadStart call, then treat the call as a revert and don't query the user. If it is NULL on the formatSelectorReadStart call, then query the user as appropriate and set up this field to store a handle containing the information necessary to read the file without querying the user for additional parameters (essential for reverting the file) and if possible to write the file without querying the user. The contents of this field are sticky to a document and will be duplicated when we duplicate the image format information for a document. On all formatSelectorOptions calls, leave revertInfo containing enough information to revert the document. Photoshop will dispose of this field when it disposes of the document, hence, the plug–in must call on Photoshop to allocate the data as well using the following callbacks or the callbacks provided in the Handle suite.
NewPIHandleProc	hostNewHdl	This is the same as the NewPIHandle callback described in chapter 3. This field existed before the Handle suite was defined.
DisposePIHandleProc	hostDisposeHdl	This is the same as the DisposePIHandle callback described in chapter 3. This field existed before the handle suite was defined.
Handle	imageRsrcData	During calls to the write sequence, this field contains a handle to a block of data to be stored in the file as image resource data. Since this handle is allocated before the write sequence begins, plug–ins must add any resources they want saved to the document during the options or estimate sequence. Since options is not always called, the best time is during the estimate sequence. This field is checked after each call to formatSelectorRead and formatSelectorContinue and as soon as it is not NULL Photoshop parses the handle as a block of image resource data for the current document.
int32	imageRsrcSize	This is the size of the handle imageRsrcData. It is really only relevant during the estimate sequence when it is provided instead of the actual resource data.
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 on the Macintosh.
BufferProcs *	bufferProcs	This field contains a pointer to the Buffer suite if it is supported by the host, otherwise NULL.
ResourceProcs *	resourceProcs	This field contains a pointer to the Pseudo–Resource suite if it is supported by the host, otherwise NULL.
ProcessEventProc	processEvent	This field contains a pointer to the ProcessEvent callback documented in chapter 3. It contains NULL if the callback is not supported.

Table 8–6: FormatRecord structure (Continued)

Type	Field	Description
DisplayPixelsProc	displayPixels	This field contains a pointer to the DisplayPixels callback documented in chapter 3. It contains NULL if the callback is not supported.
HandleProcs	handleProcs	This field contains a pointer to the Handle suite if it is supported by the host, otherwise NULL.
These fields are new since version 3.0 of Adobe Photoshop.		
OSType	fileType	This field contains the file type for filtering.
ColorServicesProc	colorServices	This field contains a pointer to the ColorServices callback documented in chapter 3. It contains NULL if the callback is not supported.
AdvanceStateProc	advanceState	The advanceState callback allows you to drive the interaction through the inner formatSelectorOptionsContinue loop without actually returning from the plug–in. If it returns an error, then the plug–in generally should treat this as an error formatSelectorOptionsContinue and pass it on when it returns. See chapter 3.
These fields are new since version 3.0.4 of Adobe Photoshop.		
PropertyProcs *	propertyProcs	A pointer to the Property callback suite. See chapter 3.
int16	tileWidth	The host reports the width and height of a tile, which would be the best unit to work in, if possible.
int16	tileHeight	
int16	tileOrigin	The origin of the tiling system.
These fields are new since version 4.0 of Adobe Photoshop.		
PIDescriptorParameters *	descriptorParameters	Descriptor callback 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*”.
char[212]	reserved	Reserved for future use. Set to zero.

9. Selection Modules

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

Table 9–1: Selection file types

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

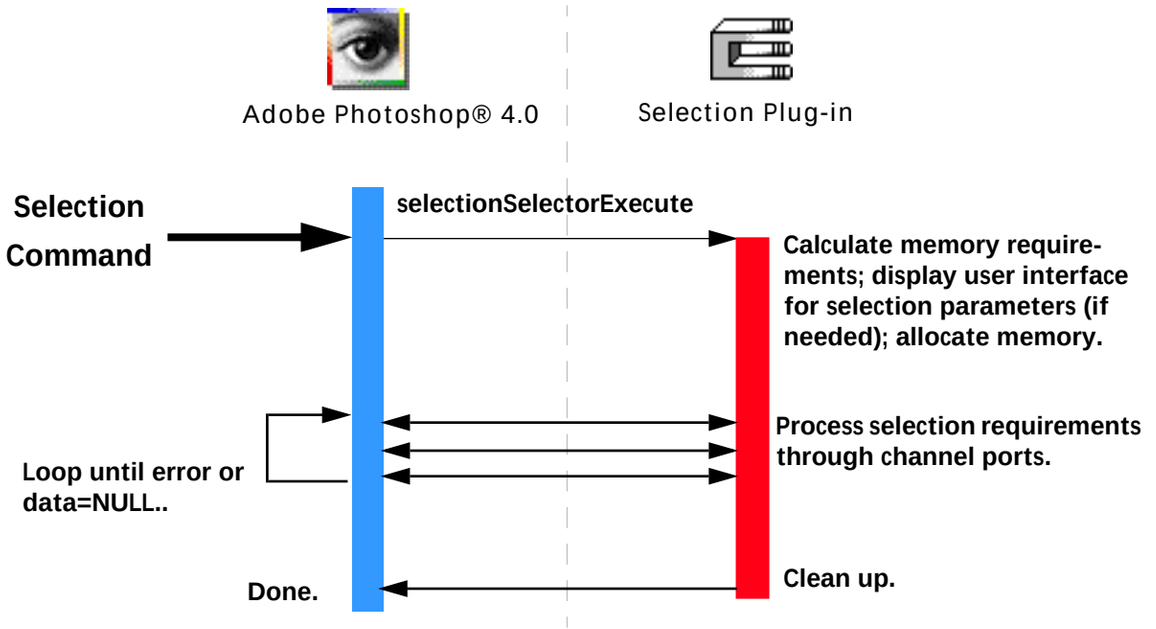
Examples/Selection/Selectorama

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

Examples/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 9–2: 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 9–3.
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 9–3: 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 9–3: 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 9–5.
void *	contextInfo	Pointer to additional info dependent on context.
const char *	name	Name of the channel.

Table 9–4: 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 9–5.
int16	padding	Reserved. Defaults to zero.
void *	contextInfo	Pointer to additional info dependent on context.
const char *	name	Name of the channel.

Table 9–5: 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 9–5: 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 9–6: 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 9–7: 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 9–7: 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 9–2.
WriteChannelDesc *	newSelection	Output for new selection. See table 9–4.
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 9–6.
int32	supportedTreatments	Mask indicating host supported treatments. See table 9–6.
char[256]	reservedBlock	Reserved for future use. Set to zero.

10. Scripting Plug-ins

Adobe Photoshop 4.0 introduces 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.

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, additional OLE automation has been added. Refer to Appendix B, *Adobe Photoshop 4.0 OLE Automation Programming Guide*.

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`.

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 10–14.
                'tyID',                     // parameter type ID. See table 10–4.
                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 10–14.
                'tyID',                     // property type ID. See table 10–4.
                "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 10–1: 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 10–2: 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 10–3: Scripting command syntax

Name	Description
event	Command being executed.
target	Item being acted upon.
key	Parameter key. See table 10–14.
value	Parameter value type. See table 10–4.

Table 10–4: 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 10–5: 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 10–2.

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 10–6: 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 10–7: 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 10–8: PIDescriptorSimpleReference structure

Type	Field	Description
DescriptorTypeID	desiredType	typeInteger, typeFloat, etc. See table 10–4.
DescriptorFormID	keyForm	Item type=formIndex, formName, or formProperty.
struct	_keyData	keyForm specific info. See table 10–9.

Table 10–9: 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 10–10: 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 10–11.

Table 10–11: 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 10–12.

Table 10–12: 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 10–13: 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 10–14: 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.
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.
keyNew	'Nw '	New.

Table 10–14: Predefined common keys and parameters (Continued)

Name	Code	Title/description
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

Even though version 4.0 of Photoshop does not support much more than basic calls from AppleScript back into the host, the 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, 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
```

11. Document File Formats

Adobe Photoshop saves a user's document in one of several formats, which are listed under the pop–up menu in the **Save** dialog. This chapter documents these standard formats.

The formats discussed in this chapter include Photoshop 3.0 native format, Photoshop 4.0 additions to the 3.0 file format, Photoshop EPS format, Filmstrip format, and TIFF format.

For more information about file formats, you may wish to consult the *Encyclopedia of Graphics File Formats* by James D. Murray & William vanRyper (1994, O'Reilly & Associates, Inc., Sebastopol, CA, ISBN 1–56592–058–9).

Image resource blocks

Image resource blocks are the basic building unit of several file formats, including Photoshop's native file format, JPEG, and TIFF. Image resources are used to store non–pixel data associated with an image, such as pen tool paths. (They are referred to as resource data because they hold data that was stored in the Macintosh's resource fork in early versions of Photoshop.)

The basic structure of Image Resource Blocks is shown in table .

Table 11–1: Image resource block

Type	Name	Description
OSType	Type	Photoshop always uses its signature, `8BIM`.
int16	ID	Unique identifier (see table 10–2).
PString	Name	A pascal string, padded to make size even (a null name consists of two bytes of 0)
int32	Size	Actual size of resource data. This does not include the `Type`, `ID`, `Name`, or `Size` fields.
Variable	Data	Resource data, padded to make size even

Image resources use several standard ID numbers, as shown in table 11–2. Not all file formats use all ID's. Some information may be stored in other sections of the file.

Table 11–2: Image resource IDs

ID		Description
Hex	Dec	
0x03E8	1000	Obsolete—Photoshop 2.0 only. Contains five `int16` values: number of channels, rows, columns, depth, and mode.
0x03E9	1001	Optional. Macintosh print manager print info record.
0x03EB	1003	Obsolete—Photoshop 2.0 only. Contains the indexed color table.
0x03ED	1005	`ResolutionInfo` structure. See Appendix A.
0x03EE	1006	Names of the alpha channels as a series of Pascal strings.
0x03EF	1007	`DisplayInfo` structure. See Appendix A .
0x03F0	1008	Optional. The caption as a Pascal string.
0x03F1	1009	Border information. Contains a fixed-number for the border width, and an int16 for border units (1=inches, 2=cm, 3=points, 4=picas, 5=columns).
0x03F2	1010	Background color. See the Colors file information in chapter 9.
0x03F3	1011	Print flags. A series of one byte boolean values (see Page Setup dialog): labels, crop marks, color bars, registration marks, negative, flip, interpolate, caption.
0x03F4	1012	Grayscale and multichannel halftoning information.
0x03F5	1013	Color halftoning information.
0x03F6	1014	Duotone halftoning information.
0x03F7	1015	Grayscale and multichannel transfer function.
0x03F8	1016	Color transfer functions.
0x03F9	1017	Duotone transfer functions.
0x03FA	1018	Duotone image information.
0x03FB	1019	Two bytes for the effective black and white values for the dot range.

Table 11–2: Image resource IDs (Continued)

ID		Description
Hex	Dec	
0x03FC	1020	Obsolete.
0x03FD	1021	EPS options.
0x03FE	1022	Quick Mask information. 2 bytes containing Quick Mask channel ID, 1 byte boolean indicating whether the mask was initially empty.
0x03FF	1023	Obsolete.
0x0400	1024	Layer state information. 2 bytes containing the index of target layer. 0=bottom layer.
0x0401	1025	Working path (not saved). See path resource format later in this chapter.
0x0402	1026	Layers group information. 2 bytes per layer containing a group ID for the dragging groups. Layers in a group have the same group ID.
0x0403	1027	Obsolete.
0x0404	1028	IPTC-NAA record. This contains the **File Info...** information.
0x0405	1029	Image mode for raw format files.
0x0406	1030	JPEG quality. Private.
0x07D0-0x0BB6	2000-2998	Path Information (saved paths)
0x0BB7	2999	Name of clipping path
0x2710	10000	Print flags information. 2 bytes version (=1), 1 byte center crop marks, 1 byte (=0), 4 bytes bleed width value, 2 bytes bleed width scale.

Path resource format

Photoshop stores the paths saved with an image in an image resource block. These resource blocks consist of a series of 26 byte path point records, and so the resource length should always be a multiple of 26.

Photoshop stores its paths as resources of type `8BIM` with IDs in the range 2000 through 2999. These numbers should be reserved for Photoshop. The name of the resource is the name given to the path when it was saved.

If the file contains a resource of type `8BIM` with an ID of 2999, then this resource contains a Pascal–style string containing the name of the clipping path to use with this image when saving it as an EPS file.

The path format returned by `GetProperty()` call is identical to what is described below. Refer to the `IllustratorExport` sample plug–in code to see how this resource data is constructed.

Path points

All points used in defining a path are stored in eight bytes as a pair of 32–bit components, vertical component first.

The two components are signed, fixed point numbers with 8 bits before the binary point and 24 bits after the binary point. Three guard bits are reserved in the points to eliminate most concerns over arithmetic overflow. Hence, the range for each component is `0xF0000000` to `0x0FFFFFFF` representing a range of –16 to 16. The lower bound is included, but not the upper bound.

This limited range is used because the points are expressed relative to the image size. The vertical component is given with respect to the image height, and the horizontal component is given with respect to the image width. `[0,0]` represents the top–left corner of the image; `[1,1]` (`[0x01000000,0x01000000]`) represents the bottom–right.

In Windows, the byte order of the path point components are reversed; you should swap the bytes when accessing each 32–bit value.

Path records

The data in a path resource consists of one or more 26-byte records. The first two bytes of each record is a selector to indicate what kind of path it is. For Windows, you should swap the bytes before accessing it as a short (`int16`).

Table 11–3: Path data record types

Selector	Description
0	Closed subpath length record
1	Closed subpath Bezier knot, linked
2	Closed subpath Bezier knot, unlinked
3	Open subpath length record
4	Open subpath Bezier knot, linked
5	Open subpath Bezier knot, unlinked
6	Path fill rule record
7	Clipboard record

The first 26-byte path record contains a selector value of 6, path fill rule record. The remaining 24 bytes of the first record are zeroes. Paths use even/

odd ruling. Subpath length records, selector value 0 or 3, contain the number of Bezier knot records in bytes 2 and 3. The remaining 22 bytes are unused, and should be zeroes. Each length record is then immediately followed by the Bezier knot records describing the knots of the subpath.

In Bezier knot records, the 24 bytes following the selector field contain three path points (described above) for:

1. the control point for the Bezier segment preceding the knot,

2. the anchor point for the knot, and

3. the control point for the Bezier segment leaving the knot.

Linked knots have their control points linked. Editing one point modifies the other to preserve collinearity. Knots should only be marked as having linked controls if their control points are collinear with their anchor. The control points on unlinked knots are independent of each other. Refer to the *Adobe Photoshop User Guide* for more information.

Clipboard records, `selector=7`, contain four fixed-point numbers for the bounding rectangle (top, left, bottom, right), and a single fixed-point number indicating the resolution.

Photoshop 3.0 files

This is the native file format for Adobe Photoshop 3.0. It supports storing all layer information.

Table 11–4: Photoshop 3.0 file types

OS	Filetype/extension
Mac OS	8BPS
Windows	.PSD

Photoshop 3.0 files under Windows

All data is stored in big endian byte order; under Windows you must byte swap short and long integers when reading or writing.

Photoshop 3.0 files under Mac OS

For cross–platform compatibility, all information needed by Adobe Photoshop 3.0 is stored in the data fork. For interoperability with other Macintosh applications, however, some information is duplicated in resources stored in the resource fork of the file.

For compatibility with image cataloging applications, the pnot resource id 0 contains references to thumbnail, keywords, and caption information stored in other resources. The thumbnail picture is stored in a 'PICT' resource, the keywords are stored in 'STR#' resource 128 and the caption text is stored in 'TEXT' resource 128. For more information on the format of these resources see *Inside Macintosh: QuickTime Components* and the *Extensis Fetch Awareness Developer's Toolkit.*

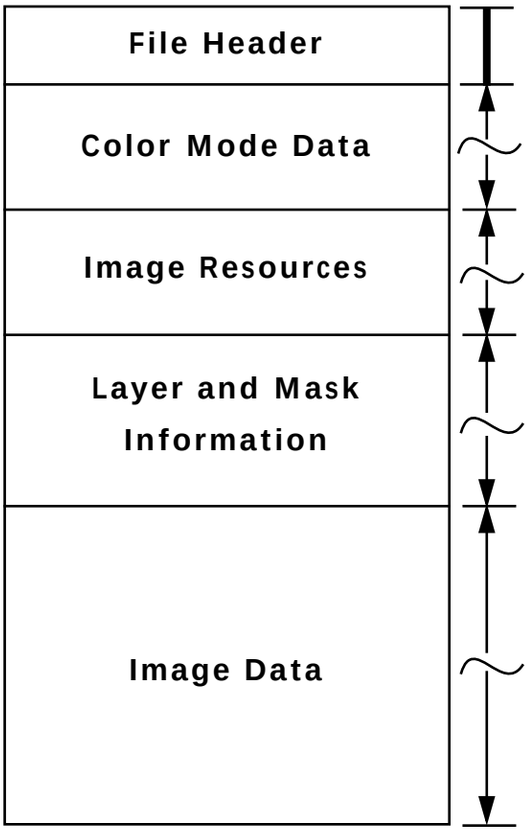
Photoshop also creates 'icl8' –16455 and 'ICN#' –16455 resources containing thumbnail images which will be shown in the Mac OS Finder.

All of the data from Photoshop's File Info dialog is stored in 'ANPA' resource 10000. The data in this resource is stored as an IPTC–NAA record 2 and should be readable by various tools from Iron Mike. For more information on the format of this resource contact:

IPTC–NAA Digital Newsphoto Parameter Record
Newspaper Association of America
The Newspaper Center
11600 Sunrise Valley Drive
Reston VA 20091

Photoshop 3.0 file format

The file format for Photoshop 3.0 is divided into five major parts.



The file header is fixed length, the other four sections are variable in length.

When writing one of these sections, you should write all fields in the section, as Photoshop may try to read the entire section. Whenever writing a file and skipping bytes, you should explicitly write zeros for the skipped fields.

When reading one of the length delimited sections, use the length field to decide when you should stop reading. In most cases, the length field indicates the number of bytes, not records, following.

File header section

The file header contains the basic properties of the image.

Table 11–5: File header

Length	Name	Description
4 bytes	Signature	Always equal to 8BPS. Do not try to read the file if the signature does not match this value.
2 bytes	Version	Always equal to 1. Do not try to read the file if the version does not match this value.
6 bytes	Reserved	Must be zero.
2 bytes	Channels	The number of channels in the image, including any alpha channels. Supported range is 1 to 24.
4 bytes	Rows	The height of the image in pixels. Supported range is 1 to 30,000.
4 bytes	Columns	The width of the image in pixels. Supported range is 1 to 30,000.
2 bytes	Depth	The number of bits per channel. Supported values are 1, 8, and 16.
2 bytes	Mode	The color mode of the file. Supported values are: Bitmap=0; Grayscale=1; Indexed=2; RGB=3; CMYK=4; Multichannel=7; Duotone=8; Lab=9.

Color mode data section

Only indexed color and duotone have color mode data. For all other modes, this section is just 4 bytes: the length field, which is set to zero.

For indexed color images, the length will be equal to 768, and the color data will contain the color table for the image, in non–interleaved order.

For duotone images, the color data will contain the duotone specification, the format of which is not documented. Other applications that read Photoshop files can treat a duotone image as a grayscale image, and just preserve the contents of the duotone information when reading and writing the file.

Table 11–6: Color mode data

Length	Name	Description
4 bytes	Length	The length of the following color data.
Variable	Color data	The color data.

Image resources section

The third section of the file contains image resources. As with the color mode data, the section is indicated by a length field followed by the data. The image resources in this data area are described in detail earlier in this chapter.

Table 11–7: Image resources

Length	Name	Description
4 bytes	Length	Length of image resource section.
Variable	Resources	Image resources.

Layer and mask information section

The fourth section contains information about Photoshop 3.0 layers and masks. The formats of these records are discussed later in this chapter. If there are no layers or masks, this section is just 4 bytes: the length field, which is set to zero.

Table 11–8: Layer and mask information

Length	Name	Description
4 bytes	Length	Length of the miscellaneous information section.
Variable	Layers	Layer info. See table 11–11.
Variable	Masks	One or more layer mask info structures. See table 11–14.

Image data section

The image pixel data is the last section of a Photoshop 3.0 file. Image data is stored in planar order, first all the red data, then all the green data, etc. Each plane is stored in scanline order, with no pad bytes.

If the compression code is 0, the image data is just the raw image data.

If the compression code is 1, the image data starts with the byte counts for all the scan lines (rows * channels), with each count stored as a two–byte value. The RLE compressed data follows, with each scan line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine `PackBits`, and the TIFF standard.

Table 11–9: Image data

Length	Name	Description
2 bytes	Compression	Compression method. Raw data = 0, RLE compressed = 1.
Variable	Data	The image data. Planar order = RRR GGG BBB, etc.

Layer and mask records

Information about each layer and mask in a document is stored in the fourth section of the file. The complete, merged image data is not stored here; it resides in the last section of the file.

The first part of this section of the file contains layer information, which is divided into layer structures and layer pixel data, as shown in table 11–10. The second part of this section contains layer mask data, which is described in table 11–17.

Table 11–10: Layer info section

Length	Name	Description
4 bytes	Length	Length of the layers info section, rounded up to a multiple of 2.
Variable	Layers structure	Data about each layer in the document. See table 11–11.
Variable	Pixel data	Channel image data for each channel in the order listed in the layers structure section. See table 11–16.

Table 11–11: Layer structure

Length	Name	Description
2 bytes	Count	Number of layers. If <0, then number of layers is absolute value, and the first alpha channel contains the transparency data for the merged result.
Variable	Layer	Information about each layer (table 11–16).

Table 11–12: Layer records

Length	Name	Description
4 bytes	Layer top	The rectangle containing the contents of the layer.
4 bytes	Layer left	
4 bytes	Layer bottom	
4 bytes	Layer right	
2 bytes	Number channels	The number of channels in the layer.
Variable	Channel length info	Channel information. This contains a six byte record for each channel. See table 11–13.
4 bytes	Blend mode signature	Always 8BIM.
4 bytes	Blend mode key	'norm' = normal 'dark' = darken 'lite' = lighten 'hue ' = hue 'sat ' = saturation 'colr' = color 'lum ' = luminosity 'mul ' = multiply 'scrn' = screen 'diss' = dissolve 'over' = overlay 'hLit' = hard light 'sLit' = soft light 'diff' = difference
1 byte	Opacity	0 = transparent ... 255 = opaque
1 byte	Clipping	0 = base, 1 = non–base

Table 11–12: Layer records (Continued)

Length	Name	Description
1 byte	Flags	bit 0 = transparency protected bit 1 = visible
1 byte	(filler)	(zero)
4 bytes	Extra data size	Length of the extra data field. This is the total length of the next five fields.
24 bytes, or 4 bytes if no layer mask.	Layer mask data	See table 11–14.
Variable	Layer blending ranges	See table 11–15.
Variable	Layer name	Pascal string, padded to a multiple of 4 bytes.

Table 11–13: Channel length info

Length	Name	Description
2 bytes	Channel ID	0 = red, 1 = green, etc. –1 = transparency mask –2 = user supplied layer mask
4 bytes	Length	Length of following channel data.

Table 11–14: Layer mask data

Length	Name	Description
4 bytes	Size	Size of layer mask data. This will be either 0x14, or zero (in which case the following fields are not present).
4 bytes	Top	Rectangle enclosing layer mask.
4 bytes	Left	
4 bytes	Bottom	
4 bytes	Right	
1 byte	Default color	0 or 255
1 byte	Flags	bit 0 = position relative to layer bit 1 = layer mask disabled bit 2 = invert layer mask when blending
2 bytes	Padding	Zeros

Table 11–15: Layer blending ranges data

Length	Name	Description
4 bytes	Length	Length of layer blending ranges data
4 bytes	Composite gray blend source	Contains 2 black values followed by 2 white values. Present but irrelevant for Lab & Grayscale.
4 bytes	Composite gray blend destination	Destination Range
4 bytes	First channel source range	First channel source
4 bytes	First channel destination range	First channel destination

Table 11–15: Layer blending ranges data (Continued)

Length	Name	Description
4 bytes	Second channel source range	Second channel source
4 bytes	Second channel destination range	Second channel destination
...	...	...
4 bytes	Nth channel source range	Nth channel source
4 bytes	Nth channel destination range	Nth channel destination

Table 11–16: Channel image data

Length	Name	Description
2 bytes	Compression	0 = Raw Data, 1 = RLE compressed.
Variable	Image data	If the compression code is 0, the image data is just the raw image data calculated as `((LayerBottom–LayerTop)*(LayerRight–LayerLeft))`. If the compression code is 1, the image data starts with the byte counts for all the scan lines in the channel `(LayerBottom–LayerTop)`, with each count stored as a two–byte value. The RLE compressed data follows, with each scan line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine PackBits, and the TIFF standard. If the Layer's Size, and therefore the data, is odd, a pad byte will be inserted at the end of the row.

Table 11–17: Layer mask data

Length	Name	Description
2 bytes	Overlay color space	Overlay color space (undocumented).
8 bytes	Color components	4 * 2 byte color components
2 bytes	Opacity	0 = transparent, 100 = opaque.
1 byte	Kind	0=Color selected—i.e. inverted; 1=Color protected;128=use value stored per layer. This value is preferred. The others are for backward compatibility with beta versions.
1 byte	(filler)	(zero)

Photoshop 4.0 file format

The Photoshop 4.0 file format is an extension of the Photoshop 3.0 file format. It is essentially the same, with some additional image resources and resource blocks. Listed below are additional resources, each of which is stored in a separate image resource block.

Table 11–18: Photoshop 4.0 additional image resources

Length	Name	Description
1 byte	kCopyrightID	Boolean. Indicates whether image is copywritten or not.
Variable	kURLID	String with a URL for the document.

8 bytes	kGuidesID	int32 fVersion = 1; // version
		VPoint fGridCycle; // ignored for now
		int32 fGuideCount; // number of guides, can be zero

6 bytes	fGuideCountX	int32 fLocation; // location of guide
		VHSelect fDirection; // direction of guide (horizontal/ vertical)

An additional image resource block type has been added for Adjustment Layers. That, blending and mode differences, and more extensive documentation on the above additional resources will be forthcoming in the next revision of this SDK.

Photoshop EPS files

Photoshop 3.0 and later writes a high–resolution bounding box comment to the EPS file immediately following the traditional EPS bounding box comment. The comment begins with "%%HiResBoundingBox" and is followed by four numbers identical to those given for the bounding box except that they can have fractional components (i.e., a decimal point and digits after it). The traditional bounding box is written as the rounded version of the high resolution bounding box for compatibility.

Photoshop writes its image resources out to a block of data stored as follows:

```
%BeginPhotoshop: <length> <hex data>
```

Table 11–19: EPS parameters for BeginPhotoshop

Field	Definition
length	Length of the image resource data.
hex data	Image resource data in hexadecimal.

Photoshop includes a comment in the EPS files it writes so that it is able to read them back in again. Third party programs that write pixel–based EPS files may want to include this comment in their EPS files, so Photoshop can read their files.

The comment must follow immediately after the %% comment block at the start of the file. The comment is:

```
%ImageData: <columns> <rows> <depth> <mode> <pad channels> <block size>
<binary/hex> "<data start>"
```

Table 11–20: EPS parameters for ImageData

Field	Definition
columns	Width of the image in pixels.
rows	Height of the image in pixels.
depth	Number of bits per channel. Must be 1 or 8.
mode	Image mode. Bitmap/grayscale=1; Lab=2; RGB=3; CMYK=4.
pad channels	Number of other channels store in the file. Ignored when reading. Photoshop uses this to include a grayscale image that is printed on non-color PostScript printers.
block size	Number of bytes per row per channel. Will be either 1 or formula (below): 1=Data is interleaved. (columns*depth+7)/8=Data is stored in line-interleaved format, or there is only one channel.
binary/ascii	1=Data is in binary format. 2=Data is in hex ascii format.
data start	Entire PostScript line immediately preceding the image data. This entire line should not occur elsewhere in the PostScript header code, butit may occur at part of a line.

Filmstrip files

Adobe Premiere 2.0 and later supports the filmstrip file format. Premiere users can export any video clip as a filmstrip. Refer to the *Adobe Premiere User Guide* for more information.

Adobe Photoshop 3.0 supports the filmstrip file type to allow each frame to be individually painted. The filmstrip file format is fairly simple, and is described in this section.

A filmstrip consists of a sequence of equal sized 32–bit images, known as frames. The channel order in the file is Red, Green, Blue, Alpha.

After each frame is an arbitrarily sized leader area, in which any type of information may be embedded. Adobe Premiere puts the timecode and frame number for the frame in this area. This area is ignored by Photoshop when the file is read.

Following all the frames is a 16 row trailer frame (it has the same width as the other frames). Adobe Premiere writes a yellow and black diagonal pattern in this area. The lower right corner of this area is actually an information record that exists at the very end of the file. This record is located by seeking to the end of the file minus the size of the record, then reading the record and verifying the signature field that it contains.

```
// Definition for filmstrip info record

typedef struct {
    long                signature;   // 'Rand'
    long                numFrames;   // number of frames in file
    short               packing;     // packing method
    short               reserved;    // reserved, should be 0
    short               width;       // image width
    short               height;      // image height
    short               leading;     // horiz gap between frames
    short               framesPerSec;// frame rate
    char                spare[16];   // some spare data.
} FilmStripRec, **FilmStripHand;
```

Table 11–21: FilmStripRec structure

Type	Field	Description
long	signature	This field must be set to the code Rand and is used to verify the validity of the record.
long	numFrames	This is the total number of frames in the file.
short	packing	This is the packing method used, currently only a value of 0 is defined, for no packing.
short	width	The width of each image, in pixels.
short	height	The height of each image, in pixels.
short	leading	The height of the leading areas, in pixels.
short	framesPerSec	The rate at which the frames should be played.

To locate the filmstrip info record, seek to the end of the file minus (`sizeof(FilmStripRec)`), then read in the FilmStrip record. Check the signature field for the code Rand to test for validity.

To locate the data for a particular frame, seek to

```
(frame * width * (height+leading) * 4)
```

then read the number of bytes in

```
(width * height * 4).
```

If the data is being placed into a Mac OS GWorld, the channels must be re–arranged from Red–Green–Blue–Alpha to Alpha–Red–Green–Blue.

To write a FilmStrip file, write each frame sequentially into the file, including the leading areas.

Then write this block of bytes:

```
((width * (height+leading) * 4) – sizeof(FilmStripRec)).
```

Finally, fill in and write the FilmStrip record to the file.

Note: The packing field should currently be zero. In the future packing methods may be defined for filmstrips, so any software which reads filmstrips should examine this field before opening the file.

TIFF files

The same image resources information found in Photoshop 3.0 files are stored in TIFF files under tag number 34377 (see Image Resource Blocks and Image Resources earlier in this chapter).

For TIFF files the caption data is stored in an image description tag 270 and all the information is stored as an IPTC–NAA record 2 in tag 33723. The tag number was chosen by inspecting files written by Iron Mike software, and is supposed to be defined in a Rich TIFF specification. The tag is also specified in:

NSK TIFF

The Japan Newspaper Publishers & Editors Association
Nippon Press Center Building
2–2–1 Uchlsaiwai–cho
Chiyoda–ku, Tokyo 100

For more information about the TIFF format see:

TIFF Revision 6.0

`http://www.adobe.com/supportservice/devrelations/resources.html#tiff`

In reading the files, the following order is used with information read lower on the list replacing information read higher:

Image Description Tag (TIFF only)
IPTC–NAA Tag (TIFF only)

Note: It is a bug that the TIFF information comes prior to the image resource information on this list. This means that an edit to the TIFF info will not be recognized unless the image resource information is removed. The TIFF data may be moved to after the image resource information in a future version of Photoshop.

Table 11–22 describes the standard TIFF tags and tag values that Photoshop 3.0 and later is able to read and write.

TIFF files under the Mac OS

For cross–platform compatibility, all TIFF information is stored in the data fork. For interoperability with other Macintosh applications, however, some information is duplicated in resources stored in the resource fork of the file.

For compatibility with image cataloging applications, the pnot resource id 0 contains references to thumbnail, keywords, and caption information stored in other resources. The thumbnail picture is stored in a 'PICT' resource, the keywords are stored in 'STR#' resource 128 and the caption text is stored in 'TEXT' resource 128. For more information on the format of these resources see *Inside Macintosh: QuickTime Components* and the *Extensis Fetch Awareness Developer's Toolkit.*

All of the data from Photoshop's File Info dialog is stored in 'ANPA' resource 10000. The TIFF file also contains 'STR ' resource -16396 indicating the application that created the TIFF file. The string is "Adobe Photoshop™ 3.0" for Photoshop 3.0 and "Adobe Photoshop® 4.0" for Photoshop 4.0.

Photoshop also creates 'icl8' –16455 and 'ICN#' –16455 resources containing thumbnail images which will be shown in the Mac OS Finder.

Table 11–22: TIFF Tags

Tag	Photoshop reads	Photoshop writes
IFD	First IFD in file	Only one IFD per file
NewSubFileType	Ignored	0
ImageWidth	1 to 30000	1 to 30000
ImageLength	1 to 30000	1 to 30000
BitsPerSample	1, 2, 4, 8, 16 (all same)	1, 8, 16
Compression	1, 2, 5, 32773	1, 5
PhotometricInterpretation	0, 1, 2, 3, 5, 8	0 (1–bit), 1 (8–bit), 2, 3,5,8
FillOrder	1	No
ImageDescription	Printing Caption	Printing Caption
StripOffsets	Yes	Yes
SamplesPerPixel	1 to 24	1 to 24
RowsPerStrip	Any	Single strip if not compressed, multiple strips if compressed.
StripByteCounts	Required if compressed	Yes
XResolution	Yes	Yes
YResolution	Ignored (square pixels assumed)	Yes
PlanarConfiguration	1 or 2	1
ResolutionUnit	2 or 3	2
Predictor	1 or 2	1 or 2
ColorMap	Yes	Yes
TileWidth	Yes	No
TileLength	Yes	No
TileOffsets	Yes	No
TileByteCounts	Required if compressed	No
InkSet	1	No
DotRange	Yes, if CMYK	Yes
ExtraSamples	Ignored (except for count)	0

12. Load File Formats

Besides documents that the user creates in Adobe Photoshop (discussed in the previous chapter), there are a number of other files used by Photoshop to store information about colors, brushes, etc. These can be saved to files and loaded into Photoshop for use at a later time or with different images. These are referred to generically as “load files”.

Each load file has a unique file type and file extension associated with it. Photoshop for Macintosh will recognize either, but does not require the use of the extension. Photoshop for Windows will look for the given file extension automatically; this can be overridden.

Many of the files, but not all, have version numbers written as short integers in the first two bytes of the file.

Under the Mac OS, all information is stored in the data forks of Photoshop’s load files. The files are completely interchangable with Windows or any other platform.

Note: Consistent byte ordering is required across platforms when reading and writing load files. Photoshop stores multi–byte values with the high–order bytes first, big–endian, like on 680x0 systems with the Mac OS. This is opposite of the way it is done on Intel systems with Windows. For more information, see “Macintosh and Windows development” in chapter 2.

Arbitrary Map

Arbitrary Map files are loaded and saved in Photoshop's "Curves" dialog.

Table 12-1: Arbitrary map file types

OS	Filetype/extension
Mac OS	8BLT
Windows	.AMP

There is no version number written in the file, and the file must be an even multiple of 256 bytes long. Each 256 bytes is a lookup table, where:

1. The first byte of the table corresponds to byte zero of the image.
2. The last byte of the table corresponds to byte 255 of the image.
3. A NULL table that has no effect on an image is a linear table of bytes from 0 to 255.

If the file has one table, it is applied to the image's channels according to these priorities:

1. If the image has a master composite channel, the table is applied to it. If not, then:
2. If the image has a single active channel, the table is applied to it. If not, then:
3. If the image has no composite channel and more than one active channel, the table is not applied.

If the file has exactly three tables, it is applied to the image's channels according to these priorities:

1. The tables are assumped to represent RGB lookups. They are applied to the first three channels in the image, leaving the master composite untouched. Or:
2. If the image has a single active channel, the tables are converted to grayscale and the result is applied to the active channel. Or:
3. The first table is treated as a master. The remaining tables are applied to the image channels in turn (second table is applied to first channel, third table is applied to second channel, etc.).

Single active channels

Photoshop handles single active channels in a special fashion. When saving a map applied to a single channel, only one table is written to the file. Similarly, when reading a file for application to a single active channel, the master table is the one that will be used on that channel. This allows easy application of a single file to both composite and grayscale images.

Brushes

Brushes settings files are loaded and saved in Photoshop's "Brushes" palette. These are typically stored in the Goodies/Brushes & Patterns sub–folder in the Mac OS, or the Brushes sub–directory in Windows.

Table 12-2: Brushes file types

OS	Filetype/extension
Mac OS	8BBR
Windows	.ABR

Table 12-3: Brushes file format

Length	Name	Description
2 bytes	version	=1. Short integer.
2 bytes	count	A short integer indicating how many brushes are in the remainder of the file.
Variable	brushes	Two types of brushes are currently supported: elliptical, computed brushes and sampled brushes. Computed brushes are created with the "New Brush" command; sampled brushes are created from selected image data using the "Define Brush" command. See table 12-4.

Table 12-4: Brush components

Length	Name	Description
2 bytes	type	A short integer indicating the type of brush. 1=Computed brush; 2=Sampled brush. Other values are currently undefined.
4 bytes	size	A long integer indicating the number of bytes in the remainder of the brush definition. Photoshop uses this information to skip over brush types that it doesn't understand.
size bytes	data	The contents depend on the type of brush. Computed brush data is always 14 bytes; sampled brush data varies in size depending on the image data that makes up the brush tip.

Table 12-5: Computed brush parameters

Length	Name	Description
4 bytes	miscellaneous	Long integer. Ignored.
2 bytes	spacing	Short integer from 0...999 where 0=no spacing.
2 bytes	diameter	Short integer from 1...999.
2 bytes	roundness	Short integer from 0...100.
2 bytes	angle	Short integer from -180...180.
2 bytes	hardness	Short integer from 0...100.

Table 12-6: Sampled brush parameters

Length	Name	Description
4 bytes	miscellaneous	Long. Ignored.
2 bytes	spacing	Short integer from 0...999 where 0=no spacing.
1 byte	anti-aliasing	0=no anti-aliasing when applied; 1=anti-alias when applied. Brushes with sampled data taller or wider than 32 pixels will never be anti-aliased.

Table 12-6: Sampled brush parameters (Continued)

Length	Name	Description
8 bytes	bounds	Rectangle: Four short integers giving the bounds of the sampled data in the order top, left, bottom, right.
16 bytes	bounds-long	Rectangle, same as Bounds, but in four long integers.
2 bytes	depth	Depth of the sample data. Always 8.
Variable	image data	If the bounds are taller than 16384 pixels, the data is broken into 16384-line chunks. Each chunk is streamed as shown in table 12-7.

Table 12-7: Sampled brush image data structure

Length	Name	Description
2 bytes	compression	0=Raw data, 1=RLE compressed.
Variable	data	The brush tip image data is a single plane of grayscale data, stored in scanline order, with no pad bytes. If compression=0, the data is just the raw image data. If compression=1, the data starts with the byte counts for all the scan lines (equal to the number of rows, as described by the bounds), with each count stored as a two–byte value. The RLE compressed data follows, with each scan line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine PackBits, and the TIFF standard.

Color Table

Color Table files are loaded and saved in Photoshop's "Color Table" dialog (used with Indexed Color images), and can also be loaded into the "Colors" palette.

Table 12-8: Color table file types

OS	Filetype/extension
Mac OS	8BCT
Windows	.ACT

There is no version number written in the file. The file is exactly 768 bytes long, and contains 256 RGB colors:

1. The first color in the table is index zero.

2. There are three bytes per color in the order Red, Green, Blue.

If loaded into the "Colors" palette, the colors will be installed in the color swatch list as RGB colors.

Colors

Colors files are loaded and saved in Photoshop's "Colors" palette. These are typically stored in the `Palettes` sub–directory in Windows.

Table 12-9: Colors file types

OS	Filetype/extension
Mac OS	8BCO
Windows	.ACO

Table 12-10: Colors file format

Length	Name	Description
2 bytes	version	=1. Short integer.
2 bytes	count	Short integer indicating how many colors are in the file.
`count*10` bytes	colors	Each color is 10 bytes composed of a `space` byte and `data`. Described in table 12-11.

Table 12-11: Color structure

Length	Name	Description
2 bytes	space	A short integer indicated the color space the color belongs to as shown in table 12-12.
8 bytes	data	Four short unsigned integers with the actual color data. if the color does not require four values to specify, the extra values are undefined and should be written as zeros. See table 12-12.

Table 12-12: Color space IDs

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 `HSV-Color` 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.

Photoshop allows the specification of custom colors, such as those colors that are defined in a set of custom inks provided by a printing ink manufacturer. These colors can be stored in the "Colors" palette and streamed to and from load files. The details of a custom color's color data fields are not public and should be treated as a black box.

Table 12-13 gives the color space IDs currently defined by Photoshop for some custom color spaces.

Table 12-13: Custom color spaces

Color ID	Name
3	Pantone matching system
4	Focoltone colour system
5	Trumatch color
6	Toyo 88 colorfinder 1050
10	HKS colors (European Photoshop only)

Command Settings File

Commands settings files are loaded and saved in Photoshop 3.0's "Commands" palette. This feature supplants the Function Key feature of Photoshop 2.5. The Commands palette buttons are simple mappings to Photoshop menu items, with optional function key shortcut and colorization.

Note: The Commands palette does not exist in Photoshop 4.0. Its functionality has been absorbed into the "Actions" palette. This section is provided for backwards compatibility and reference only.

Table 12-14: Command settings file types

OS	Filetype/extension
Mac OS	8BFK
Windows	.ACM

Table 12-15: Command settings file format

Length	Name	Description
2 bytes	version	=2. Short integer.
2 bytes	count	Number of command records in the file. There are no pad bytes between records.
Variable	records	Command records, one after the other. Described in table 12-16.

Table 12-16: Command record structure

Length	Name	Description
4 bytes	command ID	Command ID. Must be zero. Obsolete.
2 bytes	function key ID	Integer from –15...15. Positive numbers map directly onto the numbered function keys (F1, F2, etc.). Negative numbers indicate that the shift key must be used for the shortcut (Shift–F1, Shift–F2, etc.). Zero means the button has no keyboard shortcut. On Windows systems, values outside of –12 to 12 will be ignored as standard Windows systems have 12 function keys on the keyboard. Windows systems will also map 1 to 0, as the F1 key is reserved for Help. These numbers should be unique across all entries in a Commands file. Photoshop will ignore duplicates.
2 bytes	color index	Each command button can be assigned a color with which its background will be tinted when drawn. There are eight predefined colors: 0=None; 1=Red; 2=Orange; 3=Yellow; 4=Green; 5=Blue; 6=Purple; 7=Gray.

Table 12-16: Command record structure

Length	Name	Description
1 byte	title matching flag	Boolean flag indicating button title updating off/on. For example, a button assigned to the “Layers” palette would change text from “Show Layers” to “Hide Layers” automatically as the state of the palette and the actual menu item changes. 0=Don’t update. Button title has been changed from the menu item text by the user and shouldn’t change unless by user. 1=Update. Button title should automatically be updated to match the command’s current menu item text.
Variable	button title	Pascal-style string, with no pad bytes. This is the title of the button that will be drawn on the Command palette. Usually matches menu item text.
Variable	command key	Pascal–style string, with no pad bytes. This is the key for finding the menu item in Photoshop’s menus. To distinguish menu items which could be duplicated on different menus, a key may include the title of the menu itself followed by a colon (“Mode:RGB Color”). This text is displayed in the options dialog for the button, but not on the Commands palette itself. Even if `TitleMatching=1`, the button text never contains the menu title qualifier.

Curves

Curves settings files are loaded and saved in Photoshop's "Curves" dialog and "Black Generation" curve dialog (from within Separation Setup Preferences). Curves files can also be loaded into any of Photoshop's transfer function dialogs, such as the Duotone Curve dialog from within Duotone Options.

Note: When loaded into a transfer function dialog, only the first curve in a Curves file is used.

Table 12-17: Curves file types

OS	Filetype/extension
Mac OS	8BSC
Windows	.CRV

Table 12-18: Curves file format

Length	Name	Description
2 bytes	version	=1. Short integer.
2 bytes	count	Short integer indicating how many curves are in the file.
Variable	curves	Curves data, one after the other. Described in table 12-19.

Table 12-19: Curves data structure

Length	Name	Description
2 bytes	point count	Short integer from 2...19 indicating how many points are in the curve.
point count * 4 bytes	curve points	Each curve point is a pair of short integers where the first number is the output value (vertical coordinate on the Curves dialog graph) and the second is the input value. All coordinates have range 0 to 255. See also *Null curves*, below.

Null curves

A NULL curve (no change to image data) is represented by the following five–number, ten–byte sequence in a file:

```
2 0 0 255 255.
```

Displaying ink percentages

Photoshop allows the option of displaying ink percentages instead of pixel values; this is a display option only and the internal data is unchanged, with 100% ink equal to image data of 0 and 0% ink equal to image data of 255.

Curves data order

1. The first curve is a master curve that applies to all the composite channels (RGB) when in composite image mode.

2. The remaining curves apply to the active channels in order: curve two applies to channel one, curve three applies to channel two, etc., up until curve 17, which applies to channel 16.

Indexed color

The exception to the normal order, and the reason there are up to 19 curves, is when the mode is Indexed color. In this case:

1. The first curve is a master curve.

2. The next three curves are created for the Red, Green, and Blue portions of the image's color table, and they are applied to the first channel.

3. The remaining curves apply to any remaining alpha channel that is active: for instance, if channel two is active, curve five applies to it; if channel three is active, curve six applies to it, etc., up until curve 19, which applies to channel 16.

Single active channels
Photoshop handles single active channels in a special fashion. When saving the curves applied to a single channel, the settings are stored into the master curve, at the beginning of the file. Similarly, when reading a curves file for application to a single active channel, the master curve is the one that will be used on that channel. This allows easy application of a single file to both RGB and grayscale images.

Photoshop 3.0 and later Curves files and Photoshop 2.0
Photoshop 3.0 and later can write Curves files that Photoshop 2.0 will not be able to read, because version 3.0 and later of Photoshop's active channel support is different from Photoshop 2.0's. There could be more active channels in a Curves dialog than 2.0 supported.

For maximum compatability with version 2.0, Photoshop 3.0 and later will pad the file with NULL curves to always write at least five curves. The presence of extraneous NULL curves will not affect a load operation.

Photoshop 3.0 and later Curves files and Photoshop 2.5
It is possible, however rare, to create a Curves load file with Photoshop 3.0 or later that cannot be read by Photoshop 2.5. Version 3.0 and later of Photoshop allows a maximum of 24 channels per document, Photoshop 2.5 allows 16.

Duotone options

Duotone settings files are loaded and saved in Photoshop's "Duotone Options" dialog.

Table 12-20: Duotone file types

OS	Filetype/extension
Mac OS	8BDT
Windows	.ADO

Table 12-21: Duotone file format

Length	Name	Description
2 bytes	version	=1. Short integer.
2 bytes	count	Short integer from 1...4 indicating how many plates are in the duotone spec. 1=Monotone; 2=Duotone; 3=Tritone; 4=Quadtone.
4*10 bytes	ink colors	Four ink colors, regardless of the number of plates. The contents of the colors beyond the last plate specified by Count are undefined. Each color is 10 bytes and described in table 12-22. It is identical to the format in a Colors load file.
4*64 bytes	ink names	Four ink names, regardless of the number of plates. Each name is streamed as a Pascal-style string with a length byte followed by the string name. Names may not be more than 63 characters. Each name is padded to occupy 64 bytes, including the length byte. Any names beyond the last plate specified by Count should be empty, size=0.
4*28 bytes	ink curves	Four ink curves, regardless of the number of plates. Described in table 12-24.
2 bytes	dot gain	=20. Short integer. Kept for compatability with Photoshop 2.0. Ignored.
11*10 bytes	overprint colors	Eleven ink colors, regardless of the number of plates. The number of defined overprints depends on Count. Monotones=no overprint colors. Duotones=one overprint color. Tritones=four overprint colors. Quadtones=11 overprint colors. The contents of the colors beyond the last defined overprint are undefined. Each color is 10 bytes and described in table 12-22. It is identical to the format in a Colors load file.

Table 12-22: Duotone color structure

Length	Name	Description
2 bytes	space	A short integer indicated the color space the color belongs to as shown in table 12-23.
8 bytes	data	Four short unsigned integers with the actual color data. if the color does not require four values to specify, the extra values are undefined and should be written as zeros. See table 12-23.

Table 12-23: Duotone color space IDs

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.

Table 12-24: Ink curves structure

Length	Name	Description
26 bytes	transfer curve	Array of 13 short integers from 0...1000 representing 0.0...100.0. All but the first and last value may be –1, representing no point on the curve. See *Null transfer curve* below.
2 bytes	override	=0. Short integer for compatibility. Ignored by Photoshop 3.0.

Null transfer curve

Any curves beyond the last plate specified by `Count` should be equal to the `NULL` curve. A `NULL` transfer curve looks like this:

```
0, –1, –1, –1, –1, –1, –1, –1, –1, –1, –1, –1, 1000.
```

Halftone screens

Halftone Screens settings files are loaded and saved in Photoshop's Halftone Screens dialog (from within Page Setup).

Table 12-25: Halftone screen file types

OS	Filetype/extension
Mac OS	8BHS
Windows	.AHS

Table 12-26: Halftone screens file format

Length	Name	Description
2 bytes	version	=5. Short integer.
4*18 bytes	screens	Four screen descriptions. See table 12-27.
variable (see description)	curves	For every screen which has a custom spot function, the text of the PostScript function is written here. The functions are written one after the other with no header information, in the same order as the screen settings. The size of each custom spot is the absolute value of its negative shape code.

Table 12-27: Halftone screen parameter structure

Length	Name	Description
4 bytes	frequency value	Ink's screen frequency, in lines per inch. Binary fixed point value with 16 bits representing the integer and fractional parts from 1.0...999.999.
2 bytes	frequency scale	Units for the screen frequency. Lines per inch = 1; lines per centimeter = 2. Only affects display, not `Frequency Value`.
4 bytes	angle	Angle for screen. Binary fixed point value with 16 bits representing the integer and fractional parts from -180.0000 ... 180.0000, measured in degrees.
2 bytes	shape code	Code representing the shape of the halftone dots. 0=Round; 1=Ellipse; 2=Line; 3=Square; 4=Cross; 6=Diamond. Custom shapres are represented by a negative number. The absolute value of the negative number is the size in bytes of the custom spot function described in table 12-26.
4 bytes	miscellaneous	=0. Not currently used by Photoshop.
1 byte	accurate screens	Boolean. 1=Use accurate screens; 0=Use other.
1 byte	default screens	Boolean. 1=Use printer's default screens; 0=Use other.

Hue/Saturation

Hue/Saturation settings files are loaded and saved in Photoshop's "Hue/Saturation" dialog.

Table 12-28: Hue/saturation file types

OS	Filetype/extension
Mac OS	8BHA
Windows	.HSS

Table 12-29: Hue/saturation file format

Length	Name	Description
2 bytes	version	=1. Short integer.
1 byte	mode	Boolean. 0=Use settings for hue-adjustment; 1=Use settings for colorization.
1 byte	padding	Padding byte must be present but is ignored by Photoshop.
6 bytes	colorization	Three short integers Hue, Saturation, and Lightness from –100...100. The user interface represents hue as –180...180, as the traditional HSB color wheel, with red=0.
42 bytes	hue-saturation settings	Three sets of seven short integers, from –100...100. Described in table 12-30.

Table 12-30: Hue-saturation settings

Length	Name	Description
14 bytes	hue settings	Seven short integers. The first value is the master hue change, followed by six values. For RGB and CMYK, those values apply to each of the six hextants in the HSB color wheel: those image pixels nearest to red, yellow, green, cyan, blue, or magenta. These numbers appear in the user interface from –60...60, however the slider will reflect each of the possible 201 values from –100...100. For Lab, the first four of the six values are applied to image pixels in the four Lab color quadrants, yellow, green, blue, and magenta. The other two values are ignored (=0). The values appear in the user interface from –90 to 90.
14 bytes	saturation settings	Seven short integers. The first is a master saturation value. The other six are applied to pixels exactly the same as the hue settings.
14 bytes	lightness settings	Seven short integers. The first is a master lightness value. The other six are applied to pixels exactly the same as the hue settings.

Ink colors setup

Ink Colors settings files are loaded and saved in Photoshop 3.0's "Ink Colors Setup" dialog, via the "Preferences" sub-menu.

Table 12-31: Ink colors file types

OS	Filetype/extension
Mac OS	8BIC
Windows	.API

Table 12-32: Ink colors setup file format

Length	Name	Description
2 bytes	version	=4. Short integer.
27*2 bytes	ink colors	Nine sets of three short integers specifying the xyY (CIE) values for the inks and their combinations. The inks are specified in the order cyan, magenta, yellow, magenta–yellow (red), cyan–yellow (green), cyan–magenta (blue), cyan–magenta–yellow, followed by the white and black points. Each set is written in the order xyY where: x=0...10000, representing 0.0...1.0000. y=1...10000, representing 0.0001...1.0000. Y=0...20000, representing 0.00...200.00.
4*2 bytes	gray balance	Four short integers from 50...200, representing 0.5 to 2.00. Specifies the gray color balance for CMYK.
2 bytes	dot gain	Short integer from -10...40, representing –10%...40%.

Custom kernel

Kernel settings files are loaded and saved in Photoshop's "Custom filter" dialog.

Table 12-33: Custom kernel file types

OS	Filetype/extension
Mac OS	8BCK
Windows	.ACF

Format:

1. There is no version number written in the file.
2. The file is expected to be exactly 54 bytes long, representing 27 short integers, described in table 12-34.

Table 12-34: Custom filter structure

Length	Name	Description
50 bytes	weights	The first 25 values are the custom weights from –999...999, applied to pixels offset from each pixel by [-2,-2] to [2,2]. The values progress through horizontal offsets first. See *Weight offset progression* below.
27*2 bytes	ink colors	Nine sets of three short integers specifying the xyY (CIE) values for the inks and their combinations. The inks are specified in the order cyan, magenta, yellow, magenta–yellow (red), cyan–yellow (green), cyan–magenta (blue), cyan–magenta–yellow, followed by the white and black points. Each set is written in the order xyY where: x=0...10000, representing 0.0...1.0000. y=1...10000, representing 0.0001...1.0000. Y=0...20000, representing 0.00...200.00.
2 bytes	scale	Short integer from 1...9999.
2 bytes	offset	Short integer from –9999...9999.

Weight offset progression
This is sample matrix for the weight offset progression:

```
{[-2,-2],[-1,-2],[ 0,-2],[ 1,-2],[ 2,-2],
 [-2,-1],[-1,-1],[ 0,-1],[ 1,-1],[ 2,-1],
 [-2, 0],[-1, 0],[ 0, 0],[ 1, 0],[ 2, 0],
 [-2, 1],[-1, 1],[ 0, 1],[ 1, 1],[ 2, 1],
 [-2, 2],[-1, 2],[ 0, 2],[ 1, 2],[ 2, 2]}
```

Levels

Levels settings files are loaded and saved in Photoshop's "Levels" dialog.

Table 12-35: Levels file types

OS	Filetype/extension
Mac OS	8BLS
Windows	.ALV

Table 12-36: Levels file format

Length	Name	Description
2 bytes	version	=2. Short integer.
290 bytes	levels records	29 sets of levels, each level contains 10 bytes of five short integers. Described in table 12-37.

Table 12-37: Level record structure

Length	Name	Description
2 bytes	input floor	Short integer from 0...253.
2 bytes	input ceiling	Short integer from 2...255.
2 bytes	output floor	Short integer from 0...255. Matched to input floor.
2 bytes	output ceiling	Short integer from 0...255.
2 bytes	gamma	Short integer from 10...999 representing 0.1...9.99. Applied to all image data.

Level record sets order

1. The first set of levels is the master set that applies to all of the composite channels (RGB) when in composite image mode.

2. The remaining sets apply to the active channels individually; set two applies to channel one, the set three to channel two, etc., up until set 25, which applies to channel 24.

3. Sets 28 and 29 are reserved and should be set to zeros.

Indexed color

The exception to the normal order is when the mode is Indexed:

1. The first set is a master set.

2. The next three sets are created for the Red, Green, and Blue portions of the image's color table, and they are applied to the first channel.

3. The remaining sets apply to any remaining alpha channels that are active: for instance, if channel two is active, set five applies to it; if channel three is active, set six applies to it, etc., up until channel 27, which applies to channel 24.

4. Sets 28 and 29 are reserved and should be set to zeros.

Single active channels

Photoshop handles single active channels in a special fashion. When saving the levels applied to a single channel, the settings are stored into the master set, at the beginning of the file. Similarly, when reading a levels file for application to a single active channel, the master levels are the ones that will be used on that channel. This allows easy application of a single file to both RGB and grayscale images.

Photoshop 3.0 and later Levels files and Photoshop 2.5

There are two versions of the Levels file format. Photoshop 3.0 and later reads both but only writes version 2. Because the maximum number of channels was increased in Photoshop 3.0 from 16 to 24, Photoshop 3.0 and later actually writes a longer Levels file than Photoshop 2.5. Photoshop 2.5 is still capable of reading these files and ignores the extra data.

Monitor setup

Monitor settings files are loaded and saved in Photoshop's "Monitor Setup" dialog, via the "Preferences" sub-menu in Photoshop 3.0, and under the **File** menu in Photoshop 4.0.

Table 12-38: Monitor setup file types

OS	Filetype/extension
Mac OS	8BMS
Windows	.AMS

Table 12-39: Monitor setup file format

Length	Name	Description
2 bytes	version	=2. Short integer.
2 bytes	gamma	Short integer from 75...300 representing 0.75...3.00.
2*2 bytes	white point	Two short integers as CIE chromaticity coordinates: x, y. x=0...10000 representing 0.0...1.0000. y=1...10000 representing 0.0001...1.0000.
6*2 bytes	phosphors	Three sets of two integers giving x, y coordinates of the red, green, and blue phosphors. x=0...10000 representing 0.0...1.0000. y=1...10000 representing 0.0001...1.0000. In the order *red* x, *red* y; *green* x, *green* y; *blue* x, *blue* y.

Replace color/Color range

Replace Color settings files are loaded and saved in Photoshop's "Replace Color" dialog. They are also used to load and save settings from the "Color Range" dialog.

Table 12-40: Replace color/Color range file types

OS	Filetype/extension
Mac OS	8BXT
Windows	.AXT

Table 12-41: Replace color/Color range file format

Length	Name	Description
2 bytes	version	=1. Short integer.
2 bytes	color space	Short integer indicating what space the color components are in. 7=Lab color, 8=grayscale. No other values are supported.
6 bytes	component ranges	Six unsigned byte values representing the range of colors within which a pixel's color must fall to be considered selected for color replacement, or color range selecting. Described in table 12-42.
2 bytes	fuzziness	Short integer from 0...200 controlling how colors close to selected colors are affected.
6 bytes	transform settings	When used with Replace Color: Three short integers from –100...100. Described in table 12-43. When used with Color Range: Writes zeros into the three short integers and ignores.

Table 12-42: Component range structure

Length	Name	Description
1 byte	low endpoint 1	if Lab (color space=7): low endpoint of L value if grayscale (color space=8): low endpoint of gray range
1 byte	high endpoint 1	if Lab: high endpoint of L value if grayscale: 0
1 byte	low endpoint 2	if Lab: low endpoint of a chromanance value if grayscale: 0
1 byte	high endpoint 2	if Lab: high endpoint of a chromanance value if grayscale: 0
1 byte	low endpoint 3	if Lab: low endpoint of b chromanance value if grayscale: low endpoint of gray range
1 byte	high endpoint 3	if Lab: high endpoint of b chromanance value if grayscale: high endpoint of gray range

Table 12-43: Replace color transform settings

Length	Name	Description
2 bytes	hue	Short integer from –100...100. Hue change.
2 bytes	saturation	Short integer from –100...100. Saturation change.
2 bytes	lightness	Short integer from –100...100. Lightness change.

Scratch Area

Scratch Area settings files are loaded and saved in Photoshop's Scratch palette.

Table 12-44: Scratch area file types

OS	Filetype/extension
Mac OS	8BSR
Windows	.ASR

Table 12-45: Scratch area file format

Length	Name	Description
2 bytes	version	=1. Short integer.
Variable	data	Scratch area data in the form of RGB image data. The three planes are written one after the other in the order *red*, *green*, *blue*. Described in table 12-46.

Table 12-46: Scratch area data structure

Length	Name	Description
16 bytes	bounds	Four long integers giving the bounds of the scratch data rectangle in the order *top*, *left*, *bottom*, *right*. Photoshop 3.0 has a fixed Scratch palette size and this will always be `[0,0,89,200]`
2 bytes	depth	Depth of the current data plane. Always 8.
Variable	image data	Image data. Described in table 12-47.

Table 12-47: Scratch area image data structure

Length	Name	Description
2 bytes	compression	0=Raw data, 1=RLE compressed.
Variable	data	Each plane of the image data is stored in scanline order, with no pad bytes. If `compression=0`, the data = raw data. If `compression=1`, the data starts with the byte counts for all the scan lines (equal to the number of rows, as described by the bounds), with each count stored as a two–byte value. The RLE compressed data follows, with each scan line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine `PackBits`, and the TIFF standard.

Selective color

Selective Color settings files are loaded and saved in Photoshop's "Selective Color" dialog.

Table 12-48: Selective color file types

OS	Filetype/extension
Mac OS	8BSV
Windows	.ASV

Table 12-49: Selective color file format

Length	Name	Description
2 bytes	version	=1. Short integer.
2 bytes	correction method	Short integer. 0=Apply color correction in relative mode; 1=Apply color correction in absolute mode.
80 bytes	plate corrections	Ten eight-byte correction records, described in table 931.

Table 12-50: Plate correction structure

Length	Name	Description
2 bytes	cyan	Short integer from –100...100. Amount of cyan correction.
2 bytes	magenta	Short integer from –100...100. Amount of magenta correction.
2 bytes	yellow	Short integer from –100...100. Amount of yellow correction.
2 bytes	black	Short integer from –100...100. Amount of black correction.

Record order

1. The first record is ignored by Photoshop 3.0 and is reserved for future use. It should be set to all zeroes.

2...10. The rest of the records apply to specific areas of colors or lightness values in the image, in the following order: reds, yellows, greens, cyans, blues, magentas, whites, neutrals, blacks.

Separation setup

Separation settings files are loaded and saved in Photoshop's "Separation Setup" dialog, via the "Preferences" sub-menu.

Table 12-51: Separation file types

OS	Filetype/extension
Mac OS	8BSS
Windows	.ASP

Table 12-52: Separation file format

Length	Name	Description
2 bytes	version	=300. Short integer.
2 bytes	separation type	Boolean. 0=UCR separations; 1=GCR separations.
2 bytes	black limit	Short integer from 0...100 giving the black ink limit.
2 bytes	total limit	Short integer from 200...400 giving the total ink limit.
2 bytes	UCA amount	Short integer from 0...100 giving the undercolor addition for GCR separations.
Variable	black generation curve	Spline curve detailed in table 12-53. Identical to the Curves data format in table 12-19.

Table 12-53: Black generation curve data structure

Length	Name	Description
2 bytes	point count	Short integer from 2...19 indicating how many points are in the curve.
2* point count bytes	curve points	Each curve point is a pair of short integers where the first number is the output value (vertical coordinate on the Black Generation dialog graph) and the second is the input value. All coordinates have range 0 to 255. See *Null curves*.

Null curves

A NULL curve (no change to image data) is represented by the following five–number, ten–byte sequence in a file:

```
2 0 0 255 255.
```

Note:
The `black generation curve` and the `UCA limit` must both be present even if the `separation type` is set to UCR (=0).

Separation tables

Separation Table files are loaded and saved in Photoshop's "Separation Tables" dialog.

Table 12-54: Separation tables file types

OS	Filetype/extension
Mac OS	8BST
Windows	.AST

Format:

1. If the size of the file is 33 * 33 * 33 * 4, then the file consists only of an Lab–>CMYK table as currently documented.

2. If the size of the file is 33 * 33 * 33 + 256 * 3, then the file consists only of a CMYK–>Lab table as currently documented.

3. Otherwise, the file has the format listed in table 12-55.

Table 12-55: Separation table file format

Length	Name	Description
2 bytes	version	=300. Short integer.
1 byte	has Lab to CMYK	Boolean. 0=No; 1=Contains Lab–>CMYK table.
1 byte	has CMYK to Lab	Boolean. 0=No; 1=Contains CMYK–>Lab table.
33*33*33*4 bytes	Lab to CMYK table	If `hasLabtoCMYK=1` then this section contains CMYK colors for 33*33*33 Lab colors. The CMYK colors are written in interleaved order, one byte each ink. 0=100%, 255=0%. See *Generating Lab source colors* below.
(33*33*33 +256)*3 bytes	CMYK to Lab table	If hasCMYKtoLab=1 then this section contains Lab colors for 33*33*33+256 CMYK colors. The Lab colors are written in interleaved order, one byte per component. See *Generating CMYK source colors* below.
1 byte	has gamut table	Boolean. 0=No; 1=gamut table follows.
1 byte	filler	If `hasGamutTable=0` then this byte will not be present. If `hasGamutTable=1` then this byte should be set to 1 for compatibility.
(((33*33*33L)+7)>>3) bytes	gamut table	If `hasGamutTable=0` then this field will not be present. If `hasGamutTable=1` then this is the gamut table. The gamut table is a bit table indexed in the same way as the `Lab->CMYK` table with the high bit of the first byte at index 0. See *Testing for bits in the gamut table*, below.

Generating Lab source colors

The Lab colors that are the source colors can be generated from the Lab->CMYK table with the following routine:

```
for (i = 0; i< 33; i++)
    for (j = 0; j < 33; j++)
        for (n = 0; n < 33; n++)
        {
            L = Min (i * 8, 255);
            a = Min (j * 8, 255);
            b = Min (n * 8, 255);
        }
```

Generating CMYK source colors

The CMYK colors that are the source colors can be generated from the CMYK->Lab table with the following routine:

```
for (i = 0; i< 33; i++)
    for (j = 0; j < 33; j++)
        for (n = 0; n < 33; n++)
        {
            c = Min (i * 8, 255);
            m = Min (j * 8, 255);
            y = Min (n * 8, 255);
            k = 255;
        }

for (i = 0; i < 256; i++)
{
    c = 255;
    m = 255;
    y = 255;
    k = i;
}
```

Testing for bits in the gamut table

To test the bit at bitIndex, use table:

```
[bitIndex >> 3] & (0x0080 >> (bitIndex & 0x07))) != 0.
```

bitIndex itself is calculated in the same way you would calculate an index into the Lab->CMYK table.

A result of 1 indicates that the color is in gamut and 0 indicates that it is out of gamut.

Transfer function

Transfer Function settings files are loaded and saved in Photoshop's "Duotone Curve" dialog from within "Duotone Options" and "Transfer Function" dialogs from within **Page Setup**. Transfer Function files can also be loaded into any of Photoshop's curves dialogs, such as the Curves color adjustment dialog.

Table 12-56: Transfer function file types

OS	Filetype/extension
Mac OS	8BTF
Windows	.ATF

Table 12-57: Transfer function file format

Length	Name	Description
2 bytes	version	=4. Short integer.
112 bytes	functions	There are four transfer functions in the file, described in table 12-58.

Table 12-58: Transfer function structure

Length	Name	Description
26 bytes	curve	Array of 13 short integers from 0...1000 representing 0.0...100.0. All but the first and last value may be –1, representing no point on the curve. See *Null transfer curve* below.
2 bytes	override	Boolean. 0=Let printer supply curve; 1=Override printer's default transfer curve.

Null transfer curve

Any curves beyond the last plate specified by Count should be equal to the NULL curve. A NULL transfer curve looks like this:

```
0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1000.
```

Note: The file always contains four functions. When writing the printer transfer functions for grayscale images, for instance, Photoshop writes four copies of the single transfer function specified in the user interface.

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 IndexedColorMode, 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 *Color space ID*, table 12-12.

B. OLE Automation

by Robert A. Swirsky, Photoshop Engineering

Adobe Photoshop 4.0 supports OLE automation. With an *OLE automation controller*, like Microsoft's Visual Basic, Visual Basic for Applications, or Borland's Delphi, Adobe Photoshop 4.0 can open and close documents and execute Action scripts.

OLE automation is only available on Windows 95 and Windows NT platforms. It is not available on Windows 3.1 or Macintosh. A similar external automation mechanism exists on the Macintosh using AppleScript.

Automation basics

Photoshop 4.0 implements the “Actions” palette, permitting a user to record a sequence of actions and play them back. See the chapter on Scripting for more information.

The actions in the actions palette are exposed via OLE Automation. Once an Action has been recorded, it can be played back using OLE Automation in addition to interactively by pressing the *play* button in the Actions palette.

Automation objects

Several Adobe Photoshop *automation objects* can be instantiated from an OLE automation controller. By accessing properties and methods associated with different objects, you can make Photoshop open, close, and save documents, as well as run pre-recorded scripts. The Automation Objects are:

1. Application
2. Document
3. Actions Collection
4. Action

Application objects

Use an *application object* to start or quit the host; create a document object, or run a script by name.

Table B-1: Application object attributes

Name	Type	Parameters	Description
Actions	Property	n/a	Returns an Actions Collection, which contains all the actions int he currently loaded Actions palette.
FullName	Property	n/a	The full name of the application.
Open	Method	BSTR	Opens a new document and returns a document object.
PlayAction	Method	BSTR	Plays an action by name on the current document.
Quit	Method	none	Exits the host.

Document objects

Document objects are instantiated by calling Open from the application object.

Table B-2: Document object attributes

Name	Type	Parameters	Description
Activate	Method	None	Make this document the active document and default target.
Close	Method	None	Save changes and close document.
SaveTo	Method	BSTR	Save the document under a different name.
Title	Property	n/a	The title (filename) of this document.

Actions collection object

The *actions collection object* represents all the scripts currently loaded in the Actions palette. In addition to the attributes in table B-3, it also supports the *For Each* construct in Visual Basic automation controllers.

Table B-3: Actions collection object attributes

Name	Type	Parameters	Description
Count	Method	None	Returns the number of scripts in the Actions palette.
Item	Method	Integer	Returns a particular action object.

Action objects

Action objects are the individual scripts in the Actions palette.

Table B-4: Action object attributes

Name	Type	Parameters	Description
Name	Property	n/a	The name (title) of this script.
Play	Method	None	Play this script.

Creating OLE Automation with Visual Basic

This section contains programming examples that show how to use Microsoft Visual Basic to access the OLE automation objects for Photoshop 4.0.

Creating and destroying an application object

Use Visual Basic's `CreateObject` procedure to instantiate a Photoshop application object. The object can be destroyed with the application object's `Quit` method, or by setting the object to `Nothing`, causing the reference count to decrement to zero.

```
Dim App as Object
Set App = CreateObject("Photoshop.Application")
App.Quit
```

Photoshop's automation class factory is a *single use* object that can only be used by one automation controller at a time. You'll get a message that the Photoshop object can't be created if it is already in use by another application.

Opening and closing documents

The application object's `Open` method creates a new `document` object. it takes a file name (with path) as a parameter and returns a `document` object. Exceptions are raised:

1. If the file can't be opened because it doesn't exist;
2. If the file is in an unrecognized format;
3. If Photoshop is in a modal state and can't process requests at this time.

These exceptions can be caught with Visual Basic's `On Error` statement.

To close a document and save any changes that have been made, use the document object's `Close` method.

```
Dim App as Object
Dim PhotoDoc as Object
Set App = CreateOjbject("Photoshop.Application")
Set PhotoDoc = App.Open("C:\files\photoshop\MyPicture.PSD")
PhotoDoc.Close
App.Quit
```

Running an action script by name

Typically, you'll want to perform an action on the current document by executing a script from the palette. You can run a script by specifying its name, or you can iterate among all the currently loaded scripts and run any or all of them.

To run an action by name, use the PlayAction method from the Application object. Adding to our previous example, we'll run an action called "BlurMe" on the active document. If you have more than one document object instantiated, target one of them by calling its Activate method.

```
Dim App as Object
Dim PhotoDoc as Object
Set App = CreateOjbject("Photoshop.Application")
Set PhotoDoc = App.Open("C:\files\photoshop\MyPicture.PSD")
App.PlayAction("BlurMe")
PhotoDoc.Close
App.Quit
```

PlayAction returns a Boolean value that indicates whether the action was found and played or not. If the action doesn't exist, PlayAction will return FALSE. If the action cannot be played because the host is in a modal state, this method will raise an exception that can be handled with Visual Basic's On Error statement.

Saving under a different name

To save the file under a different name, use the document object's SaveTo method to specify a name.

```
Dim App as Object
Dim PhotoDoc as Object
Set App = CreateOjbject("Photoshop.Application")
Set PhotoDoc = App.Open("C:\files\photoshop\MyPicture.PSD")
App.PlayAction("BlurMe")
PhotoDoc.SaveTo("MyNewPicture.PSD"
PhotoDoc.Close
App.Quit
```

If you don't specify a fully qualified path name, the file will be saved relative to the directory of the original file. Fully qualified path names beginning with a backslash or a drive letter are used as-is. If the file cannot be saved to the specified path, the host will raise a "Can't open file" exception.

Iterating through a collection of actions

The applciation object's `Actions` method returns a collection object that can be used to step through all the action objects currently loaded in the palette. The following example steps through all the available actions, asking the user to run a particular script. The name of an individual action in the collection is obtained through the action object's `Name` method.

If an action's `Play` method cannot play the script, it raises an "Unexpected" exception that can be caught with Visual Basic's `On Error` statement.

```
Dim App as Object
Dim PhotoDoc as Object
Set App = CreateOjbject("Photoshop.Application")
Set PhotoDoc = App.Open("C:\files\photoshop\MyPicture.PSD")
For Each Action in App.Actions
   response = MsgBox(Action.Name, vbYesNo, "Run this Action?")
   if response = vbYes then
      Action.Play
   End If
PhotoDoc.SaveTo("MyNewPicture.PSD"
PhotoDoc.Close
App.Quit
```

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y