Adobe

Adobe Photoshop 3.0.4

Software Development Kit

Revised: 16 July 1995

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.

Version History

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

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 that you are proficient in C language programming and tools. The source code files in this toolkit are written for the Apple MPW compiler and Metrowerks CodeWarrior on the Mac, and Microsoft Visual C++ on Windows and Windows NT.

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, masks, etc. 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.

How to use this guide

This toolkit documentation starts with information that is 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 into Photoshop.

Chapters 4 and 5 discuss PiPL and PiMI resources, which provide the plug–in host with information about plug–in modules.

Chapters 6 through 9 cover four types of plug–in modules (Acquire, Export, Filter, and Format) in detail.

The best way to use this guide is to first read chapters 1 through 5. Then turn to the chapter containing specific information on the type of plug–in you're going to write.

If you are new to writing plug–ins, you should also study and understand the source code to the sample plug–ins. You may choose to use these source files as the starting point for creating your own plug–in module.

Contents of the Photoshop plug–in toolkit

The files included with this toolkit include C language header files PITypes.h, PIGeneral.h, PIAqcuire.h, PIExport.h, PIFilter.h, and PIFormat.h that define the structures and constants you will need to build plug–in modules. The "Examples" directory contains complete source code examples for each plug–in type.

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

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 font family is used throughout the manual.

To print this manual from within Adobe Acrobat Reader, select the “Shrink to Fit” option on the Print dialog.

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 chapters on specific types of plug–in modules.

This chapter also contains information about compiling and testing plug–in modules under Mac OS and Microsoft Windows. For additional compiler–specific information, read 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 in conjunction with Adobe Systems, to extend the standard Adobe Photoshop program. The Adobe Photoshop software includes plug–in modules for importing and exporting images and plug–in filter modules for producing special effects. Plug–in modules can be added or updated independently by end users to customize the application.

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.

Other Adobe applications such as Adobe Premiere support Photoshop plug–in modules. In addition, many applications from third–party developers support the use of Photoshop plug–in modules. (Most plug–in hosts are application programs, but this not a requirement. A Photoshop plug–in host may itself be a plug–in to another application, for example.)

This toolkit and guide are not designed for developers interested in creating plug–in hosts; the emphasis in this guide is clearly on plug–in modules.

Unless otherwise stated, Adobe Photoshop 3.0.4 is assumed to be the plug–in host throughout this manual.

A short history lesson

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

Perhaps the best known example is Apple’s HyperCard, with its support for XCMD’s and XFCN’s. 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’s plug–in implementation was well designed; plug–in modules reside in individual files (rather than having to be pasted into the application using ResEdit), allowing 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 is similar to 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.

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 needed 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 for the WIndows platform, some of the terminology may be unfamiliar.

The other important area where the differences between Macintosh and Windows affect you is byte ordering. Motorola and PowerPC processors store pointers, 16–, and 32– bit numbers in big endian format, while Intel processors user little endian format. 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 2.5 versus version 3.0 plug–in modules

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 support 'PiMI' based plug–in modules. 'PiPL' and 'PiMI' resources are discussed in chapters 4 and 5, respectively.

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.

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, acquire, and export modules. Again, see chapter 3 for details.

Types of plug–in modules

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

This document describes four different types of plug–in modules:

1. **Acquire modules** open an image in a new window. Acquire 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 Acquire sub–menu.

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

3. **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.

4. **Filter modules** modify a selected area of an existing image. These modules appear under the Filter menu. These are the types of plug–ins most Photoshop users are familiar with, and may have very sophisticated features.

In addition, Adobe Photoshop supports two other types of plug–in modules: parser modules and hardware accelerator modules (also known as extension modules). These are beyond the scope of this toolkit and will not be discussed further. They are mentioned here only for completeness. If you need information about these module types, please contact the Adobe Developers Association.

Plug–in module files

Plug–in module files must reside in specific directories for Adobe Photoshop to recognize them. Under Mac OS, plug–in files must be in either the same folder as the Adobe Photoshop application, or in the folder identified in the Photoshop preferences dialog, or in a sub–folder of that folder. Under Windows, plug–in files must be in the directory identified by the PLUGINDIRECTORY profile string in PHOTOSHO.INI.

Usually, a plug–in module file contains a single plug–in. You can create files with multiple plug–in modules, however, in most cases you should not do this since 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 acquisition/export modules, although these frequently are better implemented as a file format module. Another example is a set of closely related filters, since the decrease in user control may be offset by an improvement in the ease with which users can manage their plug–in files.

Plug–in file types and extensions

Plug–in module files should follow the guidelines in table 2–1 for the type identifier under Mac OS, and the file extension under Windows. While these are only recommendations under Adobe Photoshop 3.0, these must be used if your plug–in module runs under earlier versions of Photoshop.

Table 2–1: Plug–in file types and extensions

Plug–in Type	Macintosh File Type	Windows File Extension
General (any type of plug–in)	8BPI	.8BP
Acquire modules	8BAM	.8BA
Export modules	8BEM	.8BE
Filter plug–ins	8BFM	.8BF
File Format plug–ins	8BIF	.8BI
Accelerator Extensions	8BXM	.8BX
Parser plug–ins	8BYM	.8BY

On the Macintosh, your plug–in should use the same creator ID as Adobe Photoshop ('8BIM') if you wish to use the standard plug–in icons defined in Photoshop.

Basic data types

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

Table 2–2: Basic data types

Name	Description
int8, int16, int32, unsigned8, unsigned16, unsigned32	These are 8, 16 and 32 bit integers respectively.
Boolean	Boolean flags are stored in a single byte, 0=FALSE, any other value=TRUE.
OSType	Same representation as an int32 but typically denotes a Macintosh style 4 character code like 'PiPL'.
TypeCreatorPair	A structure of two OSTypes denoting a file type and creator code. The type code is the first field of the structure and the creator code is second.
FlagSet	This is an array of boolean values where the first boolean is contained in the high order bit of the first byte. The eighth entry would be in the high–order bit of the second byte, etc.
PString	A Pascal style string where the first byte gives the length of the string and the content bytes follow.
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	These are like Macintosh 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
```

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 plug–in module type.

Within your main function, you will typically have a switch statement that dispatches the pluginParamBlock, pluginData, and result parameters to different handlers for each selector that your plug–in module responds to. The example plug–in modules show one style of dispatching to selector handlers.

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 exact definition of this parameter block fields depends on the type of plug–in module. Refer to chapters 6 through 9 for descriptions of each plug–in type's parameter block.

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

One standard use for this field is to store a pointer or 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.

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

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

Returning a negative value also means that your plug–in module encountered an error, but that the plug–in host should display its standard error dialog describing the error.

Table 2–3 shows the error code ranges used by the different types of plug–in modules. Refer to chapters 6 through 9 for more details.

Table 2–3: Error code ranges

Type of module	Error Range
Acquire	–30000 to –30099
Filter	–30100 to –30199
Export	–30200 to –30299
Format	–30500 to –30599
General Errors	–30900 to –30999

Mac OS and Windows operating system error codes may be returned to the plug–in host also. In PITypes.h, several common Mac OS error codes are defined for use under Windows, simplifying programming for both Mac OS and Windows.

About boxes

All plug–in modules should respond to a selector value of zero, which means display an about box. You have complete freedom to display any kind of about box you wish, but to fit in smoothly with the Adobe Photoshop interface you should obey the following 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. Under Mac OS (System 7 or later), you can specify a flag in the DLOG resource that automatically positions the about box in the proper position.

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.

When Photoshop attempts to bring up the about box for a plug–in module, it will make the about box selector call to each of the plug–ins in the same file. If you have compiled more than one plug–in module into a single file, only one of them should display an about box, which should describe all of the modules. All other plug–in modules should ignore the about box selector call and just return to the plug–in host.

Memory management strategies

In most cases, the first action a plug–in module must take after the user executes it is to negotiate with Photoshop about memory usage. Other plug–in hosts may not support the same memory options.

The negotiation begins when Photoshop sets the **maxData** field of the plug-inParamBlock to indicate the maximum number of bytes it would be able to free up. The plug–in module then has the option of reducing this number. Reducing **maxData** can speed up most operations, since freeing up the maximum amount of memory requires Photoshop to move all of the image data for any currently open images out of of RAM and into its virtual memory file.

If you know that your plug–in module's memory requirements will be small—if it can process the image data in pieces, or if the image size is small—you should reduce **maxData** to your actual requirements. This will allow many plug–in operations to be performed entirely in RAM.

In many cases, your plug–in only needs a small amount of memory, but will operate faster if given more. You must make a tradeoff.

One strategy is to divide **maxData** by 2, thus allocating half the memory to Photoshop and half to the plug–in.

Another good strategy is to reduce **maxData** to zero, and then use the buffer and handle suites to allocate memory as it is 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.

Creating plug–in modules for Mac OS

Photoshop plug–in modules for the Macintosh can be created using any of the popular C compilers including Apple MPW, Symantec THINK 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 to begin at the first byte of the resource.

Hardware and system software configuration

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

Keep in mind that older versions of Photoshop will run, and thus your plug–in may be called, on 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.

Photoshop 3.0 requires a 68020 or better, Color QuickDraw, and 32–Bit QuickDraw, so if your plug–in only runs under Photoshop 3.0, you can assume these features are present.

Resources in a plug–in module

Besides the code resources (680x0), your plug–in module may include a variety of resources for your plug–in’s user interface, stored preferences, etc.

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 are discussed in great detail in chapters 4 and 5). 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.

Your plug–in module should also have a ‘vers’ resource (ID=1) to provide Finder version information. If you want a custom icon for your plug–in module, you should include the appropriate BNDL resource and colored icon resources.

Global variables

Most Macintosh applications reference global variables as negative offsets from register A5 (680x0 processors). If a plug–in module declares any global variables, their location would overlap Photoshop’s global variable space; changing them will likely result in a quick and spectacular crash.

One common way code resources avoid this problem is to use the A4 register in place of the A5 register. The Metrowerks CodeWarrior C compiler, for example, contains header files (SetupA4.h, A4Stuff.h) and pre–compiled libraries designed for A4 register usage. 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. This is the approach taken in the sample plug–ins.

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.

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.

Example plug–in modules

The six sample plug–ins included with this toolkit can be built using Apple MPW or Metrowerks CodeWarrior. They have been tested against MPW 3.3.1 and CodeWarrior 6.

The toolkit also includes new header files. 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. PIAcquire.h, PIExport.h, PIFilter.h, and PIFormat.h are the header files for the respective types of plug–in modules.

Also included are two sets of utilities: DialogUtilities and PIUtilities.

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.

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 are used.

Notes for CodeWarrior Gold users

Adobe Photoshop 3.0 uses the ‘PiPL’ resource (see chapter 4) 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 example 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
```

Notes for CodeWarrior Bronze users

The sample CodeWarrior project files in this toolkit are designed for CodeWarrior Gold to create “fat” binaries. If you use CodeWarrior Bronze 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 ResEdit resource files which are then used by the PPC project files.)

For example, the Dissolve.68k.µ project in the Filters sample is set to output a code resource named “Dissolve.68k.rsrc” with creator ‘RSED’ 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 statements:

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

should be changed to:

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

See chapter 4 for more information about PiPL resources.

Creating plug–in modules for Windows

Photoshop plug–ins for Windows can be created using Microsoft® Visual C++. The example plug–ins in this toolkit include Visual C++ makefiles.

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 (routine-Name) 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 PiPLs, Photoshop does a GetP-rocAddress 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, or Windows NT 3.5. Adobe Photoshop requires at least an 80386 processor.

Structure packing

Structure packing for all plug–in parameter blocks (FilterRecord, FormatRecord, AcquireRecord, ExportRecord and AboutRecord) should be the default for the target system (this has changed for 32–bit plug–ins for speed reasons). The Info structures (FilterInfo, FormatInfo, etc.) must be packed to byte boundaries. The PiMI resource should be byte aligned as before.

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 chapter 4) 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.

Even under Windows, you are encouraged to create and edit 'PiPL' resources in the Macintosh format, and then use the CNVTPIPL.EXE utility program to convert them to Windows .RC resource files. This utility will take care of all byte ordering issues automatically. If you use a native Windows resource editor, you must be careful to do the correct byte swapping manually.

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
```

assignment in the sample makefiles.

The way that messages are packed into wParam and lParam have 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 *Win32 Application Programming Interface: An Overview* for more information on Win32 message parameter packing.)

Be sure that the definitions for your Windows callback functions (dialog box functions, etc.) 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 PHOTOSHO.INI file with the profile string PLUGINDIRECTORY.

When Adobe Photoshop starts executing, it searches the files in the PLUGINDIRECTORY, looking for plug–in modules. When it finds a plug–in, it checks its version number, and if the version is supported, it adds the plug–in’s name 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' (Note: 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). Adobe Photoshop searches for acquisition modules by examining the resources of all files in the PLUGINDIRECTORY that have file extension .8B*, for resources of type _8BAM. The nameID, the integer value which uniquely identifies the resource, for each 8BAM in the file must be consecutively numbered starting at 1.

Utility programs and source code

This toolkit includes two groups of utility functions: PIUtilities and WinUtils.

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 are used.

Winutils.c provides support for some Mac Toolbox functions used in PIUtilities.c, including memory management functions (e.g NewHandle( ), etc.) The header file PITypes.h contains definitions for common Mac result codes, data types, and structures. These simplify writing plug–in modules for both Mac OS and Windows.

The Windows version of this toolkit also includes two handy utility programs: MACTODOS.EXE and CNVTPIPL.EXE. These two utilities are included in the UTILITY sub–directory.

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 chapter 4 for more information about PiPL resources.

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

Example plug–in modules

The four sample plug–ins included with this toolkit can be built using Visual C++ 2.0 (Two other examples, HistoryExport and IllustratorExport currently work only under Mac OS.)

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 into 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 in Adobe Photoshop 3.0 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 has added an error code to indicate that the host does not supply necessary functionality:

```
#define errPlugInHostInsufficient  -30900
```

Under 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.

Direct callbacks

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

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 during the actual main operation of the plug–in, not during long operations during the preliminary user interface.

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

ProcessEventProc( )

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

This callback is only useful under Mac OS; the ProcessEventProc call function in the Windows version of Adobe Photoshop does nothing.

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

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

DisplayPixelsProc( )

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

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

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

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

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

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.

The **platformContext** parameter is not used under 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.

AdvanceStateProc ( )

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

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

You can use the AdvanceStateProc callback in situations where you expect your plug–in module to be called repeatedly during its operation, for example a scanner import module that scans and delivers images in chunks. When working with very large images (larger than available RAM), most plug–in modules must process the image in pieces.

Without the AdvanceStateProc callback, a plug–in module is called from, and returns to, the plug–in host for each chunk of data. Each repeated call must go through the plug–in's main( ) entry point and through any pre–processing done by your plug–in module.

Using AdvanceStateProc this overhead is eliminated. Your plug–in can complete its entire operation within a single call from the plug–in host (not including any 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 should 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 the later chapters on specific plug–in types for information on how to use this callback.

The AdvanceStateProc callback is new in Adobe Photoshop 3.0.

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 Photoshop color picker (actually, using the user's preferred color picker),

(2) convert color values from one color space to another,

(3) return the current sample point,

(4) return either the foreground or background color.

Note: Versions of Photoshop prior to 3.0.4 contain a bug in the ColorServicesProc callback to convert a color from one color space to another. Photoshop 3.0.3 and earlier will return an error code (paramErr), and convert the requested color to RGB, regardless of the target conversion requested.

Refer to appendix A for a description of the ColorServicesInfo data structure.

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.

HostProc ( )

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

This callback contains a pointer to a host–defined function that can do anything the plug–in host wishes. Plug–in modules 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 3.0.4 does not perform any tasks in this callback. Earlier versions of Photoshop used the HostProc for private communication between Photoshop and some plug–in modules.

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: a 2 byte version number for the suite, a 2 byte count of the number of routines in the suite, and 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 3.0.4 are:

- the buffer suite
- the pseudo–resource suite
- the handle suite
- the image services suite
- the property suite

Buffer suite

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 module can avoid doing some of the accounting for space to be reserved. This simplifies the prepare phase for acquire, filter, and format plug–ins.

For most types of plug–in modules, buffer allocations can be delayed until they are actually needed. Unfortunately, export modules must account for 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.

In Adobe Photoshop 3.0.4, the current version of the buffer suite is 2.

AllocateBufferProc( )

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

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

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

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 MacOS, 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 under 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.

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 severe crashes.

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.

Pseudo–Resource suite

This suite of callback routine provides support for storing data with 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.

In Adobe Photoshop 3.0.4, the current version of the pseudo–resource suite is 3.

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 handle returned belongs to the plug–in host, and should be treated as a read only handle.

DeletePIResourceProc( )

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

This routine deletes the resource that would have been returned by GetPIResource. Note that since resources are identified by index rather than ID, this will cause subsequent resources to be renumbered.

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 (the limit in Photoshop is 1000 pseudo–resources), this routine will return memFullErr.

Handle suite

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 Mac OS.

The following suite of routines is used primarily for cross–platform support. Although you can allocate handles directly using the Macintosh Toolbox, you should use these callbacks 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, your are strongly encouraged to use the buffer suite routines for memory allocation rather 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 in fact a Mac OS handle.

In Adobe Photoshop 3.0.4, the current version number of the handle suite is 1.

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

The image services suite is new in version 3.0.4 of Adobe Photoshop. 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 3.0.4.

The current version number of the image services suite is 1.

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

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

Table 3–1: 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 following 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.

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

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 (see below) for each pixel in the **area** rectangle.

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 and using interpolation (**method** == 1) performs the appropriate binlinear 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.

The **interpolate1DProc** callback 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)]>
```

The **interpolate2DProc** callback 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 untouched any pixels whose sample coordinates are out of bounds. You need, however, to 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 Interpolation pop–up menu on the General Preferences dialog) and use that choice. You can retrieve this value using the GetProperty callback with the *propInterpolationMethod* key. Note that this version of the resampling callback does not support the bicubic interpolation method.

Property suite

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 module types.

> ***Note:*** Unfortunately, the term “property” is used with two quite different meanings in this toolkit. Besides being used in the property suite callbacks, the term is also applied to the new PiPL data structure, documented in chapter 4. There is no connection between PiPL properties and the property suite.

Properties can consist either of a 32 bit integer returned in simpleProperty, or a handle returned in complexProperty (see below). In the case of a complex (i.e. handle based) property, your plug–in module is responsible for disposing of the handle it is passed using the DisposePIHandleProc callback defined in the handle suite.

Properties involving strings—such as channel names and path names—are returned in a Photoshop handle, where the length of the handle (obtained with PIGetHandleSizeProc) determines the size of the string. 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 0x3842494D ('8BIM').

In Adobe Photoshop 3.0.4, the current version of the property suite is 1.

GetPropertyProc( )

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

The GetPropertyProc callback 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);
```

The SetPropertyProc callback allows you to update information in the plug–in host about the document currently being processed.

Properties marked “modifiable” in table 3–2 can be altered with this callback.

Property keys

Table 3–2 contains the property keys recognized by Adobe Photoshop version 3.0.4. Refer to the section immediately before this one, which discusses the property suite callbacks, for information on how to read or write these key values.

Table 3–2: Property keys

Property ID Type	**Description**
propNumberOfChannels 'nuch' simple	The 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)	The 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	The mode of the image using the constants defined in PIGeneral.
propNumberOfPaths 'nupa' simple	The number of paths in the document.
propPathName 'nmpa' complex (string)	The name of the indexed path. The paths are indexed starting with zero.
propPathContents 'path' complex (data structure)	The 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	The index of the work path or –1 if there is no work path.
propClippingPathIndex 'clpa' simple	The index of the clipping path or –1 if there is no clipping path.
propTargetPathIndex 'tgpa' simple	The index of the target path or –1 if there is no target path.
propBigNudgeH 'bndH' simple, modifiable	The 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, modifiable	The 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	The current interpolation method: 1 = point sample, 2 = bilinear, 3 = bicubic.
propRulerUnits 'rulr' simple	The current ruler units.

Table 3–2: Property keys (Continued)

Property ID Type	**Description**
propSerialString 'sstr' complex (string)	The serial number of the plug–in host as a string. You can use this to implement copy protection for your plug–in module.
propCaption 'capt' complex, modifiable	This is the file meta information in a IPTC-NAA record.
propHardwareGammaTable 'hgam' complex	The hardware gamma table (Windows only).

4 PiPL Resources

A Plug–In Property List, often called a 'PiPL' (pronounced “pipple”) is a flexible, extensible data structure for representing a plug–in module’s metadata.

PiPLs contain all the information Photoshop needs to identify and load plug–in modules, as well as flags and other static properties that control the operation of each plug–in. Your plug–in module should contain one or more ‘PiPL’ structures.

Plug–in Property Lists were introduced with version 3.0 of Adobe Photoshop. They replace the older Plug–in Module Information structure, or 'PiMI' (pronounced “pimmy”). PiMI resources were used with versions of Photoshop prior to 3.0, and are discussed in more detail in the next chapter.

Property structures and property lists

Plug–in **property structures** (or just **properties**) are the basic units of information stored in a **property list**. Properties are variable length data structures, which are uniquely identified by a vendor code, property key, and ID number. The valid properties are documented later in this chapter.

Appendix B contains a formal grammar for properties.

Creating PiPL resources

Under Mac OS, PiPLs are stored as Macintosh resources. Under Windows, PiPLs are stored as Windows resources.

On the Macintosh, you can create and edit PiPL resources with a text editor and the Rez compiler, or you can use a graphical resource editor like Resorcerer by Mathemæsthetics, Inc. (Note that ResEdit cannot edit PiPL resources except as raw hex bytes.) If you are unfamiliar with the format of Rez source code, refer to the appropriate Apple documentation. This toolkit includes a Macintosh Rez file, PIGeneral.r, which provides a complete definition of the PiPL property types.

The Windows version of the toolkit also includes a “PiPL Parser” application (CNVTPIPL.EXE) to transform a Macintosh “.r” source file into a Windows “.rc” resource file.

If you are developing for both the Macintosh and Windows platforms, you can easily convert your Macintosh PiPL resources into Windows’ custom PiPL format using CNVTPIPL.EXE. This enables you to keep just one copy of your PiPL information, and saves you the headache of converting PiPLs by hand.

Even if you are developing a plug–in module only for Windows, you are strongly encouraged to use the Macintosh Rez language to create the PiPLs, and then use CNVTPIPL.EXE to convert them. It is much easier to create the PiPLs this way since CNVTPIPL.EXE handles padding and byte–ordering issues for you automatically. If you use a Windows resource editor, you will have to remember to byte–swap fields where necessary.

Instructions on using CNVTPIPL.EXE can be found in chapter 2.

Loading PiPL resources

When Photoshop launches, it scans all plug–in files for 'PiPL' resources. Historically each type of plug–in had its own file type, listed in chapter 2.

File types are only a matter of convention for 'PiPL' based plug–in modules. All the above file types are searched for 'PiPL' resources and for those that are found, the information contained therein is used to determine the type of plug–in, code location, etc.

If no 'PiPL' resources are found in a plug–in file, the 'PiMI' search algorithm is used as documented in chapter 5. This allows you to place both 'PiPL' and 'PiMI' resources in a plug–in module if it is designed for both version 2.5 and 3.0.x.

Plug–in property lists

The plug–in property list structure has a version number and a count followed by one or more property structures (defined below).

A “C” struct definition for the plug–in property list is:

```
typedef struct PIPropertyList
{
    int32                   version;
    int32                   count;
    PIProperty              properties[1];
} PIPropertyList;
```

Table 4–1: PIPropertyList structure

Type	Field	Description
int32	version	This denotes the version of this specification the 'PiPL' is formatted to. The current version is 0.
int32	count	This field holds the number of properties contained in the 'PiPL'. 0 is a valid value denoting a 'PiPL' with no properties.
PIProperty array	properties	A variable length array of variable length property data structures. Holds the actual contents of the 'PiPL'.

Plug–in properties

Each property has a vendor code, a key, an ID, a length field.

```
typedef struct PIProperty
{
    OSType                  vendorID;
    OSType                  propertyKey;
    int32                   propertyID;
    int32                   propertyLength;
    char                    propertyData [1];
    /* Implicitly aligned to multiple of 4 bytes. */
} PIProperty;
```

Table 4–2: PIProperty structure

Type	Field	Description
OSType	vendorID	This field identifies the vendor defining this property type. This allows other vendors to define their own properties in a way that does not conflict with either Adobe or other vendors. It is recommended that a registered application creator code be used for the vendorID to ensure uniqueness. All Photoshop properties described in this document use the vendorID '8BIM'.
OSType	propertyKey	This field specifies the type of this property. Property types used by Photoshop are documented below. (You can think of a property type as similar to a resource type.)
int32	propertyID	In theory, the propertyID field can be used to store more than one property of a given type (similar to a resource ID). In practice, this field is always zero. It should be thought of as reserved for future use.
int32	propertyLength	This field contains the length of the **propertyData** field. It does not include any padding bytes after propertyData to achieve four byte alignment. **PropertyLength** may be zero.
variable	propertyData	This field contains the property’s data. In the property list, each property must be padded so that the next property begins on a four byte boundary.

General properties

These properties are common to all types of plug–in modules. The names of the properties (such as “PIKindProperty”) are the same as the #define names for the corresponding property keys.

PIKindProperty

Property Key:	0x6b696e64L ('kind')
Property Data:	OSType

This property encodes the type or kind of a plug–in module. In Photoshop, this determines where the plug–in will appear in the menus. Valid values are: shown in table 5–3.

Table 4–3: PIKindProperty

Module Type	PIKindProperty
Acquire	'8BAM'
Export	'8BEM'
Format	'8BIF'
Filter	'8BFM'
Parser	'8BYM'
Accelerator Extension	'8BXM'

PIVersionProperty

Property Key:	0x76657273L ('vers')
Property Data:	int32

This property encodes a major and minor version number indicating which revision of the plug–in interface this plug–in was written for. The major version number indicates incompatible changes while the minor version number indicates incremental enhancements. The major version number is encoded in the most significant 16 bits of the 32 bit version number, the minor version number is encoded in the least significant 16 bits.

There are separate version numbers for each kind of plug–in. The current version for a given kind of plug–in is defined by a preprocessor macro in the header file defining the interface for that plug–in type.

PIPriorityProperty

Property Key:	0x70727479L ('prty')
Property Data:	int16

This property determines the order in which this plug–in will be loaded. This is typically only important for hardware accelerator plug–in modules.

PIPriorityProperty can be used in format modules to determine the priority of multiple modules that can read a particular file format. See chapter 9 for details. This property can also be used to control the order in which items with the same name show up in menus.

Lower numbers (including negative ones since the field is signed) load first. If no PIPriorityProperty is present, the default is zero.

PIImageModesProperty

Property Key:	0x6d6f6465L ('mode')
Property Data:	FlagSet

This is a set of flags that determines which image modes the plug–in supports. In Adobe Photoshop, eleven modes are defined: bitmap, gray scale, indexed color, RGB color, CMYK color, HSL color, HSB color, multi–channel, duotone, Lab color, gray 16 color, and RGB 48 color.

PIRequiredHostProperty

Property Key:	0x686f7374L ('host')
Property Data:	OSType

This property should be used if a plug–in relies on features of a specific host. It is typically filled in with the applications creator code. (E.g. '8BIM' for Adobe Photoshop.)

PICategoryProperty

Property Key:	0x63617467L ('catg')
Property Data:	PString

If present, this property indicates what sub–menu to list this plug–in module under. For example, Adobe Photoshop filter plug–ins are often grouped under “Noise”, “Blur”, “Stylize”, etc. sub–menus under the Filter menu.

PINameProperty

Property Key:	0x6e616d65L ('name')
Property Data:	PString

This property indicates the name that the plug–in host should use on the menu for this plug–in module. Where a plug–in modules appears in the plug–in host’s menu hierarchy depends on the type of the plug–in module and the PICategoryProperty field.

Code descriptor properties

Code descriptors tell Photoshop the type and location of a plug–in’s code. More than one code descriptor may be included to build a “fat” plug–in which will run on different types of machines. Photoshop will select the best performing option. Photoshop makes sure that the callback structure is filled in with appropriate functions for the type of code that is loaded. So for PowerPC code, native function pointers will be provided and routine descriptor operations are not required either in calling the plug–in or for the plug–in to invoke Photoshop callback functions.

Note for Windows Developers: The CNVTPIPL.EXE utility only recognizes the “PIWin32X86CodeProperty” property. It ignores all Mac–specific properties described in this section.

PI68KCodeProperty

Property Key:	0x6d36386bL ('m68k')
Property Data:	PI68KCodeDesc structure

This property indicates a 68K code resource. The PI68KCodeDesc structure is:

```
typedef struct PI68KCodeDesc
{
    OSType resourceType;
    int16  resourceID;
} PI68KCodeDesc;
```

Any resource type may be used, but you are strongly encouraged to use the same type as the PIKindProperty, shown in table 5–3.

This convention comes from Photoshop 2.5.1 where these types were required. When building a plug–in module that is backwards compatible with 2.5.1 hosts, your must use these resource types.

PI68KFPUCodeProperty

Property Key:	0x36386670L ('68fp')
Property Data:	PI68KCodeDesc structure

This descriptor is just like a PI68KCodeDesc except it will only be used on Macintosh machines that are equipped with FPU hardware. This allows vendors to easily ship plug–ins that take advantage of FPU hardware but still run on non–FPU Macs.

PIPowerPCCodeProperty

Property Key:	0x70777063L ('pwpc')
Property Data:	PICFMCodeDesc

This descriptor indicates a PowerPC code fragment in the data fork of the plug–in file. The type for this property is as follows:

```
typedef struct PICFMCodeDesc
{
    long fContainerOffset;
    long fContainerLength;
    char fEntryName[1];
} PICFMCodeDesc;
```

Table 4–4: PICFMCodeDesc structure

Type	Field	Description
long	fContainerOffset	This field contains the offset within the data fork for the start of this plug–in's code fragment. This allows more than one code fragment based plug–in per file.
long	fContainerLength	This field holds the length of this plug–ins code fragment. If the fragment extends to the end of the file (e.g. it is the only fragment in the file), the container length may be 0.
Pstring	fEntryName	This field is represented as a Pascal string and is used to lookup the address of the function to call within the fragment. If the entrypoint name is a zero length string, the default entrypoint for the code fragment will be used.

Entrypoint names allow more than one plug–in to be exported from a single code fragment.

Note: in order for the Code Fragment Manager to find an entrypoint by name, that name must be an exported symbol of the code fragment.

PIWin32X86CodeProperty

Property Key:	0x77783836L ('wx86')
Property Data:	PIWin32X86CodeDesc (NULL terminated string)

This code descriptor is used for 32 bit Windows DLLs, and contains the DLL's entrypoint name.

```
typedef struct PIWin32X86CodeDesc
{
    char                fEntryName[1];
} PIWin32X86CodeDesc;
```

The string may need to be padded with additional NULLs to satisfy the 4 byte alignment requirement.

Export–specific properties

This property is only applicable to export plug–in modules.

PIExpFlagsProperty

Property Key:	0x65787066L ('expf')
Property Data:	FlagSet

This property indicates that the acquire plug–in module can see transparency data. To indicate this, set the flag:

```
#define PIExpSupportsTransparency  0
```

Filter–specific properties

These properties are only applicable to filter plug–in modules.

PIFilterCaseInfoProperty

Property Key:	0x66696369L ('fici')
Property Data:	Array of seven bytes

The key feature of Photoshop 3.0 is support for dynamically composited layers of image data.

A layer consists of color and transparency information for each pixel it contains. Previous versions of Photoshop did not have a transparency component. Transparency introduces a greater richness as well as a number of interesting problems. First, completely transparent pixels have an undefined color. Second, filters will likely affect transparency data as well as color data. This is especially true for filters which produce spatial distortions.

Photoshop 3.0 offers flexibility in how transparency data is presented to filters. The filter case info property controls the filtering process and presentation of data to the plug–in. This property provides information to Photoshop about what image data cases the plug–in supports. Photoshop then compares the current filtering situation to the supported cases and chooses the best fitting case. The image data is then presented in that case. If none of the supported cases are usable, the filter will be disabled.

The case properties are an array of seven four byte entries, one for each case.

The seven cases are shown in table 5–5. (These #define constants are passed to a filter plug–in module in the **filterCase** field. Note that the case array is one–indexed.)

Table 4–5: Filter cases

#define name	Description
filterCaseFlatImageNoSelection (1)	This is a background layer or a flat image. There is no transparency data or selection.
filterCaseFlatImageWithSelection (2)	No transparency data, but a selection may be present. The selection will be presented as mask data.
filterCaseFloatingSelection (3)	Image data with an accompanying mask.
filterCaseEditableTransparencyNoSelection (4)	A layer with transparency editing enabled and no selection.
filterCaseEditableTransparencyWithSelection (5)	A layer with transparency editing enabled and a selection.
filterCaseProtectedTransparencyNoSelection (6)	A layer with transparency editing disabled and no selection.
filterCaseProtectedTransparencyWithSelection (7)	A layer with transparency editing disabled and a selection.

Adobe Photoshop follows an algorithm to determine whether a particular case is supported. If the editable transparency cases are unsupported, then Photoshop will try the corresponding protected transparency cases. This is important because this governs whether the filter will be expected to filter the transparency data as well as the color data.

If the protected transparency case without a selection is disabled, Photoshop will fall through from there to treating the layer data as a floating selection. As such, the transparency data will be presented via the mask portion of the interface rather than with the input data.

Each of the seven elements of the array contains a four byte FilterCaseInfo structure.

```
typedef struct FilterCaseInfo
{
    char inputHandling;
    char outputHandling;
    char flags1;
    char flags2;
} FilterCaseInfo;
```

The **inputHandling** and **outputHandling** fields specify the pre–processing and post–processing actions on the image data respectively.

Table 4–6: Handling modes

Handling mode	Description
filterDataHandlingCantFilter (0)	indicates that this case is not supported by the plug–in filter
filterDataHandlingNone (1)	indicates that the plug–in filter does not expect the plug–in host to do anything to the image data.
The next three modes are matting cases, which are useful when performing spatial distortions and blurs. You can matte the data, process it, and then dematte to remove the added color. For these cases, the matting is defined as follows: mattedValue = ((unmattedValue * transparency) + 128) / 255 + ((matConstant * (255 - transparency)) + 128) / 255 Dematting is defined as follows: unmattedValue = ((mattedValue - matConstant) ./ transparency) + matConstant with the ./ operator defined to be a suitable 8 bit fixed–point divide and the result value being pinned to the range of 0 to 255.	
filterDataHandlingBlackMat (2)	For the input case, matte the image data with black (0) values based on the transparency. For output, dematte the image data using black (0) values.
filterDataHandlingGrayMat (3)	Matte the image data with gray (128) values based on the transparency on input. Dematte the image data using gray (128) values on output.
filterDataHandlingWhiteMat (4)	Matte the image data with white (255) values based on the transparency on input. Dematte the image data using white (255) values on output.
The following modes are only useful for input:	
filterDataHandlingDefringe (5)	Defringe transparent areas filling with the nearest defined pixels using taxicab distance. Note that this only applies to fully transparent pixels.
filterDataHandlingBlackZap (6)	Set color component of totally transparent pixels to black (0).
filterDataHandlingGrayZap (7)	Set color component of totally transparent pixels to gray (128).
filterDataHandlingWhiteZap (8)	Set color component of totally transparent pixels to white (255).

Table 4–6: Handling modes (Continued)

Handling mode	Description
filterDataHandlingBackgroundZap (10)	Set color component of totally transparent pixels to the current background color.
filterDataHandlingForegroundZap (11)	Set color component of totally transparent pixels to the current foreground color.
The following mode is only useful for output:	
filterDataHandlingFillMask (9)	This mode results in the transparency mask automatically being filled with full opacity in the area affected by the filter. This is only valid for the editable transparency cases. This option is provided to make it easy to write a plug–in similar to Photoshop's Clouds plug–in, which fills an area with a value.

The **flags1** field of the FilterCaseInfo structure holds the following bits:

Note: This field is not a FlagSet. The first bit (PIFilterDontCopyToDestinationBit) is in the least–significant bit of the flag byte.

```
#define PIFilterDontCopyToDestinationBit 0
```

Normally Photoshop copies the source data to the destination before filtering. This gives a good default value for any pixels the filter does not write too, but degrades performance for filters which write all the output pixels. Setting this bit inhibits the copying behavior.

```
#define PIFilterWorksWithBlankDataBit 1
```

This flag determines whether the filter will work on "blank" areas. That is, areas that are completely transparent. If not, an error message will be given when the filter is invoked on a blank area. This is only valid for the editable transparency case because that is the only case where you could create opacity—in the protected transparency case, you would be left with what you started with: completely blank data.

```
#define PIFilterFiltersLayerMaskBit 2
```

In cases where transparency is editable, this flag determines if Layer Masks are filtered. (See the "Add Layer Mask" item in the Layers palette menu to create a layer mask.) Setting this bit adds the layer mask to the set of target channels if: transparency for the layer is editable (i.e., this is one of the editable transparency cases), the bit is set, and the layer mask is specified as being positioned relative to the layer rather than the image in Layer Mask Options. This is the same logic Photoshop uses for built–in filters like blur. The distinction based on position is made with the assumption that layer relative masks will need to be distorted along with the layer while image relative masks are independent of the layer.

The **flags2** field of the FilterCaseInfo structure is reserved, and should be zero.

Format–specific properties

These properties are only applicable to format plug–in modules.

PIFmtFileTypeProperty

Property Key:	0x666d5443L ('fmTC')
Property Data:	TypeCreatorPair

Determines the default type and creator code used for files newly created with this format plug–in.

Under Windows, files don't store TypeCreator information, except internally, so the PIFmtFileTypeProperty is not required; they are always interpreted as of type 'BINA' and creator 'mdos'.

All the info regarding what files can be read and written is obtained from the PIReadExtProperty or the PIFilteredExtProperty.

Under Windows, PiMI extensions are converted to PIReadExtPropertys, so use of PIFilteredExtProperty requires additional coding if you are porting a 16–bit plug–in format module to 32–bit.

PIReadTypesProperty

Property Key:	0x52645479L ('RdTy')
Property Data:	Array of TypeCreatorPairs

This property contains a list of type and creator pairs which the format plug–in can read. Specifying a value of four spaces (0x20202020L) matches any type or creator.

PIFilteredTypesProperty

Property Key:	0x66667454L ('fftT')
Property Data:	Array of TypeCreatorPairs

This property contains a list of type and creator pairs for which the file format plug–in should be called to determine if the file can be read. Specifying a value of four spaces (0x20202020L) matches any type or creator.

PIReadExtProperty

Property Key:	0x52644578L ('RdEx')
Property Data:	Array of OSTypes

This property contains a list of extensions which the format plug–in can read. The extension is stored in the first three characters of the OSType. The fourth character must be a space.

PIFilteredExtProperty

Property Key:	0x66667445L ('fftE')
Property Data:	Array of OSTypes

This property contains a list of extensions for which the file format plug–in should be called to determine if the file can be read. See documentation for formatSelectorFilterFile plug–in selector.

PIFmtFlagsProperty

Property Key:	0x666d7466L ('fmtf')
Property Data:	FlagSet

This property contains a set of flags which control the operation of file format plug–ins. The default value for any flag is false.

The bit fields of the flag are as follows:

```
#define PIFmtReadsAllTypesFlag 0
```

This field is obsolete.

```
#define PIFmtSavesImageResourcesFlag 1
```

Along with the pixel information for a file, Photoshop stores various image resources: printing information, pen tool paths, etc.. Collectively, these are known as image resources. The plug–in format has the option of taking responsibility for these resources by reading and writing a block of data containing the image resources. If this flag is false, Photoshop will add the image resources to the file's resource fork (Mac OS) but this will not be portable to other platforms.

```
#define PIFmtCanReadFlag 2
```

This flag should be set to true if the file format can read files.

```
#define PIFmtCanWriteFlag 3
```

This flag should be set to true if the file format can write files.

```
#define PIFmtCanWriteIfReadFlag 4
```

Flag indicating whether your plug–in can write the file if the plug–in originally read the file. For example, the plug–in to support Adobe Premiere's Filmstrip format has the can write flag set to false because it cannot in general be used to save files. It has this flag set to true, however, because it can save out filmstrips that were read in using the plug–in.

PIFmtMaxSizeProperty

Property Key:	0x6d78737aL ('mxsz')
Property Data:	Point

The maximum number of rows and columns that can be in an image saved in this format. Photoshop will use this field to screen out ineligible formats.

PIFmtMaxChannelsProperty

Property Key:	0x6d786368L ('mxch')
Property Data:	Array of int16s

An array of counts of the maximum number of channels which can/will be saved for a given image mode.

This array is indexed by the plug–in mode constants. For example, if your format plug–in supports a single alpha channel in RGB mode, you should set maxChannels[plugInModeRGBColor] to four.

A plug–in may still be asked to save more channels than it reports it can support. This field exists primarily so that Photoshop can warn the user that alpha channels will be discarded.

5 PiMI Resources

PiMI (pronounced “pimmy”) resources have been superceded by PiPL resources, but you may need to include a PiMI resource if your want your plug–in module to work with older (pre–3.0) versions of Adobe Photoshop. Adobe recommends that you also create a PiPL resource for your plug–in, as this will give you greater control over its operation under 3.0.

If your plug–in module is designed to be used only with Photoshop 3.0 or later, you should not create a PiMI resource, and can skip this chapter.

Older PiMI based plug–in modules are still fully supported in Photoshop 3.0. This is accomplished by converting the 'PiMI' resource into a 'PiPL' resource when the plug–in is first scanned. Since 'PiPL's are cached in Photoshop’s preferences file, this conversion only happens once.

If you want your plug–in to work with versions of Photoshop prior to 3.0, you must create a PiMI resource.

A PiMI resource is a fixed format structure which originally contained only a version number. With the evolution of Photoshop’s plug–in interface, this structure expanded to include other information. The addition of multiple plug–in types resulted in the PiMI becoming a variant record with generic data at the beginning and a type specific data at the end. Further plug–in interface evolution required more complex metadata, such as an array of allowable file types for file format plug–ins.The combination of variant and variable sized fields in the 'PiMI' made writing resource templates for them very difficult. Requirements for new plug–in metadata in Photoshop 3.0 introduced further complexities. The more general and flexible 'PiPL' mechanism was designed to address these issues.

The PiMI resource consists of two pieces: general information applicable to all (or most) plug–in types followed by type specific info. Since the information proceeds serially, however, all fields must be filled in through and including the last field supplied. Your plug–in should either just include the version number information, or it should include all of the information documented here.

A “C” struct definition for the PiMI resource is:

```
typedef struct PlugInInfo
{
    short               version;
    short               subVersion
    short               priority;
    short               generalInfoSize;
    short               typeInfoSize;
    short               supportsMode;
    OSType              requireHost;

} PlugInInfo;
```

Table 5–1: PlugInInfo (PiMI) structure

Type	Field	Description
short	version	The major version number for the interface used by the plug–in. This field is required.
short	subVersion	The minor version number for the interface used by the plug–in. This field is required.
short	priority	The priority which should be associated with this plug–in when it loads. Currently, this is only used for extension modules.
short	generalInfoSize	The size of the general plug–in information in this resource.
short	typeInfoSize	The size of the type–specific plug–in information in this resource. This information follows the requiredHost field. The type specific info is documented in the chapters for the various types of plug–ins.
short	supportsMode	A bitmap describing the image modes supported by the plug–in. This field applies to filter, export, and file format plug–ins. If it is not present, Photoshop will assume that the plug–in supports all image modes. This field is one of the ways Photoshop decides whether to dim plug–ins in menus. Since not all plug–in hosts may respect this field, your plug–in module should still check that it can handle the image mode it has been requested to process. The bits in the bitmap correspond to the plugInMode constants in PIGeneral.h (i.e. bit 0 corresponds to bitmaps, bit 1 to grayscale, etc.).
short	requireHost	If your plug–in requires a particular plug–in host, you should specify the signature for that host here. If you do not require a particular plug–in host, you should fill this field with spaces. Photoshop will not load plug–in modules which require a plug–in host other than Photoshop's '8BIM' signature. You should not count on other applications that support the Photoshop plug–in architecture to behave in a similar fashion.

6 Acquire Modules

Acquire 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.

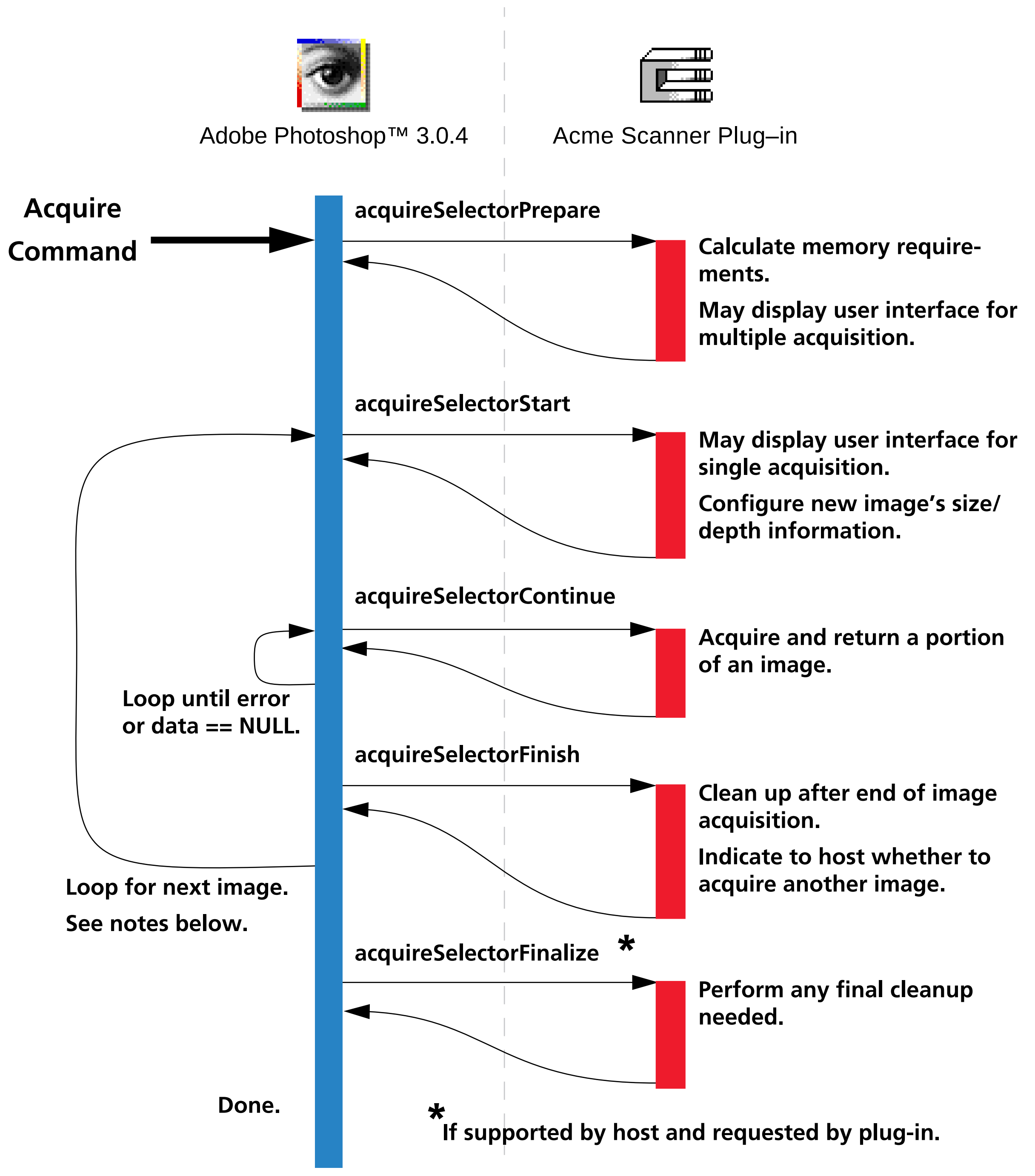
Acquire modules can also be used to read images from unsupported file formats, although file format modules often are better suited for this purpose (file format modules are accessed directly from the Open... command, while acquire modules use the Acquire sub–menu.)

Under Mac OS, the code resource and file type for acquire modules is '8BAM'. Under Windows, the file extension is .8BA.

Sample plug–in

DummyScan is a sample acquire module. This is a new version of DummyScan which is Photoshop 3.0 specific, since it uses the advanceState callback and the improved multiple acquire design.

Calling sequence


Adobe Photoshop™ 3.0.4
Acme Scanner Plug–in
Acquire Command
acquireSelectorPrepare
Calculate memory requirements.
May display user interface for multiple acquisition.
acquireSelectorStart
May display user interface for single acquisition.
Configure new image's size/ depth information.
acquireSelectorContinue
Acquire and return a portion of an image.
Loop until error or data == NULL.
acquireSelectorFinish
Clean up after end of image acquisition.
Indicate to host whether to acquire another image.
Loop for next image.
See notes below.
acquireSelectorFinalize *
Perform any final cleanup needed.
Done.
* If supported by host and requested by plug-in.

The calling sequence for acquire modules is a little more complex than other types of plug–in modules. In a single invocation, acquire 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 acquire plug–in module by selecting its name from the Acquire 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 acquireSelectorReadPrepare 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. For acquire modules to perform efficiently, you should reduce **maxData** to permit Photoshop to process the acquired image in RAM. Refer to chapter 3 for details on memory management strategies.

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

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.

The area of the image being returned is specified by **theRect** and by **loPlane** and **hiPlane**. The **data** field should point to the actual data being returned. The fields **colBytes, rowBytes** and **planeBytes** specify the organization of the data.

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, **loPlane** and the **hiPlane** should be set to 0, **colBytes** should be set to 1, and **rowBytes** should be set to the width of the area being returned (**planeBytes** is ignored in this case, since **loPlane** == **hiPlane**).

If instead, you wish to return the RGB data in interleaved form (RGBRGB...), the **loPlane** should be set to 0, **hiPlane** to 2, **planeBytes** to 1, **colBytes** to 3, and **rowBytes** to 3 times the width of area being returned.

The portion of the image being returned is specified by **theRect**. If the resolution of the acquired 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) unless the buffer for the image data was allocated using the buffer or handle suites.

The **data** field contains a pointer to the data being returned. Most plug–ins will allocate a buffer for the data using the NewPtr trap (Mac OS), GlobalAlloc function (Windows) or via the buffer suite. Your plug–in module is responsible for freeing this buffer in its acquireSelectorFinish handler.

acquireSelectorFinish

This call allows your plug–in to clean up after an image acquisition. 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 (Mac OS) or Escape (Windows) while processing the results of an acquireSelectorContinue call, it will call the acquireSelectorFinish routine. Be careful here, since normally your plug–in would be expecting another acquireSelectorContinue call.

If the following conditions are true:

(a) the plug–in host supports multiple acquisitions (which Photoshop 3.0 does)

(b) your plug–in module set **acquireAgain** = TRUE, and

(c) the acquireSelectorContinue loop finished normally (no error was returned, the loop ended with **data** == NULL),

then the plug–in host can loop back to acquireSelectorStart to begin acquiring another image.

Alternately, if the following conditions are true:

(a) the plug–in host supports multiple acquisitions,

(b) the plug–in host set **canFinalize** = TRUE,

(c) your plug–in module set **wantFinalize** = TRUE and **acquireAgain** = TRUE, and

(d) the acquireSelectorContinue loop finished with a result code >= 0 or a result code of userCanceledErr,

then the plug–in host can loop back to acquireSelectorStart to begin acquiring another image.

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.

Notes:

1. If acquireSelectorPrepare succeeds—the result value is zero—and **want-Finalize** is TRUE, then Photoshop guarantees that acquireSelectorFinalize will be called.

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

3. In the event of any error during acquisition, the document being acquired is discarded.

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

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

6. 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.

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.

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

The Acquire 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 6–1: 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 preview image for cropping. It should be used during the main, high–resolution scan.
int32	maxData	Photoshop initializes this field to the maximum of 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 acquired (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, and 8 for all other modes, except grayscale and RGB which 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. Because of the implementation of the plane map, acquire modules (and format modules) should never try to work with more than 16 planes at a time, even though acquire modules can create documents with up to 24 channels.

Table 6–1: 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 terms of pixels per inch. This is a fixed point number (16 binary digits). 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. For acquire plug–in modules, this is a change from previous versions of Photoshop's 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 acquired images as "Untitled–..." . File importing acquire modules should set this field to the file's name in their acquireSelectorStart routines, so Photoshop can display the correct window title. Scanning modules should ignore this field.
int16	vRefNum	If your plug–in module sets **fileName**, you should also set vRefNum to the file's volume reference number. This is only applicable under Mac OS, it is ignored under Windows.
Boolean	dirty	By default, newly acquired images are marked as dirty, meaning that the user will be prompted to save the image when closing the window. File importing acquire modules should set this field to FALSE to prevent this.

Table 6–1: 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 (LSB = bit 0) is 1, 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. Note that 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** is 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 acquire 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, it should set this field to –1.
SpaceProc	spaceProc	If not NULL, this field contains a pointer to the SpaceProc callback. See chapter 3 for details.

Table 6–1: AcquireRecord fields (Continued)

Type	Field	Description
PlugInMonitor	monitor	This field contains the monitor setup information for the host. Refer to Appendix A for details.
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 plug–in host, otherwise NULL. See chapter 3 for details.
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 for details.
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 acquire modules 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 for acquire modules, 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 your plug–in module wishes to be called again to acquire another image, you should set this flag in your acquireSelectorFinish handler. Plug–in hosts that support multiple image acquisition should start the acquisition process again with a call to acquireSelectorStart to begin acquiring a new image. If you do not want to put up a user interface for each acquisition, you should display your interface during the acquireSelectorPrepare call. With the addition of the finalize selector, acquire plug–in modules can now put up an interface that remains active across multiple acquisitions. Your plug–in module should not count on being called again just because it sets this flag; your acquireSelectorFinish handler 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 for details.
HandleProcs *	handleProcs	This field contains a pointer to the handle suite if it is supported by the host, otherwise NULL. See chapter 3 for details.
These fields are new in version 3.0.		

Table 6–1: AcquireRecord fields (Continued)

Type	Field	Description
Boolean	wantFinalize	This flag requests an acquireSelectorFinalize call if the host provides the newer protocol (see also **canFinalize**).
char[3]	reserved1	This 3 byte field is used for alignment to a four–byte boundary.
ColorServicesProc	colorServices	This field contains a pointer to the ColorSer-vices 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 in 3.0.4.		
ImageServicesProcs *	imageServicesProcs	This is a pointer to the image services callback suite. See chapter 3 for details.
int16	tileWidth	The width and height of a tile.
int16	tileHeight	
Point	tileOrigin	The origin of a known tile.
PropertyProcs *	propertyProcs	A pointer to the Property callback suite. Refer to chapter 3 for more information.
char[200]	reserved	These are set to zero by the plug–in host for future expansion of the plug–in standard. Do not use these.

7 Export Modules

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

Export modules can also be used to save images in unsupported or compressed file formats, although file format modules (see chapter 8) often are better suited for this purpose (file format modules are access directly from the Save or Save As... commands, while export modules use the Export sub–menu).

Under Mac OS, the code resource and file type for export modules is '8BEM'. Under Windows, the file extension is .8BE.

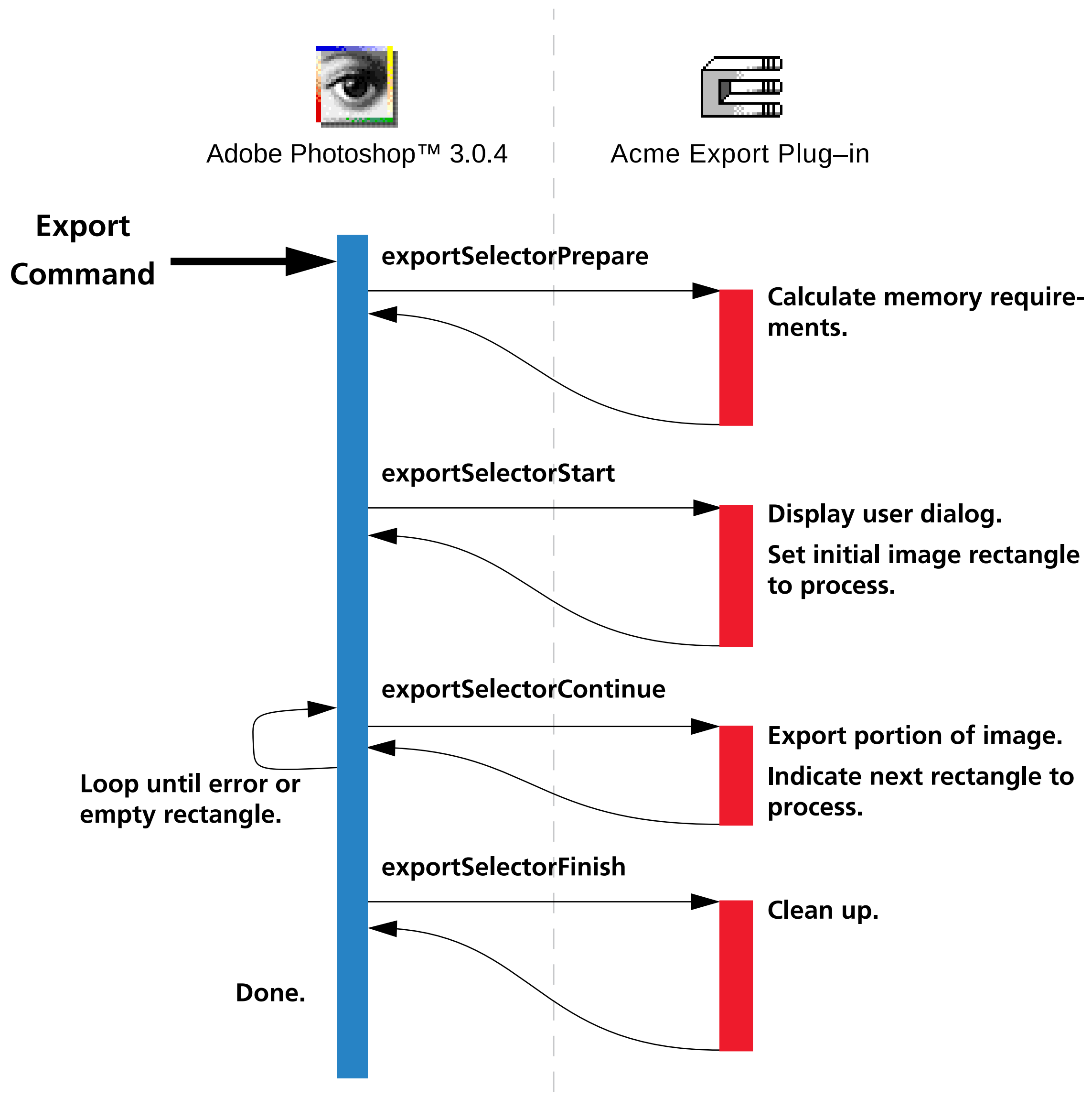
Sample plug–ins

DummyExport is a sample export module.

HistoryExport is a sample export module primarily concerned with demonstrating the pseudo–resource callbacks. It works in conjunction with the Dissolve plug–in to maintain a series of history strings for a file. This sample only works on Macintosh platforms.

Paths to Illustrator demonstrates using the getProperties callback and exporting of pen path information. The sample code works only on Macintosh platforms. It is fairly straightforward to extend the porting concepts from other examples to port this one over to the Windows platform. Please read the comments inside the sample source for important information regarding pen paths (like byte ordering etc.).

Calling sequence



When the user invokes an export plug–in module 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.

exportSelectorStart

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

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 and vertical strips are allowed as are 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 (Mac OS) or Escape (Windows) between calls to the exportSelectorContinue routine, it will call the exportSelectorFinish routine. You should be careful here, since normally the plug–in would be expecting another exportSelectorContinue call.

Notes:

1. If exportSelectorStart succeeds then Photoshop guarantees that exportSelectorFinish will be called.
2. Photoshop may choose to go exportSelectorFinish instead of exportSelectorContinue if it detects a need to terminate while building the requested buffer.
3. 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 no work being done.

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.

```
#define exportBadParameters –30200 //an error with the interface parameters
#define exportBadMode       –30201 //the 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 7–1: 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 the header file for possible 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 numbers (16 binary digits).
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 7–1: 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** is not equal to **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 module 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.
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 acquisition modules, and should be treated as a black box by plug–ins.
int16	thePlane	Currently selected channel, or –1 if a composite color channel, or –2 if some other combination of channels.
PlugInMonitor	monitor	This field contains the monitor setup information for the host. See Appendix A for more details.
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.
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. It contains NULL if the callback is not supported.
DisplayPixelsProc	displayPixels	This field contains a pointer to the DisplayPixels callback. It contains NULL if the callback is not supported.

Table 7–1: ExportRecord structure (Continued)

Type	Field	Description
HandleProcs *	handleProcs	This field contains a pointer to the handle suite if it is supported by the host, otherwise NULL.
ColorServicesProc	colorServices	This field contains a pointer to the ColorServices callback documented in the general documentation. It contains NULL if the callback is not supported.
GetPropertyProc	getProperty	This field contains a pointer to the property suite if it is supported by the host, otherwise NULL. This direct callback has been replaced by the Property suite (below), but is maintained here for backwards compatibility.
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	invertedLayer-Masks	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 in version 3.0.4 of Adobe Photoshop.		
ImageServicesProcs *	imageServicesProcs	The image services callback suite. See chapter 3 for details.
int16	tileWidth	The width of the tiles. Zero if not set.
int16	tileHeight	The height of the tiles. Zero if not set.

Table 7–1: ExportRecord structure (Continued)

Type	Field	Description
Point	tileOrigin	The origin point for the tiles.
PropertyProcs *	propertyProcs	A pointer to the property callback suite. See chapter 3 for details.
char[194]	reserved	These bytes are set to zero by the host for future expansion of the plug–in standard. Must not be used by plug–ins.

8 Filter Modules

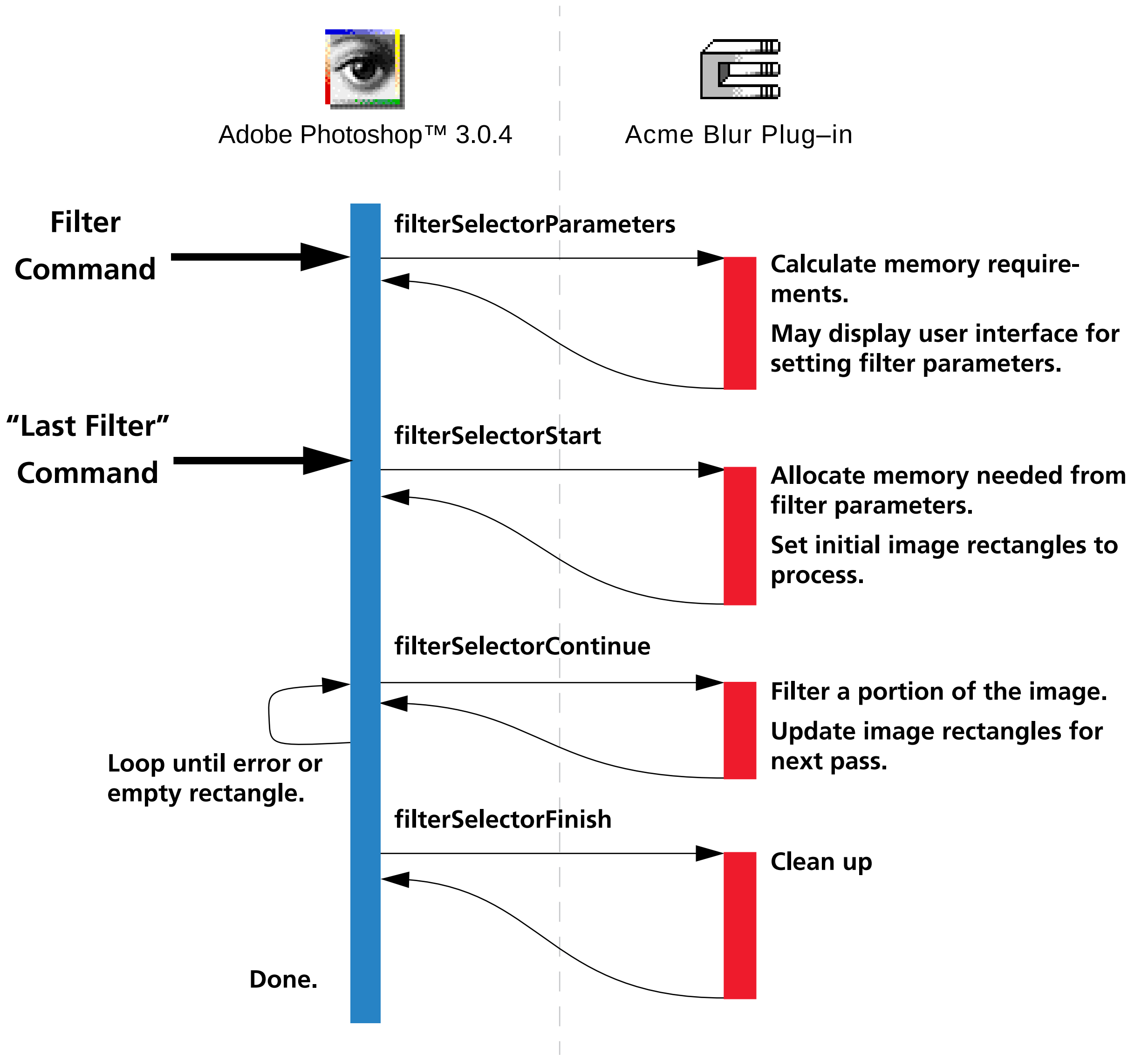
Filter modules modify a selected area of an image, and are accessed under the Filter menu. Filter actions range from subtle shifts of hue or brightness, to wild changes that create stunning visual effects.

Under Mac OS, the code resource and file type for filter modules is '8BFM'. Under Windows, the file extension is .8BF.

Sample plug–in

Dissolve is a sample filter plug–in which also demonstrates how to manipulate Photoshop’s layers.

Calling sequence


Adobe Photoshop™ 3.0.4
Acme Blur Plug–in
Filter Command
filterSelectorParameters
Calculate memory requirements.
May display user interface for setting filter parameters.
"Last Filter" Command
filterSelectorStart
Allocate memory needed from filter parameters.
Set initial image rectangles to process.
filterSelectorContinue
Filter a portion of the image.
Update image rectangles for next pass.
Loop until error or empty rectangle.
filterSelectorFinish
Clean up
Done.

When the user invokes a filter plug–in module by selecting its name from the Filter menu, 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 under the Filter menu). In this case, the plug–in host does not call filterSelectorParameters, and the user will not be shown any dialogs to enter new 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).

A future version of Photoshop may have a macro processor which would save the block pointed to by the parameters field when recording, so that it can operate the filter without user input during play back (Adobe Premiere™ actually uses this feature). To be compatible with this feature, all parameters must be saved in a relocatable block whose handle is stored in the parameters field.

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).

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

You may wish to design your filter module so that you store default values or the last used set of values for your parameter block in the filter module's resource fork (Mac OS) or another file (Windows). This way, you can save preference settings for your filter plug–in across invocations of the plug–in host.

filterSelectorPrepare

The filterSelectorPrepare 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.

The fields such as **imageSize**, **planes**, **filterRect**, etc. have now been defined, and can be used in computing your buffer size requirements.

If your plug–in filter module is planning on allocating any large (>= about 32K) buffers or tables, 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.

filterSelectorStart

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.

filterSelectorContinue

Your filterSelectorContinue handler is called repeatedly as long as at least one of the **inRect**, **outRect**, or **maskRect** fields is non–empty.

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 (Mac OS) or Escape (Windows) between calls to the filterSelectorContinue routine, it will call the filterSelectorFinish routine. Be careful here, since normally the plug–in would be expecting another filterSelectorContinue call.

Notes:

1. If filterSelectorStart succeeds, then Photoshop guarantees that filterSelectorFinish will be called.
2. Photoshop may choose to go to filterSelectorFinish instead of filterSelectorContinue if it detects a need to terminate while fulfilling a request.
3. 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.

```
#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 8–1: 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 pre–version 4 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	

Table 8–1: FilterRecord structure (Continued)

Type	Field	Description
int32	maxSpace	This lets the plug–in know the maximum number of bytes of information it can expect to be able to access at once (input area size + output area size + mask area size + bufferSpace).
int32	bufferSpace	If the plug–in is planning on allocating any large internal buffers or tables, it should set this field during the filterSelectorPrepare call to the number of bytes it is planning to allocate. Photoshop will then try to free up the requested amount of space before calling the filterSelectorStart routine.
Rect	inRect	Set this field in your filterSelectorStart and filterSelectorContinue handlers to request access to an area of the input image. The area requested must be a subset of the image's bounding rectangle. After the entire filterRect has been filtered, this field should be set to an empty rectangle.
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** is not equal to **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** is not equal to **outHiPlane**), the data is interleaved.
int32	outRowBytes	The offset between rows of the output image data. There may or may not be pad bytes at the end of each row.
Boolean	isFloating	This field is set TRUE if and only if the selection is floating.
Boolean	haveMask	This field is set TRUE if and only if a non–rectangular area has been selected.

Table 8–1: FilterRecord structure (Continued)

Type	Field	Description
Boolean	autoMask	By default, Photoshop automatically masks any changes to the area actually selected. If **isFloating** is FALSE, and **haveMask** is TRUE, your plug–in can turn off this feature by setting this field to FALSE. It can then perform its own masking.
Rect	maskRect	If **haveMask** is 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.
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 the header file for possible values. Your filterSelectorStart handler should return a filterBadMode result code 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 binary digits).
Fixed	imageVRes	
Point	floatCoord	If isFloating is TRUE, the coordinate of the top–left corner of the floating selection in the main image's coordinate space.
Point	wholeSize	If isFloating is TRUE, the size in pixels of the entire main image.
PlugInMonitor	monitor	This field contains the monitor setup information for the host. See appendix A for details.
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 for details.

Table 8–1: FilterRecord structure (Continued)

Type	Field	Description
ResourceProcs *	resourceProcs	This field contains a pointer to the pseudo–resource suite if it is supported by the host, otherwise NULL. See chapter 3 for details.
ProcessEventProc	processEvent	This field contains a pointer to the ProcessEvent callback. It contains NULL if the callback is not supported. See chapter 3 for details.
DisplayPixelsProc	displayPixels	This field contains a pointer to the DisplayPixels callback. It contains NULL if the callback is not supported. See chapter 3 for details.
HandleProcs *	handleProcs	This field contains a pointer to the handle callback suite if it is supported by the host, otherwise NULL. See chapter 3 for details.
These fields are new in version 3.0.		
Boolean	supportsDummyPlanes	Does the host support the plug–in requesting non–existent planes? (see dummy planes 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.
int16	dummyPlaneValue	The value to store into any dummy planes. 0..255 = specific value. –1 = leave undefined (i.e., random)
void *	premiereHook	See the Adobe Premiere Plug–in Developer's Kit.
AdvanceStateProc	advanceState	The AdvanceState callback. See chapter 3 for details on this callback function.
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 any, followed by the transparency, followed by any layer masks, followed by any alpha channels.
Boolean	wantsAbsolute	Enable absolute channel indexing for the input. This is only useful if **supportsAbsolute** is TRUE. Absolute indexing is useful for things like accessing alpha channels.

Table 8–1: FilterRecord structure (Continued)

Type	Field	Description
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 for details.
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.
int16	inputPadding	The input, output, and mask can be padded when loaded. The options for padding include specifying a specific value (0..255), specifying edge replication (plugInWantsEdgeReplication), specifying that the data be left random (plugInDoesNotWantPadding), or requesting that an error be signaled for an out of bounds request (plugInWantsErrorOnBoundsException). The error case is the default since previous versions would have errored out in this event.
int16	outputPadding	
int16	maskPadding	
char	samplingSupport	Does the host support non–1:1 sampling of the input and mask? Photoshop 3.0.1 supports integral sampling steps (it will round up to get there). This is indicated by the value hostSupportsIntegralSampling. Future versions may support non–integral sampling steps. This will be indicated with hostSupportsFractionalSampling.
char	reservedByte	(for alignment)
Fixed	inputRate	The sampling rate for the input. The effective input rectangle (in normal sampling coordinates) is **inRect** * **inputRate** (i.e., 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.
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 8–1: 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 structure of the input data when **wantsAbsolute** is 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** is 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 it is zero, you should assume that the plug–in host has not set it.
int32	inPlaneBytes	The step from plane to plane in the input. Normally one, but this changes if the plug–in uses the layout options. If it is zero, you should assume that the plug–in host has not set it.
int32	outColumnBytes	The output equivalent of the previous two fields.
int32	outPlaneBytes	
These fields are new in version 3.0.4.		
ImageServicesProcs *	imageServicesProcs	This is a pointer to the image services callback suite. See chapter 3 for details.
int16	inTileHeight	Tiling for the input.
int16	inTileWidth	
Point	inTileOrigin	
int16	absTileHeight	Tiling for the absolute data.
int16	absTileWidth	
Point	absdTileOrigin	
int16	outTileHeight	Tiling for the output.
int16	outTileWidth	
Point	outTileOrigin	
int16	maskTileHeight	Tiling for the mask.
int16	maskTileWidth	
Point	maskTileOrigin	

Table 8–1: FilterRecord structure (Continued)

Type	Field	Description
char[94]	reserved	These bytes are set to zero by the host for future expansion of the plug–in standard. Must not be used by plug–ins.

9 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 is shipped with several file format modules to read and write different formats including GIF, MacPaint, and BMP.

Acquire and export modules may also be used to read and write files. You should create a format module rather than an acquire and/or export module if you want your users to treat your files in the same fashion as native Photoshop files. In particular you should use a format module if:

- 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.
- You want users to be able to double–click a document to launch Photoshop (Mac OS), or associate your file extension with the Photoshop application (Windows).

You may not want to use a format module if:

- With respect to Photoshop, your file format is read–only or write–only.
- The image compression and/or color space conversion necessary for your file format would result in unacceptable image degradation if users read and saved repeatedly.

Under Mac OS, the code resource and file type for format modules is '8BIF'. Under Windows, the file extension is .8BI.

Sample plug–in

Sample Format is a sample format module. This module is written to use the new AdvanceStateProc callback, introduced in Photoshop 3.0.

Format module operations

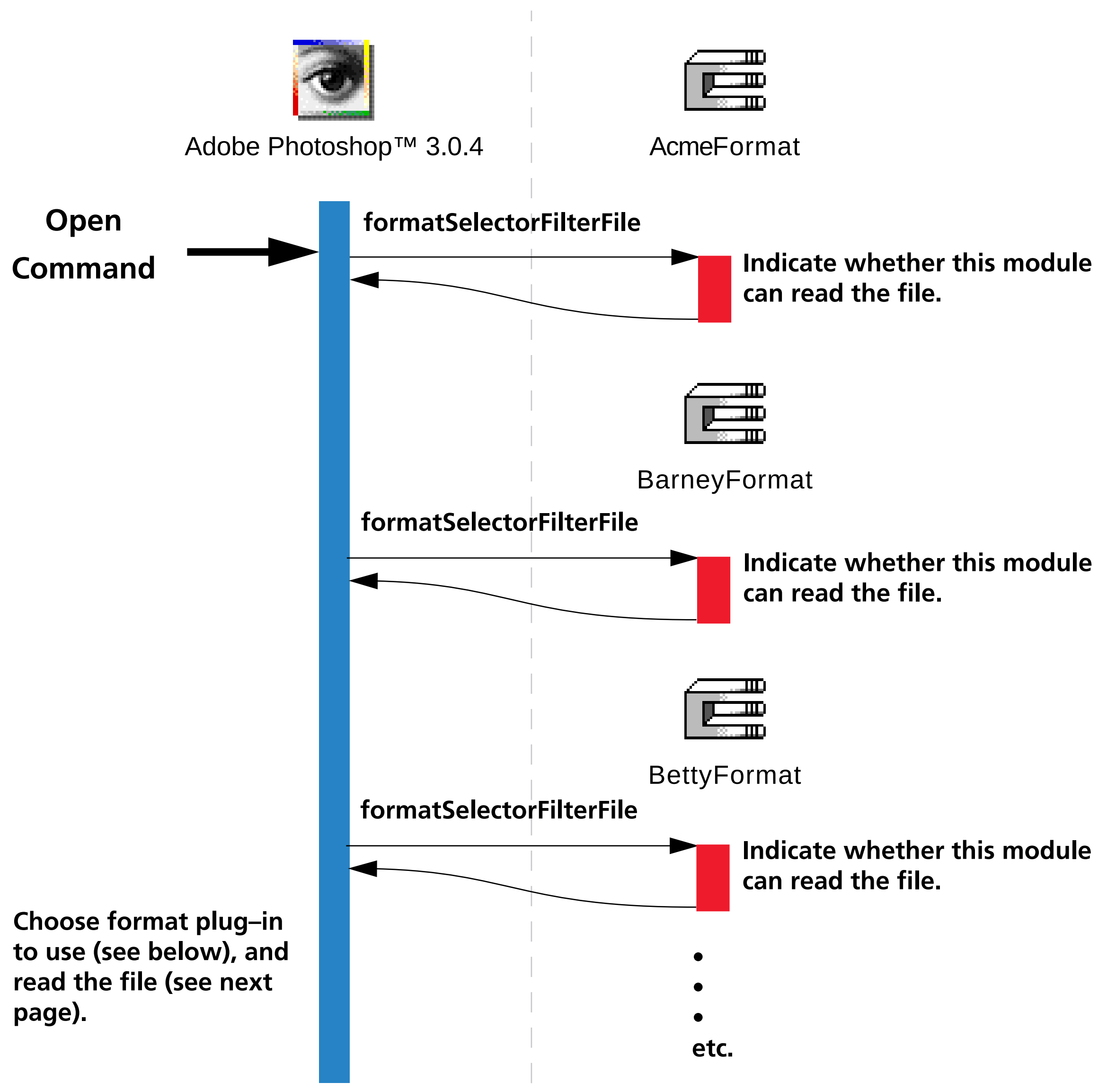
File format plug–in modules have two main functions: reading an image from a file, and writing an image to a file. Reading a file is a two step process:

- The **formatSelectorFilterFile** selector 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.
- The **read sequence** is used to read image files.

Writing a file consists of three sequences:

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

Reading a file (file filtering)


Adobe Photoshop™ 3.0.4
AcmeFormat
Open Command
formatSelectorFilterFile
Indicate whether this module can read the file.
BarneyFormat
formatSelectorFilterFile
Indicate whether this module can read the file.
BettyFormat
formatSelectorFilterFile
Indicate whether this module can read the file.
Choose format plug–in to use (see below), and read the file (see next page).
etc.

When the user selects a file with the Open... command from the file menu, there may be one or more format module that lists the file type (Mac OS) or file extension (Windows) as a supported format. For each such plug–in module, Photoshop will call the plug–in with a **formatSelectorFilterFile** selector. 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;
```

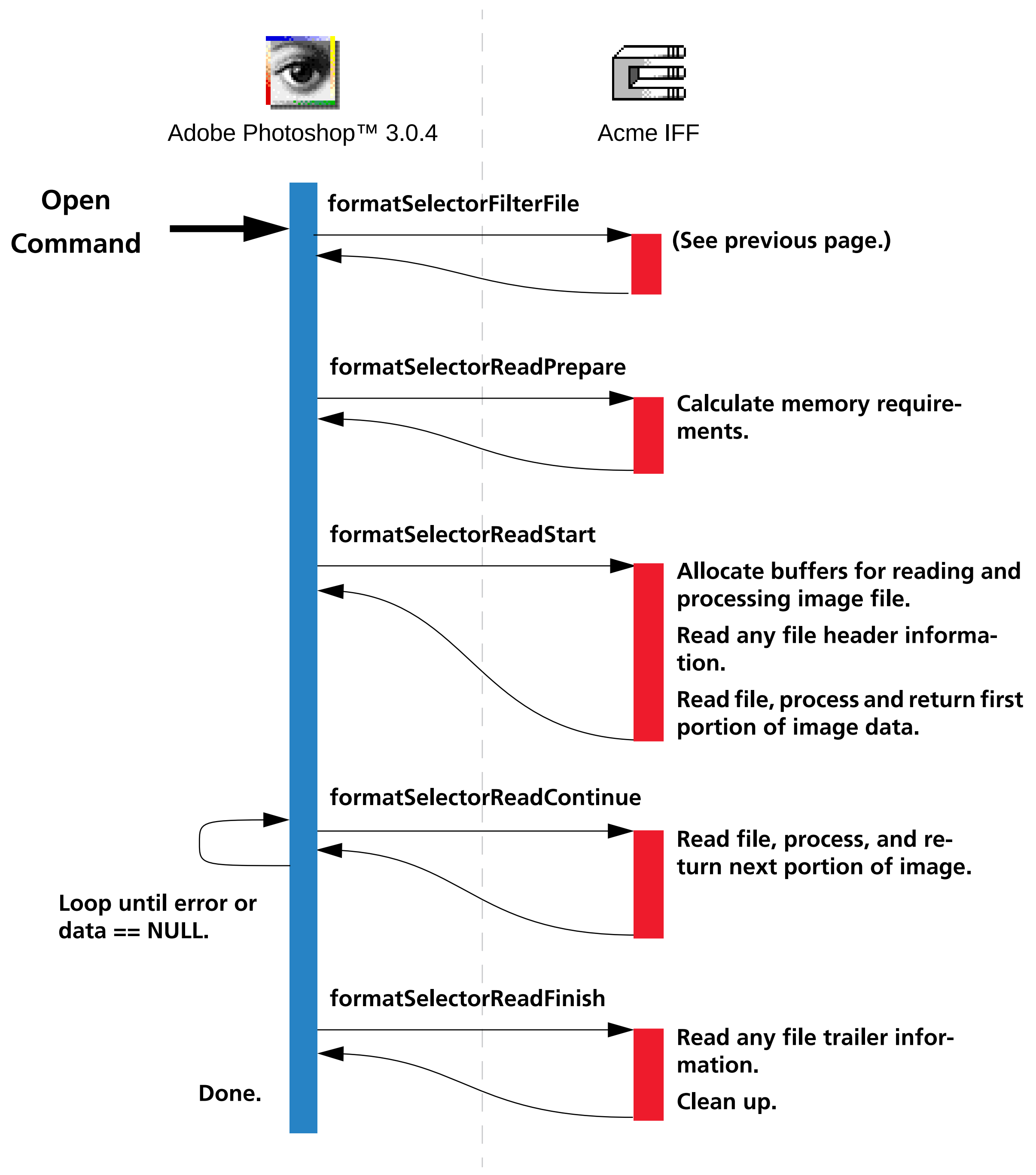
If more than one format module can read the file, Photoshop uses the following priority scheme to determine which plug–in module to use:

(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)


Adobe Photoshop™ 3.0.4
Acme IFF
Open Command
formatSelectorFilterFile
(See previous page.)
formatSelectorReadPrepare
Calculate memory requirements.
formatSelectorReadStart
Allocate buffers for reading and processing image file.
Read any file header information.
Read file, process and return first portion of image data.
formatSelectorReadContinue
Read file, process, and return next portion of image.
Loop until error or data == NULL.
formatSelectorReadFinish
Read any file trailer information.
Clean up.
Done.

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.

You should initialize **imageMode**, **imageSize**, **depth**, **planes**, **imageHRes** and **imageVRes**. If an indexed color image is being opened, you should also set **redLUT**, **greenLUT** and **blueLUT**. 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. See chapter 10 for more information about Photoshop image resources.

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**. 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. Here are two examples.

(1) To return just the red plane of an RGB color image, **loPlane** and **hiPlane** should be set to 0, **colBytes** should be set to 1, and **rowBytes** should be set to the width of the area being returned (**planeBytes** is ignored in this case, since **loPlane** == **hiPlane**).

(2) To return the RGB data in interleaved form (RGBRGB...), **loPlane** should be set to 0, **hiPlane** to 2, **planeBytes** to 1, **colBytes** to 3, and **rowBytes** to 3 times the width of area being returned.

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** to NULL.

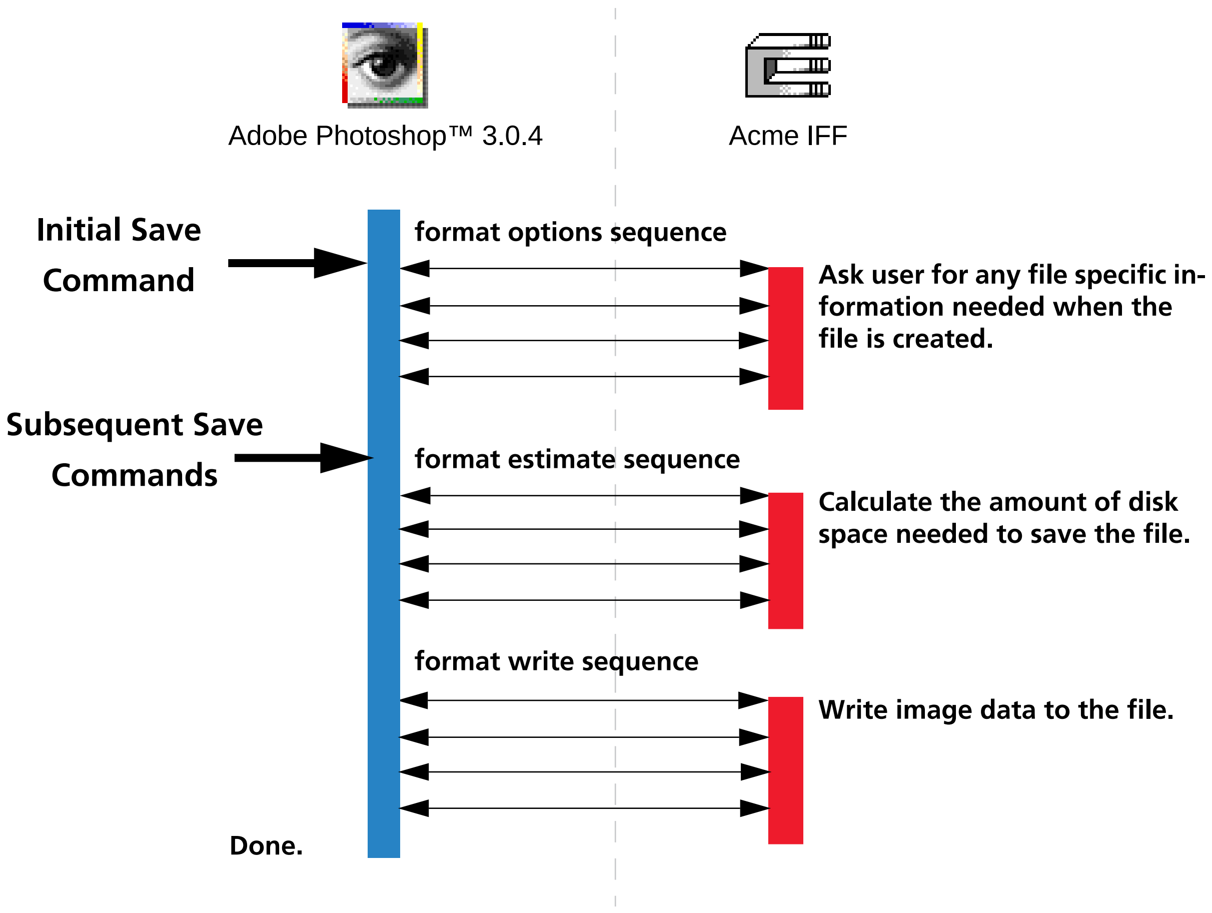
formatSelectorReadFinish

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 format-SelectorReadContinue 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 (Mac OS) or Escape (Windows) while processing the results of a formatSelectorReadContinue call, it will call the formatSelectorReadFinish routine. Be careful here, since normally the plug–in would be expecting another formatSelectorReadContinue call.

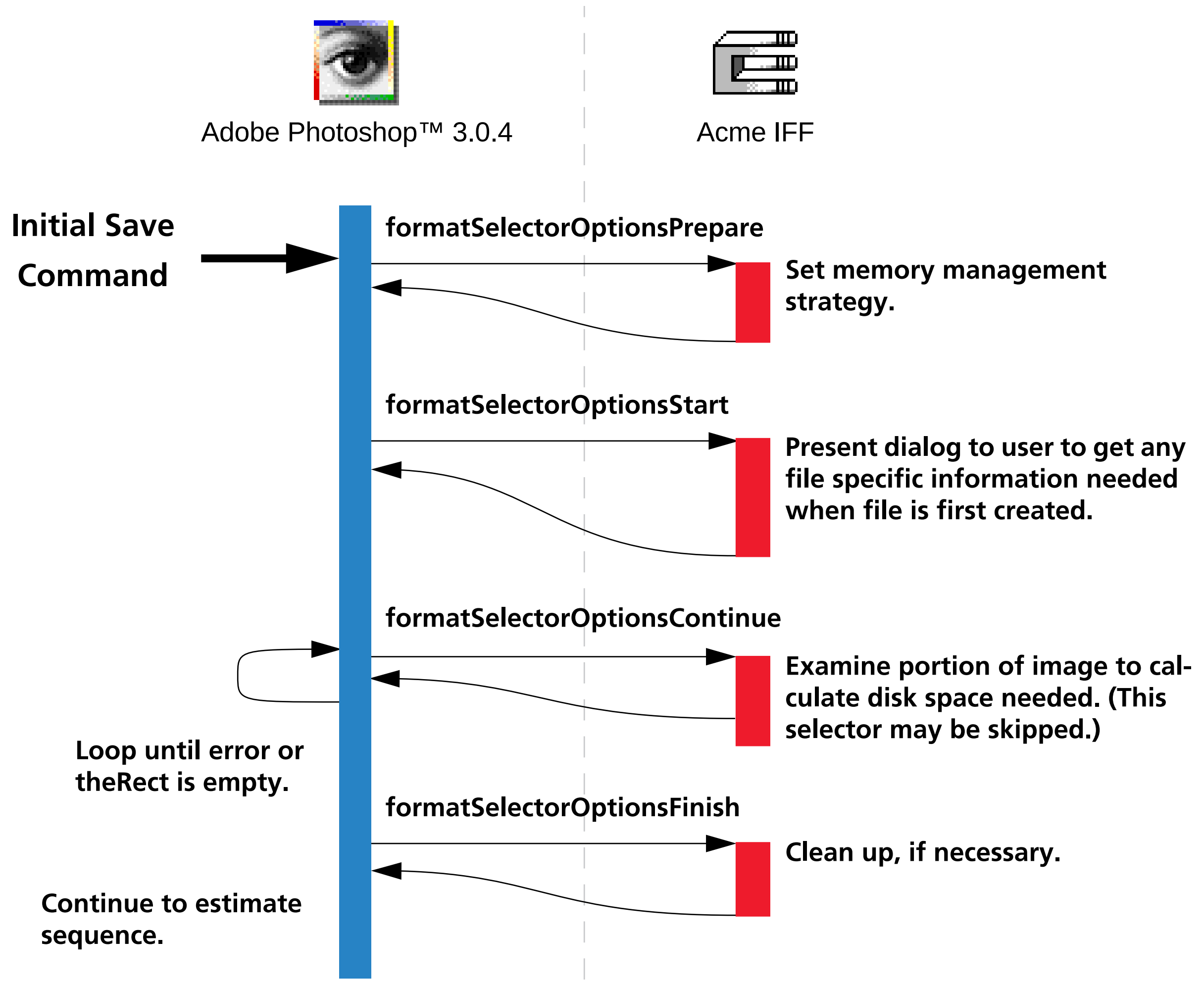
Writing a file


Adobe Photoshop™ 3.0.4
Acme IFF
Initial Save Command
Subsequent Save Commands
format options sequence
Ask user for any file specific information needed when the file is created.
format estimate sequence
Calculate the amount of disk space needed to save the file.
format write sequence
Write image data to the file.
Done.

Writing a file involves either two or three distinct sequences, each similar in structure. When a document is first saved, Photoshop calls your format plug–in module with the **options sequence,** followed by the **estimate sequence** and the **write sequence**. After a document has been saved once, each time the user saves the file again, only the estimate and write sequences are called.

These sequences are discussed in more detail on the following pages.

Writing a file (options sequence)


Adobe Photoshop™ 3.0.4
Acme IFF
Initial Save Command
formatSelectorOptionsPrepare
Set memory management strategy.
formatSelectorOptionsStart
Present dialog to user to get any file specific information needed when file is first created.
formatSelectorOptionsContinue
Examine portion of image to calculate disk space needed. (This selector may be skipped.)
Loop until error or theRect is empty.
formatSelectorOptionsFinish
Clean up, if necessary.
Continue to estimate sequence.

formatSelectorOptionsPrepare

The formatSelectorOptionsPrepare selector call 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

The formatSelectorOptionsStart selector call 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.

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

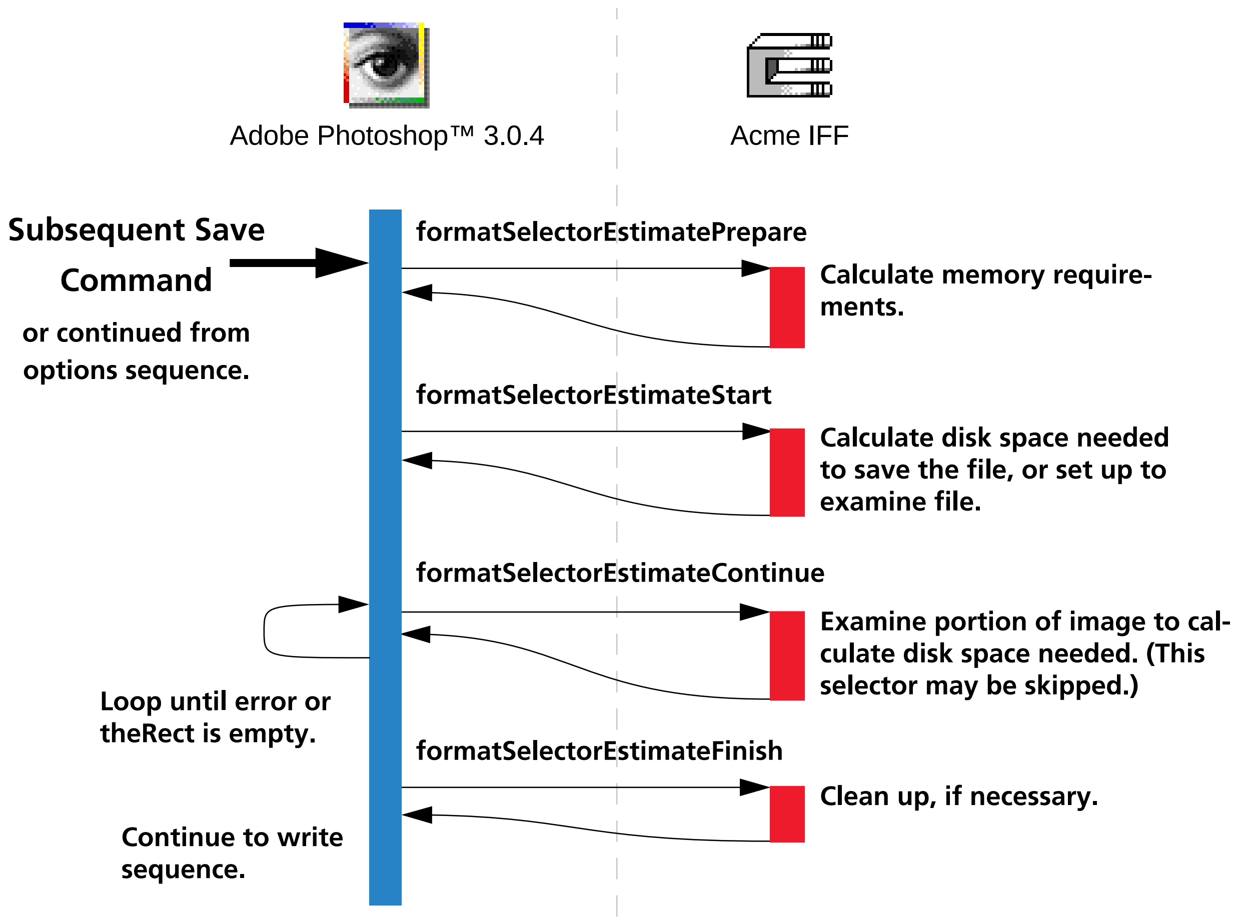
If the **data** field in the FormatRecord is set to NULL in the formatSelectorOptionsStart call, this selector will not be called at all. Otherwise, your plug–in can request parts of the image from which you determine whether your plug–in module can store the file. Refer to formatSelectorWriteStart and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


Adobe Photoshop™ 3.0.4
Acme IFF
Subsequent Save Command
or continued from options sequence.
formatSelectorEstimatePrepare
Calculate memory requirements.
formatSelectorEstimateStart
Calculate disk space needed to save the file, or set up to examine file.
formatSelectorEstimateContinue
Examine portion of image to calculate disk space needed. (This selector may be skipped.)
Loop until error or theRect is empty.
formatSelectorEstimateFinish
Clean up, if necessary.
Continue to write sequence.

formatSelectorEstimatePrepare

The formatSelectorWritePrepare selector call 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** to 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

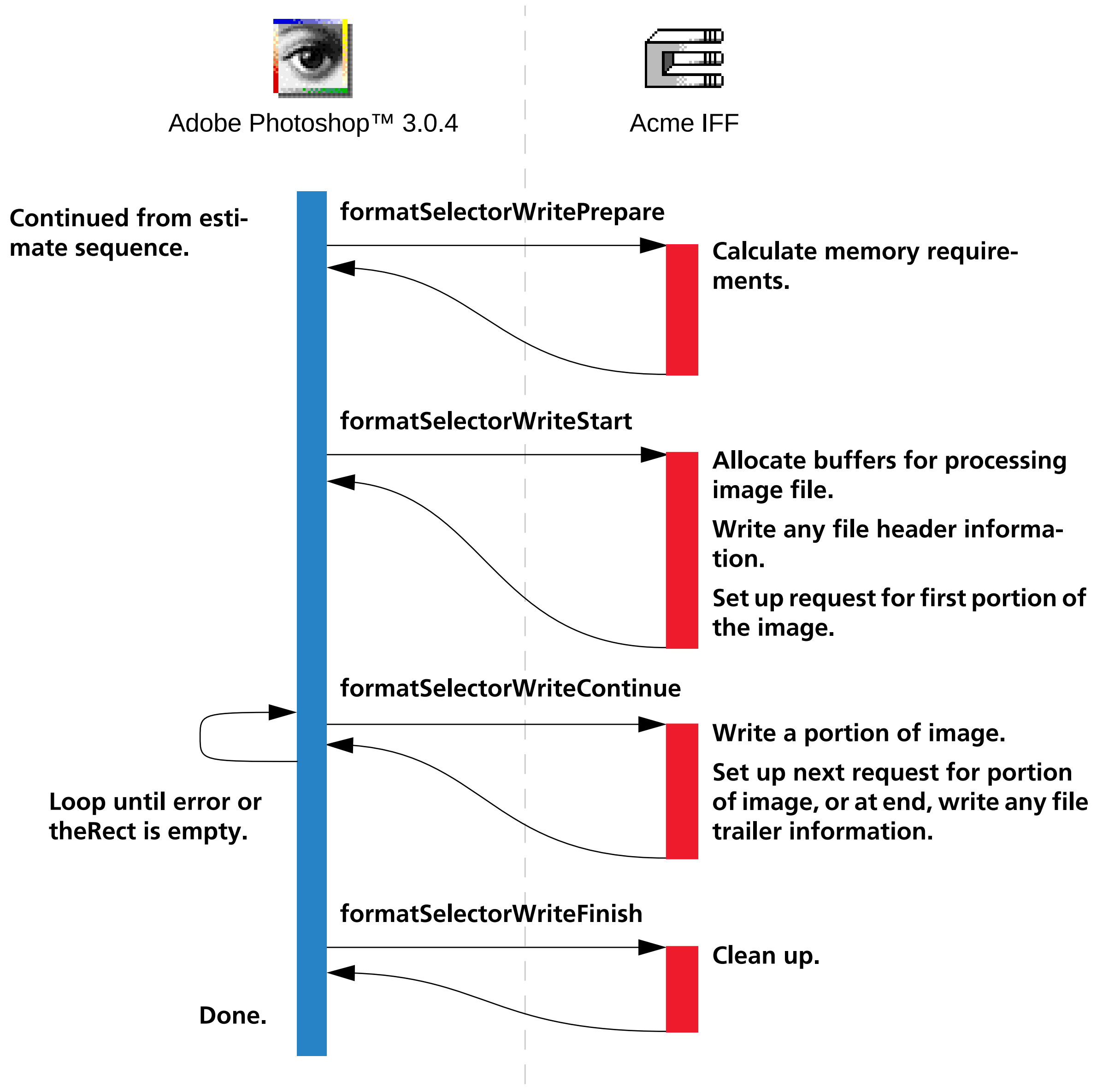
If the **data** field in the FormatRecord is set to NULL in the formatSelectorEstimateStart call, this selector will not be called at all. Otherwise, your plug–in can request parts of the image from which you can compute the minimum and maximum bytes to store the file. Refer to formatSelectorWriteStart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


Adobe Photoshop™ 3.0.4
Acme IFF
Continued from estimate sequence.
formatSelectorWritePrepare
Calculate memory requirements.
formatSelectorWriteStart
Allocate buffers for processing image file.
Write any file header information.
Set up request for first portion of the image.
formatSelectorWriteContinue
Write a portion of image.
Set up next request for portion of image, or at end, write any file trailer information.
Loop until error or theRect is empty.
formatSelectorWriteFinish
Clean up.
Done.

formatSelectorWritePrepare

The formatSelectorWritePrepare selector call 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. The area of the image being requested from the plug–in host is specified by **theRect**, **loPlane**, and **hiPlane**. The actual pixel data is pointed by **data**.

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 module. Here are two examples:

(1) To return just the red plane of an RGB color image, **loPlane** and **hiPlane** should be set to 0, **colBytes** should be set to 1, and **rowBytes** should be set to the width of the area being returned (**planeBytes** is ignored in this case, since **loPlane == hiPlane**).

(2) To return the RGB data in interleaved form (RGBRGB...), **loPlane** should be set to 0, **hiPlane** to 2, **planeBytes** to 1, **colBytes** to 3, and **rowBytes** to 3 times the width of area being returned.

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

The formatSelectorWriteFinish selector 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 (Mac OS) or Escape (Windows) while processing the results of a formatSelectorWriteContinue call, it will call the formatSelectorWriteFinish routine. Be careful here, since normally the plug–in would be expecting another formatSelectorWriteContinue call.

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.

```
#define formatBadParameters –30500  // a problem with the module interface
#define formatCannotRead    –30501  // a problem interpreting file data
```

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 options, estimate, or write selectors will be called.

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 9–1: 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 for more details.
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. On Windows, this is the file handle of the file returned by OpenFile ( ) API.
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. On Windows, this field is undefined.
int16	imageMode	The formatSelectorReadStart routine should set this field to inform Photoshop what mode image is being acquired (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 formatSelector-WriteStart.

Table 9–1: 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 formatSelector-OptionsStart, 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 modules (and acquire 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 formatSelector-WriteStart.
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, respectively.
int16	hiPlane	
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.

Table 9–1: FormatRecord structure (Continued)

Type	Field	Description
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] to 3, planeMap [1] to 0, planeMap [2] to 1, and planeMap [3] to 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 is 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 (LSB = bit 0) is 1, the mode is supported. This field can be used by plug–ins to disable reading unsupported file formats.

Table 9–1: 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 formatSelector-ReadStart call, then query the user as appro-priate and set up this field to store a handle containing the information necessary to read the file without querying the user for addi-tional 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 dupli-cated when we duplicate the image format information for a document. On all formatSe-lectorOptions 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 call-backs 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.
DisposePIHan-dleProc	hostDisposeHdl	This is the same as the DisposePIHandle call-back 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. During the read sequence, Photoshop checks this field after each call to formatSelectorRead and formatSe-lectorContinue and the first time it is non–NULL parses the handle as a block of image resource data for the current document.
int32	imageRsrcSize	This is the size of the handle in imageRsrc-Data. It is really only relevant during the esti-mate sequence when it is provided instead of the actual resource data.
PlugInMonitor	monitor	This field contains the monitor setup informa-tion for the host. See Appendix A for more details.
void *	platformData	This field contains a pointer to platform spe-cific 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 ProcessEv-ent callback documented in chapter 3. It con-tains NULL if the callback is not supported.

Table 9–1: 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 in 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. For more information, see chapter 3.
These fields are new in version 3.0.4 of Adobe Photoshop.		
PropertyProcs *	propertyProcs	A pointer to the property callback suite. See chapter 3 for details.
int16	tileWidth	The width of the tiles. Zero if not set.
int16	tileHeight	The height of the tiles. Zero if not set.
int16	tileOrigin	The origin point for the tiles.
char[220]	reserved	Set to zero by the host for future expansion of the plug–in standard. Must not be used by plug–ins.

10 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 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 10–1.

Table 10–1: 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 10–2. Not all file formats use all ID's. Some information may be stored in other sections of the file.

Table 10–2: Image resource ID's

ID	Description
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)	Resolution information. See appendix A for description of the ResolutionInfo structure.
0x03EE (1006)	Names of the alpha channels as a series of Pascal strings.
0x03EF (1007)	Display information for each channel. See appendix A for a description of the DisplayInfo structure.
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 11.
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.

Table 10–2: Image resource ID's

ID	Description
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.
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 (always zero), 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 “Paths To Illustrator” 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.

On Intel processors (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 data record it is. Under Windows, you should swap the bytes before accessing it as a short (int16).

Table 10–3: 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 all zeroes which indicates that paths use even/odd rule. 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 edits the other one 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

Macintosh file type:	'8BPS'
Windows file extension:.PSD

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

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 Adobe Fetch, a 'pnot' 0 resource 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 a 'STR#' 128 resource and the caption text is stored in a 'TEXT' 128 resource. For more information on the format of these resources see *Inside Macintosh: QuickTime Components* and the *Adobe Fetch Awareness Developer's Toolkit.*

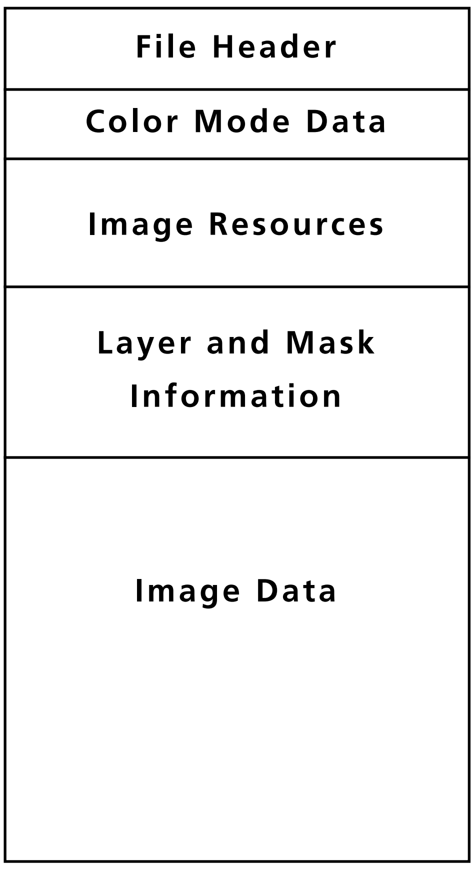
All of the data from Photoshop's File Info dialog is stored in an 'ANPA' 10000 resource. The data in this resource is stored as an IPTC–NAA record 2 and should be readable by various tools from Iron Mike. For more information on the format of this resource contact:

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

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

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 10–4: 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.

Table 10–4: File header (Continued)

Length	Name	Description
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 Color = 2 RGB Color = 3 CMYK Color = 4 Multichannel = 7 Duotone = 8 Lab Color = 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 10–5: 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 10–6: 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 10–7: Layer and mask information

Length	Name	Description
4 bytes	Length	Length of the miscellaneous information section.
Variable	Layers	Layer info. See table 10–10.
Variable	Masks	One or more layer mask info structures. See table 10–13.

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 10–8: Image data

Length	Name	Description
2 bytes	Compression	Compression method. Raw data = 0, RLE compressed = 1.
Variable	Data	The image data.

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 10–9. The second part of this section contains layer mask data, which is described in table 10–16.

Table 10–9: 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 10–10.
Variable	Pixel data	Channel image data for each channel in the order listed in the layers structure section. See table 10–15.

Table 10–10: 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 10–11).

Table 10–11: 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 10–12.
4 bytes	Blend mode signature	Always ‘8BIM’.

Table 10–11: Layer records (Continued)

Length	Name	Description
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
1 byte	Flags	bit0: transparency protected bit1: 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 10–13.
Variable	Layer blending ranges	See table 10–14.
Variable	Layer name	Pascal string, padded to a multiple of 4 bytes.

Table 10–12: 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 10–13: 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	

Table 10–13: Layer mask data

Length	Name	Description
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 10–14: 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	
4 bytes	First channel destination range	
4 bytes	Second channel source range	
4 bytes	Second channel destination range	
...	...	...
4 bytes	Nth channel source range	
4 bytes	Nth channel destination range	

Table 10–15: 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 ((Layer Bottom – Layer Top) * (Layer Right – Layer Left)). If the compression code is 1, the image data starts with the byte counts for all the scan lines in the channel (Layer Bottom – Layer Top), 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 data since the Layers Size is odd, a pad byte will be inserted.

Table 10–16: Layer mask data

Length	Name	Description
2 bytes	Overlay color space	
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 EPS files

Photoshop 3.0 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>
```

<length> is the length of the image resource data.

<hex data> is the 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>"
```

<columns> is the width of the image in pixels.

<rows> is the height of the image in pixels.

<depth> is the number of bits per channel. Must be 1 or 8.

<mode> is the image mode. 1 for bitmap and gray scale images (determined by depth), 2 for Lab images, 3 for RGB images, and 4 for CMYK images.

<pad channels> is the number of other channels stored in the file, which are ignored when reading. (Photoshop uses this to include a gray scale image that is printed on non–color PostScript printers).

<block size> is the number of bytes per row per channel. This will be equal to (<columns> * <depth> + 7) / 8 if the data is stored in line–interleave format (or if there is only one channel), or equal to 1 of the data is interleaved.

<binary/hex> is 1 if the data is in binary format, and 2 if the data is in hex format.

<data start> contains the entire PostScript line immediately preceding the image data. (This entire line should not occur elsewhere in the PostScript header code, but it may occur at part of a line.)

Filmstrip files

Adobe Premiere 2.0 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 Premiere 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 10–17: 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 (width * height * 4) bytes. If the data is being placed into a GWorld (Mac OS), 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 a block of ((width * (height+leading) * 4) – sizeof(FilmStripRec)) bytes. 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 film-strips 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
(206) 628–5693

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)

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 10–16 describes the standard TIFF tags and tag values that Photoshop 3.0 is able to read and write.

TIFF files under 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 Adobe Fetch, a 'pnot' 0 resource 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 a 'STR#' 128 resource and the caption text is stored in a 'TEXT' 128 resource. For more information on the format of these resources see *Inside Macintosh: QuickTime Components* and the *Adobe Fetch Awareness Developer's Toolkit.*

All of the data from Photoshop's File Info dialog is stored in an 'ANPA' 10000 resource. The TIFF file also contains a 'STR ' -16396 resource that contains the string “Adobe Photoshop™ 3.0” which indicates the application that created the TIFF file.

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

Table 10–18: 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

11

Load File Formats

Besides documents that the user creates in Adobe Photoshop (discussed in chapter 10), 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 at a later time, even for use in a different image. 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, but not all, of the files have version numbers written as short integers in the first two bytes of the file.

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

Note that this requires consistent byte ordering between the all platforms when reading and writing these files. Photoshop stores multi–byte values with the high–order bytes first (big–endian, used on 680x0 systems), the reverse of the way this is done on Intel platforms (little–endian).

Arbitrary Map

Macintosh file type: '8BLT'
Windows file extension:.AMP

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

There is no version number written in the file. The file must be an even multiple of 256 bytes long.

Each 256 bytes is a lookup table, where the first byte corresponds to zero in the image data and the last byte to 255 in the image data. A “null” table that has no effect on an image is a linear table of bytes from 0 to 255.

If there is one table in the file, Photoshop applies it to the master composite channel, if the image has one, or to the single active channel if there is only one. If there is no composite channel, but more than one active channel, the load operation will have no effect. If the file has exactly three tables then it is assumed to represent an RGB lookup table and they are applied to the first channels in the image (the master composite map is untouched). If there is a single active channel, then the RGB lookup table is converted to gray-scale and the result is applied to the active channel. In any other case, the first map is treated as a master and the remainder are applied to the image channels in turn (i.e. the second map is associated with the first channel, the third map with the second channel, etc.)

Photoshop handles single active channels in a special fashion. When saving the map applied to a single channel, only one map is written to the file. Similarly, when reading a map file for application to a single active channel, the master map 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

Macintosh file type: '8BBR'
Windows file extension:.ABR

Brushes settings files are loaded and saved in Photoshop’s Brushes palette. These are typically stored in the “Goodies:Brushes & Patterns” sub–folder (Mac OS), or BRUSHES sub–directory (Windows).

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Count (2 bytes)
A short integer indicating how many brushes are in the remainder of the file.

3. Brushes (variable)
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.

Each brush contains the following components:

a. type (2 bytes)
A short integer indicating the type of brush. A value of 1 means a computed brush, a value of 2 means a sampled brush. Other values are currently undefined.

b. size (4 bytes)
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.

c. data (size bytes)
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.

Computed brushes:

i. miscellaneous (4 bytes): a long value which is ignored.

ii. spacing (2 bytes): a short integer ranging from 0 to 999 (0 means no spacing)

iii. diameter (2 bytes): a short integer ranging from 1 to 999

iv. roundness (2 bytes): a short integer ranging from 0 to 100

v. angle (2 bytes): a short integer ranging from –180 to 180

vi. hardness (2 bytes): a short integer ranging from 0 to 100

Sampled brushes:

i. miscellaneous (4 bytes): a long value which is ignored.

ii. spacing (2 bytes): a short integer ranging from 0 to 999 (0 means no spacing)

iii. anti–aliasing (1 byte): indicates whether the brush is to be anti–aliased when applied; 0 means no anti–aliasing. (Note that brushes with sampled data size either taller or wider than 32 pixels will not be anti–aliased by Photoshop in any event.)

iv. bounds (8 bytes): a rectangle, four short integers giving the bounds of the sampled tip data (in the order top, left, bottom, right)

v. bounds2 (16 bytes): a rectangle, exactly repeating the previous bounds entry, but in four long integers instead of four short integers.

vi. depth (2 bytes): depth of the sampled data, which is always 8

vii. image data (variable): if the bounds are taller than 16384, the data is broken into 16384–line chunks. Each chunk is streamed as follows:

   a. compression (2 bytes): two values are currently defined: 0 = Raw Data, 1 = RLE compressed

   b. data (variable): the brush tip image data is a single plane of grayscale data, stored in scanline order, with no pad bytes.

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

If the compression code is 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

Macintosh file type: '8BCT'
Windows file extension:.ACT

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

There is no version number written in the file. The file is exactly 768 bytes long.

This file contains 256 RGB colors, starting with the first color in the table (index 0), with 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

Macintosh file type: '8BCO'
Windows file extension:.ACO

Colors files are loaded and saved in Photoshop’s Colors palette. These are typically stored in the PALETTES sub–directory (Windows)

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Count (2 bytes)
A short integer indicating how many colors are in the remainder of the file.

3. Colors (Count * 10 bytes)
Each color is ten bytes in size, and is made up of the following subsections:

a. color space ID (2 bytes)
A short integer indicating the color space the color belongs to as shown in table 1.

b. color data (8 bytes)
Four short integers (possibly unsigned) that are the actual color data. If the color does not require four values to specify, the extra values are undefined and should be written as zeroes. The most basic color spaces are outlined below.

Table 11–1: Color data

Color ID	Name	Description
0	RGB	The first three values in the color data are, respectively, the color’s red, green, and blue components. They are full unsigned 16–bit values as in Apple’s RGBColor data structure (e.g. pure red is defined as 65535, 0, 0).
1	HSB	The first three values in the color data are, respectively, the color’s hue, saturation, and brightness components. They are full unsigned 16–bit values as in Apple’s HSV-Color data structure (e.g. pure red is defined as 0, 65535, 65535).
2	CMYK	The four values in the color data are, respectively, the color’s cyan, magenta, yellow, and black components. They are full unsigned 16–bit values, with 0 representing 100% ink (e.g. pure cyan is defined as 0, 65535, 65535, 65535).
7	Lab	The first three values in the color data are, respectively, the color’s lightness, a chrominance, and b chrominance components. The lightness component is a 16–bit value ranging from 0 to 10000. The chromanance components are each 16–bit values ranging from –12800 to 12700. Gray values are represented by chrominance components of 0 (e.g. pure white is defined as 10000, 0, 0).
8	Grayscale	The first value in the color data is the gray value; it ranges from 0 to 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 2 gives the color space IDs currently defined by Photoshop for some custom color spaces:.

Table 11–2: Custom color spaces

Color ID	Name
4	FOCOLTONE COLOUR SYSTEM
10	HKS colors (European Photoshop only)
3	PANTONE MATCHING SYSTEM
6	TOYO 88 COLORFINDER 1050
5	TRUMATCH color

Command Settings File

Macintosh file type: '8BFK'
Windows file extension:.ACM

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.

1. Version (2 bytes)
Equal to 2, written as a short integer.

2. Count (2 bytes)
The number of command records that follow. There are no pad bytes between records.

3. Command Records (variable)
The remainder of the file contains the Command records, one after the other. Each one is composed of the following:

a. Command ID (4 bytes)
This field is obsolete and must be set to zero.

b. Function Key ID (2 bytes)
This is an integer ranging from –15 to 15. Positive numbers map directly onto the numbered function keys (F1, F2, etc.) that are present on many personal computer keyboards. Negative numbers indicate that the shift key must be used as well for the keyboard 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 calling up Help. These numbers should be unique across all entries in a Commands file, however Photoshop will ignore duplicates.

c. Color Index (2 bytes)
Each command button can be assigned a color with which its background will be tinted when drawn. There are eight predefined colors, with matching values as follows: 0 = None (button drawn in black and white), 1 = Red, 2 = Orange, 3 = Yellow, 4 = Green, 5 = Blue, 6 = Purple, 7 = Gray.

d. Title Matching Flag (1 byte)
If set to 1, this boolean flag indicates that the button title should automatically be updated to match the command's current menu item text. 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. If set to 0, the button title has been changed from the menu item text by the user and shouldn't change unless changed by the user again.

e. Button Title (variable)
This is the title of the button that will be drawn on the Command palette. It usually matches the corresponding menu item text. It is stored as a Pascal–style string, with no pad bytes.

f. Command Key (variable)
This is the key for finding the menu item in Photoshop's menus. To distinguish menu items from each other, which could be duplicated on different menus, a key may include the title of the menu itself followed by a colon (e.g. "Mode:RGB Color"). This text is displayed in the options dialog for the button, but not on the Commands palette itself. (Note that even if the Title

Matching flag is turned on, the title of the button text on the screen never contains the menu title qualifier.) It is stored as a Pascal–style string, with no pad bytes.

Curves

Macintosh file type: '8BSC'
Windows file extension:.CRV

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

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Count (2 bytes)
A short integer indicating how many curves are in the file. Must be in the range 1 to 27.

3. Curves (variable)
The remainder of the file contains the curves, one after the other.

Each curve is written as follows (i.e. each curve is made up of the following subsections):

a. point count (2 bytes)
A short integer in the range 2 to 19 indicates how many points are in the curve.

b. curve points (point count * 2 bytes)
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.
Hence 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 that 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.)

Generally, the first of the curves is a master curve that applies to all of the composite channels in a composite image mode (e.g. the Red, Green, and Blue channels are all modified by the master curve for an RGB document). The remaining curves apply to the active channels individually; the second curve applies to channel one (if it is an active channel), the third curve to channel two, etc., up until the seventeenth curve, which applies to channel sixteen. The exception to this, and the reason there are up to nineteen curves, is when the original image is indexed color. In this case three curves are created for the red, green, and blue portions of the image's color table, and they replace the curve that represents the first channel of the image. This adds two curves for indexed images, and so for indexed color images any alpha channel that is active corresponds to its channel number plus three (e.g. if channel two is active it corresponds to curve number 5).

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 slot, 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.

Note that Photoshop 3.0 can write Curves files that Photoshop 2 will not be able to read, because Photoshop 3.0's active channel support is different from Photoshop 2.0's, and there could be more active channels in a Curves dialog than 2.0 supported. Photoshop 3.0 will always write at least five curves to a curves file, for maximum compatability with version 2.0. However, beyond the curve for the fourth channel, it does not write null curves past the last non–null curve that has been specified in the dialog. The presence of extraneous null curves will not affect a load operation.

Also note that it is possible to create a Curves load file with Photoshop 3.0 that cannot be read by Photoshop 2.5; Photoshop 3.0 allows a maximum of 24 channels per document, Photoshop 2.5 allows 16. Such use of the Curves function is rare, however.

Duotone options

Macintosh file type: '8BDT'
Windows file extension:.ADO

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

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Count (2 bytes)
A short integer indicating how many plates are in the duotone spec: 1 for monotones, 2 for duotones, 3 for tritones, 4 for quadtones. Must be in the range 1 to 4.

3. Ink Colors (4 * 10 bytes)
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 streamed in the same fashion as in the Colors load file, and consists of the following subsections:

a. color space (2 bytes)
A short integer indicating the color space the color is in.

b. color data (8 bytes)
Four short integers (possibly unsigned) that are the actual color data.

Please refer to the Colors file format for details on the contents of the color records.

4. Ink Names (4 * 64 bytes)
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 characters in the string. Names may not be more than 63 characters in length. Each name is padded to occupy 64 bytes including the initial length byte. Any names beyond the last plate specified by Count should be the empty string (size = 0).

5. Ink Curves (4 * 28 bytes)
Four ink curves, regardless of the number of plates. Each curve has the following subsections:

a. transfer curve (26 bytes)
An array of 13 short integers, each ranging from 0 to 1000 (representing 0.0 to 100.0). In addition, all but the first and last value may be –1 (representing no point on the curve). Hence a null transfer curve looks like this: 0, –1, –1, –1, –1, –1, –1, –1, –1, –1, –1, –1, 1000.

b. override (2 bytes)
For compatability with Photoshop 2.0, this short integer should be 0. It is ignored by Photoshop 3.0.

Any curves beyond the last plate specified by Count should be equal to the null curve.

6. Dot Gain (2 bytes)
For compatability with Photoshop 2.0, this short integer should be 20. It is ignored by Photoshop 3.0.

7. Overprint Colors (11 * 10 bytes)
Eleven ink colors, regardless of the number of plates. The number of defined overprints depends on the number of plates, Count. For monotones, there are no overprint colors. For duotones, there is one overprint color. For tritones, there are four overprint colors. For quadtones, there are 11 overprint colors. The contents of the colors beyond the last defined overprint are undefined. Each color is streamed in the same fashion as in the Colors load file, and consists of the following subsections:

a. color space (2 bytes)
A short integer indicating the color space the color is in.

b. color data (8 bytes)
Four short integers (possibly unsigned) that are the actual color data.

Halftone screens

Macintosh file type: '8BHS'
Windows file extension:.AHS

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

1. Version (2 bytes)
Equal to 5, written as a short integer.

2. Screens (4 * 18 bytes)
Four screen descriptions, each of which has the following subsections:

a. frequency value (4 bytes)
This ink's screen frequency, in lines per inch. This is a binary fixed point value with sixteen bits representing each of the integer and fractional parts of the number. Values range from 1.0 to 999.999, with units in lpi (lines per inch).

b. frequency scale (2 bytes)
The units for the screen frequency. Line per inch = 1, lines per centimeter = 2. Does not affect the frequency value itself, merely the way the value will be displayed on the screen.

c. angle (4 bytes)
Angle for this screen, a binary fixed point value with sixteen bits representing each of the integer and fractional parts of the number. Values range from –180.0000 to 180.0000, measured in degrees.

d. shape code (2 bytes)
A code representing the shape of the halftone dots in this screen. Round = 0, Ellipse = 1, Line = 2, Square = 3, Cross = 4, Diamond = 6. Custom shapes are represented by a negative number. The absolute value of this number is the size in bytes of the custom Spot Function, which is outlined below.

e. miscellaneous (4 bytes)
For compatability, this should be set to 0. It is not currently used by Photoshop.

f. accurate screens (1 byte)
Boolean flag which is true (1) if accurate screens should be used, false (0) otherwise.

g. default screens (1 byte)
Boolean flag which is true (1) if printer's default screens should be used, false (0) otherwise.

3. Spot Functions (size is the sum of the absolute values of all negative shape codes)
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 (screen description 1's spot function, if it has one, followed by number 2's, etc.). The shape code for those screens that have custom functions provides enough information to separate the various functions and assign them.

Hue/Saturation

Macintosh file type: '8BHA'
Windows file extension:.HSS

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

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Mode (1 byte)
Photoshop's Hue/Saturation dialog has two overall modes: in one, the settings represent shifts in the image data's hue and saturation, in the other the entire image is colorized to a single hue. This byte is a boolean flag indicating whether the colorization data or the hue–adjustment data in the file should be used. If the byte is zero, the hue–adjustment data will be used. If the byte is non–zero (Photoshop writes it as a 1) the colorization data will be used. (Both sets of data are present, but only one is used depending on the value of this byte.)

3. Padding (1 byte)
This pad byte must be present but is ignored by Photoshop.

4. Colorization (6 bytes)
Three short integers representing colorization settings. All values are in the range –100 to 100. The first number is the hue in which the image data will be colorized; the user interface represents the range of values as –180 to 180, where the number represents the hue in the traditional HSB color wheel, with zero equal to red. The next number is the saturation, the third number is the lightness adjustment.

5. Hue–Saturation Settings (42 bytes)
This data consists of three sets of seven short integers; all values range from –100 to 100:

a. hue settings (14 bytes)
One master value and six other values. The first value is the master hue change. For RGB and CMYK images, the other six 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 as being in the range –60 to 60; the values are nevertheless stored as –100 to 100 and the slider will reflect each of the possible 201 values.) For Lab images, the first four of these values are applied to image pixels in the four Lab color quadrants (yellow, green, blue, magenta), and the other two values are ignored (Photoshop sets them to zero). (The values that are used range from –90 to 90 in the user interface.)

b. saturation settings (14 bytes)
Seven short integers representing the saturation adjustments. The first is a master value. The other six are applied to pixels using the same hue sextant or quadrant breakdown as for the hue adjusments; as before the last two are ignored for Lab documents.

c. lightness settings (14 bytes)
The last seven short integers are the lightness adjustments. The first is a master value. The other six are applied to pixels using the same hue sextant or quadrant breakdown as for the hue and saturation adjusments; as before the last two are ignored for Lab documents.

Ink colors setup

Macintosh file type: '8BIC'
Windows file extension:.API

Ink Colors settings files are loaded and saved in Photoshop’s Ink Colors Setup dialog, via the Preferences submenu.

1. Version (2 bytes)
Equal to 4, written as a short integer.

2. Ink Colors (27 * 2 bytes)
Nine short integer triples 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 triple is written in the order x (range: 0 to 10000, representing 0.0 to 1.0000), y (range: 1 to 10000, representing 0.0001 to 1.0000), Y (range: o to 20000, representing 0.00 to 200.00).

3. Gray Balance (4 * 2 bytes)
Four short integers specifying the gray color balance for Cyan, Magenta, Yellow, and Black. Each ranges from 50 to 200 (representing 0.5 to 2.00).

4. Dot Gain (2 bytes)
A short integers specifying the dot gain. Ranges from –10 to 40 (–10% to 40%).

Custom kernel

Macintosh file type: '8BCK'
Windows file extension:.ACF

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

There is no version number written in the file. The file is expected to be exactly 54 bytes long, representing 27 short integers.

1. Weights (50 bytes)
The first 25 values are the custom weights, applied to pixels offset from (–2, –2) to (2, 2) off of each image pixel. The values progress through horizontal offsets first, e.g. the first five values all represent a vertical offset of –2. Each value can range from –999 to 999.

2. Scale (2 bytes)
This value can range from 1 to 9999.

3. Offset (2 bytes)
This value can range from –9999 to 9999.

Levels

Macintosh file type: '8BLS'
Windows file extension:.ALV

Levels settings files are loaded and saved in Photoshop's Levels dialog. There are two versions of this file format. Photoshop 3.0 reads both but only writes version 2. Note that because the maximum number of channels that a document can contain was increased in Photoshop 3.0 (from 16 to 24), Photoshop 3.0 actually writes a longer Levels file than Photoshop 2.5. Photoshop 2.5 is still capable of reading these files, however, and will simply ignore the extra data.

1. Version (2 bytes)
Equal to 2, written as a short integer.

2. Levels Records (290 bytes)
This consists of 29 sets of levels. Each set of levels contains five short integers, in ten bytes.

The first number in a set is the input floor setting, and must range from 0 to 253. The second number is the input ceiling, and must range from 2 to 255. Third is the output value to which the input floor willbe matched. It can range from 0 to 255. Fourth is the ceiling output, also ranging from 0 to 255. The fifth value is the gamma to be applied to the image data. It ranges from 10 to 999 (representing the values 0.1 to 9.99).

The first set of levels are the master levels that apply to all of the composite channels in a composite image mode (e.g. the Red, Green, and Blue channels are all modified by the master levels settings for an RGB document).

The remaining sets apply to the active channels individually; the second set applies to channel one (if it is an active channel), the third set to channel two, etc., up until the 25th set, which applies to channel 24.

The exception to this is when the original image is indexed color. In this case three sets of levels are created for the red, green, and blue portions of the image's color table, and they replace the levels that represent the first channel of the image. This adds two sets of levels for indexed images, and so for indexed color images any alpha channel that is active corresponds to its channel number plus three (e.g. if channel two is active it corresponds to set number 5). The 28th and 29th sets are reserved for future use and should be set to zeroes.

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 slot, 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.

Monitor setup

Macintosh file type: '8BMS'
Windows file extension:.AMS

Monitor settings files are loaded and saved in Photoshop's Monitor Setup dialog, via the Preferences submenu.

1. Version (2 bytes)
Equal to 2, written as a short integer.

2. Gamma (2 bytes)
A short integer indicating the monitor's gamma. Must be in the range 75 to 300 (representing 0.75 to 3.00).

3. White Point (2 * 2 bytes)
Two short integers giving the monitor's white point: the first is the x value, ranging from 0 to 10000 (representing 0.0 to 1.0000), the second is the y value, ranging from 1 to 10000 (representing 0.0001 to 1.0000).

4. Phosphors (6 * 2 bytes)
Three sets of two integers giving the x–y coordinates of the red, green, and blue phosphors. First comes red x, then red y; then green x, etc. The x values range from 0 to 10000 (representing 0.0 to 1.0000); the y values range from 1 to 10000 (representing 0.0001 to 1.0000).

Replace color/Color range

Macintosh file type: '8BXT'
Windows file extension:.AXT

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

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Color Space (2 bytes)
A short integer indicating what space the color components are in. 7 indicates Lab color, 8 indicates Grayscale. No other values are supported.

3. Component Ranges (6 bytes)
These six unsigned byte values represent the range of colors within which a pixel’s color must fall to be considered selected for color replacement, or color range selecting. If the Color Space is grayscale, the first two bytes are the low and high endpoints of the range of gray values that are to be selected. The other four bytes should be zeroed. If the Color Space is Lab, then the first two bytes are the low and high endpoints of a range of ‘L’ values, the second two bytes are the low and high endpoints of a range of ‘a’ chromanance values, and the third pair bytes are the low and high endpoints of a range of ‘b’ chromanance values.

4. Fuzziness (2 bytes)
This short integer records the fuzziness setting, which controls how colors close to the selected colors are to be affected. It ranges from 0 to 200.

5. Transform Settings (6 bytes)
For files loaded into the Color Range dialog, these values are ignored. The Color Range dialog will write zeroes here. For Replace Color, this consists of three short integers; all values range from –100 to 100:

a. hue transform (2 bytes)
The hue change to be applied to the selected colors.

b. saturation transform (2 bytes)
The saturation change to be applied to the selected colors.

c. lightness transform (2 bytes)
The lightness change to be applied to the selected colors.

Scratch Area

Macintosh file type: '8BSR'
Windows file extension:.ASR

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

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Scratch Area data (variable)
The Photoshop scratch area consists of RGB image data. The three planes of data are written one after the other, in the order Red, Green, Blue; each consists of the following:

a. bounds (16 bytes)
A rectangle, four long integers giving the bounds of the scratch data (in the order top, left, bottom, right); for Photoshop 3.0, this must always correspond to [0, 0, 89, 200] as the Scratch palette has a fixed size.

b. depth (2 bytes)
depth of the current plane of data, which is always 8.

c. image data (variable):

i. compression (2 bytes)
Two values are currently defined: 0 = Raw Data, 1 = RLE compressed

ii. data (variable)
Each plane of the scratch image data is stored in scanline order, with no pad bytes.

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

If the compression code is 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

Macintosh file type: '8BSV'
Windows file extension:.ASV

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

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Correction Method (2 bytes)
A short integer indicating how the color correction is to be applied: in Relative (0) or Absolute (1) mode.

3. Plate Corrections (80 bytes)
The remainder of the file contains 10 correction records, one after the other.

Each record is written as follows:

a. cyan correction (2 bytes)
A short integer in the range –100 to 100 indicating the amount of correction for the cyan component.

b. magenta correction (2 bytes)
A short integer in the range –100 to 100 indicating the amount of correction for the magenta component.

c. yellow correction (2 bytes)
A short integer in the range –100 to 100 indicating the amount of correction for the yellow component.

d. black correction (2 bytes)
A short integer in the range –100 to 100 indicating the amount of correction for the black component.

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

Macintosh file type: '8BSS'
Windows file extension:.ASP

Separation settings files are loaded and saved in Photoshop's Separation Setup dialog, via the Preferences submenu.

1. Version (2 bytes)
Equal to 300, written as a short integer.

2. Separation Type (1 byte)
A boolean flag indicating UCR (value = 0) or GCR (value = 1) separations.

3. Black Limit (2 bytes)
A short integer giving the black ink limit, ranging from 0 to 100.

4. Total Limit (2 bytes)
A short integer giving the total ink limit, ranging from 200 to 400.

5. UCA Amount (2 bytes)
A short integer giving the undercolor addition for GCR separations, ranging from 0 to 100.

6. Black Generation Curve (variable)
This is a spline curve. The format is identical to a single curve instance from the Curves file format. It is composed of two parts:

a. point count (2 bytes)
A short integer in the range 2 to 19 indicates how many points are in the curve.

b. curve points (point count * 2 bytes)
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.

Hence a null curve (no change to input values) is represented by the following five–number, ten–byte sequence in a file: 2 0 0 255 255 .

Note that the black generation curve and the UCA limit must both be present even if the Separation Type is set to UCR.

Separation tables

Macintosh file type: '8BST'
Windows file extension:.AST

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

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

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.

Otherwise, the file has the following format:

1. Version (2 bytes)
Equal to 300, written as a short integer.

2. Has Lab to CMYK (1 byte)
Boolean indicating whether the file contains an Lab to CMYK table.

3. Has CMYK to Lab (1 byte)
Boolean indicating whether the file contains an CMYK to Lab table.

4. Lab to CMYK Table (33 * 33 * 33 * 4 bytes, optional)
If field 2 is equal to 1 (true), this section contains the CMYK colors for 33 * 33 *33 Lab colors. The Lab colors that are the source colors for this can be generated:

```
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);
        }
```

The CMYK colors are written in interleaved order, one byte each ink, 0 = 100%, 255 = 0%.

5. CMYK to Lab Table ((33 * 33 * 33 + 256) * 3 bytes, optional)
If field 3 is equal to 1 (true), this section contains the Lab colors for 33 * 33 *33 + 256 CMYK colors. The CMYK colors that are the source colors for this can be generated:

```
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;
}
```

The Lab colors are written in interleaved order, one byte per component.

If after reading the above data, the file is not yet empty, then the file contains the following data.

6. Has Gamut Table (1 byte, 1 = have table, 0 = don't have table)
Flag to indicate whether a gamut table follows. If zero, then this flag is the last byte of the file; the filler byte and gamut table (described below) will not be present.

7. Filler (1 byte, optional)
If Has Gamut Table == 1, then this filler byte should also be set to 1 for compatibility reasons.

8. Gamut Table ((((33 * 33 * 33L) + 7) >> 3) bytes, optional)
If Has Gamut Table == 1, then the gamut table data consists of (((33 * 33 * 33L) + 7) >> 3) bytes of data. This is a bit table indexed in the same way as the Lab–>CMYK table with the provision that the high bit of the first byte is at index 0, etc.

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

Transfer function

Macintosh file type: '8BTF'
Windows file extension:.ATF

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.

1. Version (2 bytes)
Equal to 4, written as a short integer.

2. Functions (112 bytes)
There are four transfer functions in the file. Each function is made up of the following subsections:

a. curve (26 bytes)
A transfer curve consists of 13 short integers, each ranging from 0 to 1000 (1000 represents the value 100.0). In addition, all but the first and last value may be –1 (representing no point on the curve). Hence a null transfer curve looks like this: 0, –1, –1, –1, –1, –1, –1, –1, –1, –1, –1, –1, 1000.

b. override (2 bytes)
This is a boolean flag indicating whether the curve should override the printer's default transfer curve. If it is zero, the printer's curve will not be overridden.

Note again that the file always contains four functions. For example, when writing the printer transfer functions for Grayscale images 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 data structures in contained in chapter 4. PiMI data structures are described in chapter 5. The different plug–in parameter blocks are described in their respective chapters (6–9).

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	The version number for this structure. The current version number is 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.

Table A–1: PSPixelMap structure (Continued)

Type	Field	Description
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	Tthe phase of the checkerboard with respect to the top left corner of the PSPixelMap.
int32	maskPhaseCol	The phase of the checkerboard with respect to the top left corner of the PSPixelMap.

PSPixelMask

The PSPixelMask structure is defined as follows:

```
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:
		`#define kSimplePSMask   0`
		`#define kBlackMatPSMask 1`
		`#define kGrayMatPSMask  2`
		`#define kWhiteMatPSMask 3`
		`#define kInvertPSMask   4`

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	This field must be filled in with the 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	This field selects the operation performed by the ColorServices callback. #define plugIncolorServicesChooseColor 0 #define plugIncolorServicesConvertColor 1 #define plugIncolorServicesSamplePoint 2 #define plugIncolorServicesGetSpecialColor 3
int16	sourceSpace	This field is used for to indicate the color space of the input color contained in color-Components. For plugIncolorServicesChoose-Color the input color is used as an initial value for the picker. For plugIncolorService-sConvertColor the input color will be con-verted from the color space indicated by sourceSpace to the one indicated by resultSpace. Available color spaces: #define plugIncolorServicesRGBSpace 0 #define plugIncolorServicesHSBSpace 1 #define plugIncolorServicesCMYKSpace 2 #define plugIncolorServicesLabSpace 3 #define plugIncolorServicesGraySpace 4 #define plugIncolorServicesHSLSpace 5 #define plugIncolorServicesXYZSpace 6

Table A–3: ColorServicesInfo structure (Continued)

Type	Field	Description
int16	resultSpace	This field holds the desired color space of the result color from the ColorServices call. The result will be contained in the colorComponents field when ColorServices returns. For the plugIncolorServicesChooseColor selector, resultSpace can be set to plugIncolorServicesChosenSpace to return the color in whichever color space the user chose the color. 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, this will only be true for colors returned in the plugIncolorServicesCMYKSpace color space.
Boolean	resultInGamut	This output only field is a boolean value that indicates whether the returned color is in gamut for the currently selected printing setup. It is only meaningful if the resultGamutInfoValid field is true.
void *	reservedSourceSpaceInfo	Must be NULL. A parameter error will be returned if they are not.
void *	reservedResultSpaceInfo	Must be NULL. A parameter error will be returned if they are not.
int16	colorComponents[4]	This array contains the actual color components of the input or output color. Refer to the notes following this table for more information about the color values that are represented in this array.
void *	reserved	Must be NULL. A parameter error will be returned if they are not.
union	selectorParameter	This union is used for providing different information based on the selector field. The pickerPrompt variant contains a pointer to a Pascal string which will be used as a prompt in the Photoshop color picker for the plugIncolorServicesChooseColor call. NULL can be passed to indicate no prompt should be used. The globalSamplePoint field points to a Point record that indicates the current sample point. The specialColorID should be either: plugIncolorServicesForegroundColor (0) or plugIncolorServicesBackgroundColor (1).

Notes:
The colorComponents array contains color values as listed in the color space name. Components not used in the input color space need not be filled in and components not used in the result color space are undefined.

For **RGB colors**, the colorComponents values are:

colorComponent[0] = Red value
colorComponent[1] = Green value
colorComponent[2] = Blue value
colorComponent[3] = (undefined)

Each RGB value should be in the range 0 through 255.

For **HSB colors**, the colorComponents values are:

colorComponent[0] = Hue value
colorComponent[1] = Saturation value
colorComponent[2] = Brightness value
colorComponent[3] = (undefined)

The Hue value should be in the range 0 through 359 degrees. The Saturation and Brightness values should be in the range 0 through 255 representing 0 through 100%.

For **L*a*b colors**, the colorComponents values are:

colorComponent[0] = L value
colorComponent[1] = a value
colorComponent[2] = b value
colorComponent[3] = (undefined)

L ranges from 0 to 255 representing the range 0...100.

a and b range from 0 to 255 representing the range -128 to 127. Note that these are merely shifted by +128, it is not a two’s complement signed number.

For **CMYK colors**, the colorComponents values are:

colorComponent[0] = Cyan value
colorComponent[1] = Magenta value
colorComponent[2] = Yellow value
colorComponent[3] = blacK value

Each component ranges from 0 to 255, representing ink values from 100% down to 0%. This may be counterintuitive, since larger values represent lighter colors. This does make it easier to use CMY and RGB interchangeably.

For **Grayscale colors**, the colorComponents values are:

colorComponent[0] = Gray value
colorComponent[1] = (undefined)
colorComponent[2] = (undefined)
colorComponent[3] = (undefined)

The gray value should be in the range 0 through 255.

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;
```

The fields of this record are as follow:

Table A–4: 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 chapter 10 for more details.

```
struct ResolutionInfo
    {
    Fixed                hRes;
    int16                hResUnit;
    int16                widthUnit;
    Fixed                vRes;
    int16                vResUnit;
    int16                heightUnit;
    };
```

hRes and **vRes** are always stored in units of pixels/inch. **hResUnit** and **vResUnit** are 1 if the resolution should be displayed in pixels/inch, or 2 if it should be displayed in units of pixels/cm.

widthUnit and **heightUnit** are 1 for inches, 2 for cm, 3 for points, 4 for picas, or 5 for columns.

DisplayInfo

This structure contains display information about each channel. It is written as an image resource. See chapter 10 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
    };
```

See the table 11–1 for a list of colorSpace values.

B PiPL Grammar

This information is included as reference material. If you use the example source code and the documentation earlier in the chapter, you probably won't need to worry about the specifics of the PiPL syntax.

```
# Miscellaneous definitions

<OSType>

<int16>

<int32>

<epsilon> :=

# Beginning of real grammar.

<PiPL spec> := <resource header> <resource body>

<resource header> :=
    "resource" "'PiPL'" "("
     <resourceID> <optional resource name> <optional attribute list>
    ")"

<optional name> :=
    <epsilon> |
    "," <string>

<optional attribute list> :=
    <epsilon> |
    "," <attribute> <attribute list tail>

<attribute list tail> :=
    <epsilon> |
     "|" <attribute> <attribute list tail>

<resource body> :=
    "{" "{"
    <property list>
    "}" "}"

<property list tail> :=
    <epsilon> |
    "," <property> <property list tail>

<property list> :=
    <epsilon>
    | <property> <property list tail>

<property> :=
    <kind property> |
    <version property> |
    <priority property> |
    <required host property> |
    <name property> |
    <category property> |
    <68k code descriptor property> |
    <powerpc code descriptor property> |
    <win32 x86 code property> |
    <supported modes property> |
    <filter case info property> |
    <format file type property> |
    <read types property> |
    <write types property> |
    <filtered types property> |
    <read extensions property> |
    <write extensions property> |
    <filtered extensions property> |
    <format flags property> |
    <format maximum size property> |
    <format maximum channels property> |
    <parsable types property> |
    <parsable extensions property> |
    <filtered parsable types property> |
    <filtered parsable extensions property> |
    <parsable clipboard types property>

<kind property> := "Kind" "{" <kind ID> "}"

<kind ID> := <OSType> |
    "Filter" |
    "Parser" |
    "ImageFormat" |
    "Extension" |
    "Acquire" |
    "Export"

<version property> := "Version" "{" <version clause> "}"

<version clause> := <int32> |
    "(" <wired version ID high> "<<" "16" ")" "|"
    "(" <wired version ID low> ")" |
    <wired version ID>

<wired version ID> := "FilterVersion" |
    "ParserVersion" |
    "ImageFormatVersion" |
    "ExtensionVersion" |
    "AcquireVersion" |
    "ExportVersion"

<wired version ID high> := "latestFilterVersion" |
    "latestParserVersion" |
    "latestImageFormatVersion" |
    "latestExtensionVersion" |
    "latestAcquireVersion" |
    "latestExportVersion"

<wired version ID high> := "latestFilterSubVersion" |
    "latestParserSubVersion" |
    "latestImageFormatSubVersion" |
    "latestExtensionSubVersion" |
    "latestAcquireSubVersion" |
    "latestExportSubVersion"

<priority property> := "Priority" "{" <int16> "}"

<required host property> := "Host" "{" <OSType> "}"
```

```
<name property> := "Name" "{" <string> "}"

<category property> := "Category" "{" <string> "}"

<68k code descriptor property> := "Code68k" "{" <OSType>, <int16> "}"

<powerpc code descriptor property> := "CodePowerPC" "{"
    <int32>, <int32> <optional name> "}"

<win32 x86 code property> := "CodeWin32X86" "{" <string> "}

<bitmap support> := "noBitmap" | "doesSupportBitmap"
<grayscale support> := "noGrayScale" | "doesSupportGrayScale"
<indexed support> := "noIndexedColor" | "doesSupportIndexedColor"
<RGB support> := "noRGBColor" | "doesSupportRGBColor"
<CMYK support> := "noCMYKColor" | "doesSupportCMYKColor"
<HSL support> := "noHSLColor" | "doesSupportHSLColor"
<HSB support> := "noHSBColor" | "doesSupportHSBColor"
<multichannel support> := "noMultichannel" | "doesSupportMultichannel"
<duotone support> := "noDuotone" | "doesSupportDuotone"
<LAB support> := "noLABColor" | "doesSupportLABColor"

<supported modes property> := "SupportedModes"
    "{"
    <bitmap support> ","
    <grayscale support> ","
    <indexed support> ","
    <RGB support> ","
    <CMYK support> ","
    <HSL support> ","
    <HSB support> ","
    <multichannel support> ","
    <duotone support> ","
    <LAB support>
    "}"

<filter case info property> := "FilterCaseInfo"
    "{"
        "{"
        <filter info case> # filterCaseFlatImageNoSelection
        <filter info case> # filterCaseFlatImageWithSelection
        <filter info case> # filterCaseFloatingSelection
        <filter info case> # filterCaseEditableTransparencyNoSelection
        <filter info case> # filterCaseEditableTransparencyWithSelection
        <filter info case> # filterCaseProtectedTransparencyNoSelection
        <filter info case> # filterCaseProtectedTransparencyWithSelection
        "}"
    "}"

<filter info case> :=
    <input matting> "," <output matting> ","
    <layer mask flag> "," <blank data flag> "," <copy source flag>

<input matting> :=
    "inCantFilter" |
    "inStraightData" |
    "inBlackMat" |
    "inGrayMat" |
    "inWhiteMat" |
    "inDefringe" |
    "inBlackZap" |
    "inGrayZap" |
    "inWhiteZap" |
    "inBackgroundZap" |
    "inForegroundZap"

<ouput matting> :=
    "outCantFilter" |
    "outStraightData" |
    "outBlackMat" |
    "outGrayMat" |
    "outWhiteMat" |
    "outFillMask"

<layer mask flag> := "doesNotFilterLayerMasks" | "filtersLayerMasks"
<blank data flag> := "doesNotWorkWithBlankData" | "worksWithBlankData"
<copy source flag> := "copySourceToDestination" |
    "doNotCopySourceToDestination"

<type creator pair> :=
    <OSType> "," <OSType>

<format file type property> :=
    "{"
    <type creator pair>
    "}"

<type creator pair list tail> :=
    <epsilon> |
    "," "{" <type creator pair> "}" <type creator pair list tail>

<type creator pair list> :=
    <epsilon> |
    "{" <type creator pair> "}" <type creator pair list tail>

<read types property> :=
    "{"
    <type creator pair list>
    "}"

<write types property> :=
    "{"
    <type creator pair list>
    "}"

<filtered types property> :=
    "{"
    <type creator pair list>
    "}"

<ostype list tail> :=
    <epsilon> |
    "," "{" <OSType> "}" <ostype list tail>

<ostype list> :=
    <epsilon> |
    "{" <OSType> "}" <ostype list tail>

<read extensions property> :=
    "{"
    <ostype list>
    "}"

<write extensions property> :=
    "{"
    <ostype list>
    "}"

<filtered extensions property> :=
    "{"
    <ostype list>
    "}"

<saves image resources flag> :=
    "fmtDoesNotSaveImageResources" | "fmtSavesImageResources"

<can read flag> :=
    "fmtCannotRead" | "fmtCanRead"

<can write flag> :=
    "fmtCannotWrite" | "fmtCanWrite"

<write if read flag> :=
    "fmtWritesAll" | "fmtCanWriteIfRead"

<format flags property> :=
    "{"
    <saves image resources flag> ","
    <can read flag> ","
    <can write flag> ","
    <write if read flag>
    "}"

<format maximum size property> :=
    "{"
    <int16>, <int16>
    "}"

<format maximum channels property> :=

<parsable types property> :=
    "{"
    <type creator pair list>
    "}"

<parsable extensions property> :=
    "{"
    <ostype list>
    "}"

<filtered parsable types property> :=
    "{"
    <type creator pair list>
    "}"

<filtered parsable extensions property> :=
    "{"
    <ostype list>
    "}"

<parsable clipboard types property> :=
    "{"
    <ostype list>
    "}"
```

File Extensions and Types