Adobe Photoshop™ 3.0

PLUG-IN TOOLKIT

Adobe Photoshop 3.0 Plug-in Toolkit

Copyright © 1991-95 Adobe Systems Incorporated All rights reserved.
Portions Copyright © 1990-91 Thomas Knoll

The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in this document. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.
Adobe, Adobe Premiere, Adobe Photoshop, Adobe Illustrator, Adobe Type Manager, ATM and PostScript are trademarks of Adobe Systems Incorporated that may be registered in certain jurisdictions. Macintosh and Apple are registered trademarks of Apple Computer, Inc. MicrosoftMS, and MS-DOS are registered trademarks, and Windows is a trademark of Microsoft Corporation. All other products or name brands are trademarks of their respective holders.

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.

Version History:

Date	Author	Description
11/07/94	David J. Wise	First Draft
1/15/95	David J. Wise	First Release
2/08/95	Seetharaman Narayanan	MS-Windows Mods.

Introduction 6
How to Use This Toolkit 6
Plug-In Overview 6
Historical Note 6
Installation 7
Macintosh 7
Windows 7
Resources 7
'PiPL's 7
Definition 7
Structure 8
Notes 10
Types 10
General properties 11
Code Descriptor Properties 12
Filter specific properties 15
Format specific properties 18
Parser specific properties 20
'PiMI's 21
Execution 23
Macintosh 23
Windows 23

Callback Routines 24
TestAbort( ) 24
UpdateProgress( ) 24
ProcessEvent( ) 24
DisplayPixels( ) 25
GetPropertyProc( ) 27
Property keys 27
AdvanceStateProc ( ) 29
ColorServicesProc ( ) 29
Monitor Descriptions 31

Callback Suites 32
Buffer Suite 32
AllocateBuffer( ) 33
LockBuffer( ) 33
UnlockBuffer( ) 33
FreeBuffer( ) 33
BufferSpace( ) 33
Pseudo-Resource Suite 33
CountPIResources( ) 33
GetPIResource( ) 34
DeletePIResource( ) 34
AddPIResource( ) 34
Handle Suite 34
NewPIHandle ( ) 34
DisposePIHandle ( ) 34
GetPIHandleSize ( ) 34
SetPIHandleSize ( ) 34
LockPIHandle ( ) 35
UnlockPIHandle ( ) 35
RecoverSpaceProc ( ) 35

General Notes 35
Macintosh 35
Global Variables 35
Segmentation 35
About Boxes 36
Configuration 36
Windows 36
Configuration 36

About the Sample Plug-ins 37
Macintosh Version 37
Windows Version 37

Acquisition Modules 39
Basics 39
The AcquireRecord Structure 39
Record Fields 40
Calling Order 45
(1) acquireSelectorPrepare 45
(2) acquireSelectorStart 46
(3) acquireSelectorContinue 46
(4) acquireSelectorFinish 46
(5) acquireSelectorFinalize 47
State Machine 47
Error return values 49
Sample Plug-in 49
DummyScan 49

Export Modules 50
Basics 50
The ExportRecord Structure 50
Record Fields 51
Calling Order 55
(1) exportSelectorPrepare 55
(2) exportSelectorStart 55
(3) exportSelectorContinue 55
(4) exportSelectorFinish 55
State Machine 56
Error return values 56
Sample Plug-ins 57
DummyExport 57
HistoryExport 57
Paths to Illustrator 57

Filter Modules 58
Basics 58
The FilterRecord Structure 58
Record Fields 61
Calling Order 67
(1) filterSelectorParameters 67
(2) filterSelectorPrepare 68
(3) filterSelectorStart 68
(4) filterSelectorContinue 68
(5) filterSelectorFinish 68

State Machine 69
Error return values 70
Sample Plug-in 70
Dissolve 70

Image Format Modules 71
Basics 71
The FormatRecord Structure 71
Image Resources 72
Record Fields 73
Calling Sequences 77
(1) prepare 77
(2) start 78
(3) continue 78
(4) finish 79
Error return values 79
Sample Plug-in 79
Sample Format 79

Document File Formats 80
Image Resource Block 80
Path Resource Format 80
Photoshop 3.0 81
EPS 89
Filmstrip 91
TIFF 95

Load File Formats 97
Introduction 97
Arbitrary Map 98
Brushes 99
Color Table 101
Colors 102
Command Buttons 104
Curves 106
Duotone Options 108
Halftone Screens 110
Hue/Saturation 112
Ink Colors Setup 114
Custom Kernel 115
Levels 116
Monitor Setup 117
Replace Color/Color Range 118
Scratch Area 119
Selective Color 120
Separation Setup 121
Separation Tables 122
Transfer Function 124

Introduction

How to Use This Toolkit

The Adobe Photoshop Plug-In Toolkit is for developers who wish to write their own plug-in modules for use with Adobe Photoshop. Photoshop plug-ins are called by Photoshop to perform specific functions, such as acquiring an image or filtering a portion of an image.

This toolkit documentation starts with information that is common to all the plug-in types. The rest of the document is broken up into chapters specific to each type of plug-in, or to special types of files.

The best way to use this toolkit documentation is to read this Introduction chapter, then read the chapter specific to the type of plug-in you're writing. You should then study and understand the sample plug-ins of the type you're writing.

Plug-In Overview

Adobe Photoshop plug-ins are separate files that contain code which allows either Adobe Systems, Inc. or third-party developers to extend Adobe Photoshop, without actually modifying the base application. Adobe Photoshop version 3.0 supports five kinds of plug-in modules:

1. Acquisition modules, which open an image in a new window. Acquisition 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, which output an existing image. Export modules can be used to print to printers that do not have chooser-level driver support, or to save images in unsupported or compressed file formats. These modules are accessed through the Export sub-menu.
3. Filter modules, which modify a selected area of an existing image. These modules are inserted into the Filter menu.
4. File format modules which provide support for additional image formats. These appear in the format pop-up in the Open..., Save As... and Save a Copy... dialogs.
5. Parser modules...TBD

A quick word about types is in order here. The interface files talk in terms of **int32**'s and **int16**'s rather than longs and shorts when they specify 32-bit integers. **VRect**'s are like Macintosh **Rect**'s but they have 32-bit coordinates. All of these types are defined in **PITypes.h**.

Historical Note

The concept of plug-in modules has become popular among Macintosh application developers. Perhaps the best known example is Apple's HyperCard, with its support for XCMD'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 implementation of plug-in modules was well designed. Its good features include allowing the plug-in modules to reside in individual files (rather than having to be pasted into the application using ResEdit), allowing the plug-in modules to be placed anywhere (not just in the system folder), and allowing for future extensions by means of a version number.
Adobe Photoshop's implementation of plug-in modules is similar to that used by Silicon Beach. Its uses a similar calling sequence, and the same version number scheme.
Unfortunately, the detailed interface for Adobe Photoshop's plug-in modules is completely different from that used by Silicon Beach. The differences were required primarily to support color images and Adobe Photoshop's virtual memory scheme.

Installation

Macintosh

To install a plug-in module, all the user must do is drag the module's icon to one of the following folders: the same folder as the application or the plug-ins folder designated in the user's 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 "¬" are ignored.)

Windows

To install a plug-in module, the user must copy the plug-in 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.

Resources

'PiPL's

Definition

A Plug-In Property List, often called a 'PiPL' (pronounced "pipple") after its resource type code, is a flexible, extensible mechanism for representing plug-in metadata. This includes all information Photoshop needs to identify and load the plug-in as well as flags and other static properties that control the operation of the plug-in.

Plug-in Property Lists replace the Plug-in Module Information structure, often called a 'PiMI' (pronounced "pimmy") after its resource type code. A 'PiMI' is a fixed format record which originally contained only a version number. With the evolution of Photoshop's plug-in interface, this record 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.

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

All plug-in file types are searched for 'PiPL' resources. Historically each type of plug-in had its own file type, as follows:

Plug-in type	Macintosh file type	Windows 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

Filenames and extensions are case insensitive.

Only plug-ins of the correct type are searched for 'PiMI' resources of a given type. (This is due to the pairing up of 'PiMI' resources with an appropriate type of code resource.) File types are only a matter of convention for 'PiPL' based plug-ins. 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 the following section. This allows one to place both 'PiPL' and 'PiMI' resources in a plug-in. 3.0 or later compatible hosts will use the 'PiPL' while 2.5.1 compatible hosts will use the 'PiMI'.

Structure

The Plug-in property list has a version number and a count followed by a sequence of arbitrary length byte containers called properties. A "C " struct definition for the plug-in property list is as follows:

```
typedef struct PIPropertyList
  {
```

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

- **version**
  This denotes the version of this specification the 'PiPL' is formatted to. The current version is 0.

- **count**
  This field holds the number of properties contained in the 'PiPL'. 0 is a valid value denoting a 'PiPL' with no properties.

- **properties**
  A variable length array of variable length property data structures. Holds the actual contents of the 'PiPL'.

Each property has a vendor code, a key, an ID, a length, and property data the size indicated by the length. The "C" struct definition for the plug-in properties are as follows:

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

The fields are defined as follows:

- **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'.
- **propertyKey**
  This field specifies the type of this property. Property types used by Photoshop are documented below. (Think of a property type as similar to a resource type.)
- **propertyID**
  In theory this can be used to store more than one property of a given type (rather like a resource ID). In practice, this field is always zero. It should be thought of as reserved for future use.

- **propertyLength**
  This field contains the length of the **propertyData** field. It does not include any padding bytes after propertyData to achieve four byte alignment. This field may be zero.
- **propertyData**
  A variable length field that contains the bytes which are the contents of this property. Any values may be contained.

Padding
Each property must be padded such that the next property begins on a four byte boundary.

Notes
Specific properties can be extended in an upward compatible fashion by adding extra data at their end. The length field will allow an application to determine how much data is present, Optional properties can be omitted without concern. (As opposed to a fixed length structure where omitted fields must be given a default value.) The 'PiPL' format is fairly portable in that everything is four byte aligned. All OSType and int32 fields are represented in native byte order for a given platform so the bytes of "the same" 'PiPL' will differ between a big-endian machine (e.g. the Macintosh) and a little-endian machine (e.g. an Intel x86 based Windows machine). Although if one examines the bytes of the PiPL section of an x86 resource binary, they will be backward compared to the Mac, the user can generally not concern themselves with the difference. If they use the pre-defined PI-types, they will be interpreted and stored correctly as in the following example (see PIKindProperty). If, however, an OSType has not been defined and they wish to enter it as a 4-char series, then (since it is not interpreted as a long) they would have to supply the chars in reverse order (see "MIB8").

```
"MIB8",
PIKindProperty,
0L,
4L,
"MFB8",
```

The Macintosh plug-in kit includes a resource template for the 'PiPL' type and the Windows version of the kit includes a "PiPL Parser" application (CnvtPiPL.exe) to transform Mac ".r" files into Windows ".rc" files. If you are developing for the Macintosh platform, you can automatically convert your Macintosh PiPL resources into MSWindows' custom PiPL format by using CnvtPiPL.exe. This enables you to keep just one copy of PiPL and saves you the headache of converting PiPLs by hand and also eliminates the errors caused in the process. In order to use CnvtPiPL.exe, you need to pre-process your *.r file using the standard C pre-processor and pipe the output to CnvtPiPL. The sample makefiles illustrate the process. Even if you are not developing for the Macintosh, you are strongly encouraged to use the resource template (i.e. please refer to any of the *.r files under the "RIncludes" sub-directory) to create the PiPLs, as it is more intuitive to create the PiPLs that way, and then use CnvtPiPL.exe to convert them. CnvtPiPL.exe takes care of all byte alignment and byte-ordering issues for you automatically. If you are going to use the PiPL resource template to create your PiPLs, you can safely ignore the techno-babble about the Windows' PiPL resource format in the following sections.

It is intended for 'PiPL's to collect all plug-in metadata in a single place. This includes the name and category of the plug-in as well as all other information. (In the 'PiMI' world, the name and category were

stored as resource names on the plug-in code resource and 'PiMI' respectively.) Vendors are encouraged to define new properties for extensions to plug-in metadata rather than introducing new resource types.

Types

int16, int32:
These are 16 and 32 bit integers respectively. They are stored within the 'PiPL' in native byte order.

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.

Arrays:
Arrays are represented as a contiguous set of entries in the 'PiPL' typically with native padding and alignment constraints observed. The length of the array is usually determined by the property length for arrays of fixed length structures or types.

General properties

Plug-in Kind ***OSType***

```
#define PIKindProperty 0x6b696e64L /* 'kind' */
```

This property encodes the type or kind of a plug-in. Valid values are:

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

Version of kind specific API ***int32***

```
#define PIVersionProperty     0x76657273L /* 'vers' */
```

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.

Plug-in load order priority ***int16***

```
#define PIPriorityProperty     0x70727479L /* 'prty' */
```

This property determines the order in which this plug-in will be loaded. This is typically only important for acceleration extensions. It can however be used to control the order in which items with the same name show up in menus. Lower numbers (including negative ones as the field is signed) load first.

Supported image modes ***FlagSet***

```
#define PIImageModesProperty    0x6d6f6465L /* 'mode' */
```

This is a set of flags that determines which image modes the plug-in supports.

Required Host ***OSType***

```
#define PIRequiredHostProperty  0x686f7374L /* 'host' */
```

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.)

Plug-in category ***PString***

```
#define PICategoryProperty   0x63617467L /* 'catg' */
```

Plug-in name ***PString***

```
#define PINameProperty             0x6e616d65L /* 'name' */
```

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.

68k code descriptor ***PI68KCodeDesc***

```
PI68KCodeProperty            0x6d36386bL /* 'm68k' */
```

This descriptor indicates a 68K code resource. The type for this property is as follows:

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

Any resource type may be used, but conventions for various types of plug-ins are as follows:

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

(This convention comes from Photoshop 2.5.1 where these types were required. When building a Plug-in that is backwards compatible with 2.5.1 hosts, these resource types must be used.)

68k FPU code descriptor ***PI68KCodeDesc***

```
PI68KFPUCodeProperty    0x36386670L    /* '68fp' */
```

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.

PowerPC code fragment descriptor ***PICFMCodeDesc***

```
PIPowerPCCodeProperty  0x70777063L /* 'pwpc' */
```

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;
```

With the fields documented as follows:

- **fContainerOffset**
  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.
- **fContainerLength**
  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.
- **fEntryName**
  The entrypoint name 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. The entrypoint name allows a single code fragment to contain more than one plug-in. (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.) Entrypoint names allow more than one plug-in to be exported from a single code fragment.

Windows 32-bit DLL code descriptor ***PIWin32X86CodeDesc***

```
#define PIWin32X86CodeProperty 0x77783836L /* 'wx86' */

typedef struct PIWin32X86CodeDesc
{
  "MIB8",
  PIWin32X86CodeProperty,
  0L,
  12L,
  "ENTRYPOINT1\0",  ( if pad needed, "ENTRYPNT1\0\0\0" )
} PIWin32X86CodeDesc;
```

This code descriptor is used for 32 bit Windows DLL's. The entrypoint name is used to lookup the function which is called to invoke the plug-in. The entrypoint name is represented as a NUL terminated string. The string may need to be padded with additional NULs to satisfy the 4 byte alignment requirement.

Note for Windows' Developers:
CnvtPiPL .exe does not recognize any Code Descriptor Property other than "CodeWin32X86".

Filter specific properties

Layer case information ***Array***

```
#define PIFilterCaseInfoProperty 0x66696369L /* 'fici' */
```

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 off, 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 introduce spatial distortions.

Photoshop 3.0 offers a fair bit of 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.

So what are these "cases"? There are seven of them. The property is an array of seven four byte entries, one for each case. The cases are as follows:

```
#define filterCaseFlatImageNoSelection 1
```

This is a background layer or a flat image. There is no transparency data. Nor is there a selection.

```
#define filterCaseFlatImageWithSelection 2
```

No transparency data, but a selection may be present. The selection will be presented as mask data.

```
#define filterCaseFloatingSelection 3
```

Image data with an accompanying mask.

```
#define filterCaseEditableTransparencyNoSelection 4
```

A layer with transparency editting enabled and no selection.

```
#define filterCaseEditableTransparencyWithSelection    5
```

A layer with transparency editting enabled and a selection.

```
#define filterCaseProtectedTransparencyNoSelection    6
```

A layer with transparency editting disabled and no selection.

```
#define filterCaseProtectedTransparencyWithSelection    7
```

A layer with transparency editting disabled and a selection.

Photoshop's fall through algorithm is as follows:
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 four byte entry looks like so:

```
struct caseInfo {
  unsigned8 inputHandling;
  unsigned8 outputHandling;
  unsigned8 flags;
  unsigned8 reserved;
};
```

The inputHandling and outputHandling fields specify pre-processing and post-processing of the image data respectively. Common values for both fields are as follows:

```
#define filterDataHandlingCantFilter    0
```

This case is not supported.

```
#define filterDataHandlingNone    1
```

Do nothing to the image data.

The next three cases 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.

```
#define 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.

```
#define 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.

```
#define 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:

```
#define filterDataHandlingDefringe        5
```

Defringe transparent areas filling with the nearest defined pixels using taxicab distance. Note that this only applies to fully transparent pixels.

```
#define filterDataHandlingBlackZap        6
```

Set color component of totally transparent pixels to black (0).

```
#define filterDataHandlingGrayZap         7
```

Set color component of totally transparent pixels to gray (128).

```
#define filterDataHandlingWhiteZap        8
```

Set color component of totally transparent pixels to white (255).

```
#define filterDataHandlingBackgroundZap        10
```

Set color component of totally transparent pixels to the current background color.

```
#define filterDataHandlingForegroundZap         11
```

Set color component of totally transparent pixels to the current foreground color.

The following mode is only useful for output:

```
#define 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 things like Photoshop's Clouds plug-in which wants to fill an area with a value.

The flags field 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 PIFilterWorksWithBlankDataBit1
```

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 we could create opacity -- in the protected transparency case, we would be left with what we 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.

Format specific properties

Default file creation information ***TypeCreatorPair***

```
#define PIFmtFileTypeProperty 0x666d5443 /* 'fmTC' */
```

Determines the default type and creator code used for files newly created with this format plug-in. On the Windows platform, we don't store TypeCreator information (except internally), the PIFmtFileTypeProperty is not required, they are simply interpreted as of type 'BINA' and creator 'mdos'. All the info regarding what files can be read/written is obtained from the PIReadExtProperty or the PIFilteredExtProperty. PiMI extensions are converted to PIReadExtProperty's so use of PIFilteredExtProperty requires additional coding if the developer is porting a 16-bit plugin to 32-bit.

Readable types ***TypeCreatorPair[ ]***

```
#define PIReadTypesProperty 0x52645479 /* 'RdTy' */
```

This property contains a list of type and creator pairs which the format plug-in can read.

Filtered types ***TypeCreatorPair[ ]***

```
#define PIFilteredTypesProperty 0x66667454 /* 'fftT' */
```

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. See documentation for formatSelectorFilterFile plug-in selector.

Readable extensions ***OSType[ ]***

```
#define PIReadExtProperty 0x52644578 /* 'RdEx' */
```

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.
(The extension ".abc" would be encoded as "abc ". It is 0x61626320 on a big-endian machine. It is byte-reversed on a little-endian machine 0x20636261, but you can still encode it as "abc ".)

Filtered extensions ***OSType[ ]***

```
#define PIFilteredExtProperty 0x66667445 /* 'fftE' */
```

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.
(The extension ".abc" would be encoded as "abc ". It is 0x61626320 on a big-endian machine. It is byte-reversed on a little-endian machine 0x20636261, but you can still encode it as "abc ".)

Format flags ***FlagSet***

```
#define PIFmtFlagsProperty 0x666d7466 /* 'fmtf' */
```

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

```
#define PIFmtReadsAllTypesFlag 0
```

obsolete.

```
#define PIFmtSavesImageResourcesFlag 1
```

Along with the pixel information for a file, Photoshop stores various so-called 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 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 we can write using this plug-in if we read the file using this plug-in. 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 we can save out filmstrips that we read in using the plug-in.

Maximum supported size ***Point***

```
#define PIFmtMaxSizeProperty 0x6d78737a /* 'mxsz' */
```

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.

Maximum channels ***int16[ ]***

```
#define PIFmtMaxChannelsProperty 0x6d786368 /* 'mxch' */
```

An array of one byte 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 a format supports a single alpha channel in RGB mode, it should set maxChannelsÊ[plugInModeRGBColor] to 4. A plug-in may still be asked to save more channels than it reports it can support. This field exists primarily so that we can warn the user that alpha channels will be discarded.

Parser specific properties

Parsable types ***TypeCreatorPair[ ]***

```
#define PIParsableTypesProperty 0x70735459L /* 'psTY' */
```

This property contains a list of type and creator pairs for files which the parser plug-in can parse.

Filtered parsable types ***TypeCreatorPair[ ]***

```
#define PIFilteredParsableTypesProperty 0x70735479L /* 'psTy' */
```

This property contains a list of type and creator pairs for files which the parser plug-in may be able to parse. The plug-in will be called to make the determination. See the documentation for the parserSelectorCanRead selector.

Parsable extensions ***OSType[ ]***

```
#define PIParsableExtProperty 0x70734558L /* 'psEX' */
```

This property contains a list of extensions for files which the parser plug-in can parse.

(The extension ".abc" would be encoded as "abc ". It is 0x61626320 on a big-endian machine. It is byte-reversed on a little-endian machine 0x20636261, but you can still encode it as "abc ".)

Filtered parsable extensions***OSType[ ]***

```
#define PIFilteredParsableExtProperty 0x70734578L /* 'psEx' */
```

This property contains a list of extensions for files which the parser plug-in may be able to parse. The plug-in will be called to make the determination. See the documentation for the parserSelectorCanRead selector.

(The extension ".abc" would be encoded as "abc ". It is 0x61626320 on a big-endian machine. It is byte-reversed on a little-endian machine 0x20636261, but you can still encode it as "abc ".)

Parsable clipboard types ***OSType[ ]***

```
#define PIParsableClipTypesProperty 0x70734342L /* 'psCB'*/
```

This Macintosh specific property contains a list of clipboard type codes which can be parsed by this plug-in.
'PiMI's

'PiMI' resources are superceded by the previously described 'PiPL' resources, however for compatibility with existing plug-ins, Photoshop 3.0 will still recognize and function with plug-ins containing only 'PiMI's.

The 'PiMI' resource consists of two pieces: general information applicable to all (or most) plug-in types and type specific info. The general information precedes the type specific information. Since the information proceeds serially, however, all fields must be filled in through and including the last field supplied. Generally, a 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 as follows:

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

} PlugInInfo;
```

- **version**
  The major version number for the interface used by the plug-in. This field is required.
- **subVersion**
  The minor version number for the interface used by the plug-in. This field is required.
- **priority**
  The priority which should be associated with this plug-in when it loads. Currently, this is only used for extension modules.
- **generalInfoSize**
  The size of the general plug-in information in this resource.
- **typeInfoSize**
  The size of the type-specific plug-in information in this resource.
- **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 hosts may respect this field, the plug-in 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.).
- **requiredHost**
  If the plug-in requires a particular host proc (see below), it should specify the signature for that host proc here. If it does not require a particular host proc, it should fill this field with spaces. Photoshop will decline to load plug-ins which require host procs other than Photoshop's **'8BIM'** proc. Plug-in developers should be aware that again one cannot count on host developers to check this field.

The type specific info is documented in the documentation on the various types of plug-ins.

Note that it is possible to have multiple plug-in modules in a single file, as long as the resource numbers do not conflict (if the modules are of different types, the file type should be set to '8BPI', which is always searched as a special case). In most cases it is not a good idea to place multiple modules in a single file, since it reduces the user's control of which modules are installed. However, there are cases when this should be done. One example is matched acquisition/export, though these frequently correspond to the new file format modules. In such cases, only one of the modules should display an about box, describing both modules. 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 do maintenance on their plug-in folders.

Execution

Macintosh

When the user takes an action that causes a plug-in module to be called, Adobe Photoshop opens the resource fork of the file the module resides in, loads the resource into memory, locks it, and calls the routine starting at the first byte of the resource. The Macintosh prototype is:

- **pascal void PlugIn (short selector, Ptr stuff, long *data, short *result);**

Windows

When the user takes an action that causes a plug-in module to be called, Adobe Photoshop does a LoadLibrary call to load the module into memory. For each PiPL resource found in the file, Photoshop calls GetProcAddress (routineName) where "routineName" is the name found under "CodeWin32X86" property to get the routine's address. If the file contains only PiMI resources and no PiPLs, Photoshop does a GetProcAddress for each PiMI resource found in the file looking for the entry point ENTRYPOINT% where % is the integer nameID of the PiMI resource to get the routine's address. Once Photoshop obtains the routine's address it calls the routine following the calling conventions:

- **void Plugin(short selector, void * stuff, void *data, short *result);**

The parameters are to be interpreted as follows:

- **selector**
  This is an integer operation selector code. Selector 0 always means display an about box. The meaning of other values depends on the type of plug-in.

- **stuff**
  This is a pointer to a parameter block. The exact nature of the parameter block depends on the type of plug-in. In the case of the about box selector, this pointer leads to a record containing a single platform specific 32-bit value. In the case of the Macintosh, this field contains no useful data.

- **data**
  This is a pointer to a long integer (32-bit value) which Photoshop will maintain for the plug-in across invocations. One standard use for this field is to store a pointer or handle to a block of memory used to store the plug-in's "global" data. It will be zero the first time the plug-in is called.

- **result**
  This is a pointer to the result code to be returned by the plug-in. A value of zero means that no error has occurred. Any positive value means that execution of the plug-in should stop, but the plug-in has already displayed any appropriate error message. (If the user cancels the operation in any way, the plug-in should return a positive value and not report an error.) A negative value also means that execution of the plug-in stop, but that the host should display its standard error dialog describing the error. Each plug-in type has a one plug-in specific error code (see the header files for details) which can be returned here. Standard Maci OS error codes such as memFullErr (-108) can also be used.

Callback Routines

A number of fields in the various plug-in "**stuff**" structures are callbacks to the host program to provide specific services. A number of these routines are common to multiple plug-in types and are documented here. Those specific to a single plug-in type are documented in that type's documentation. Some of these routines are arranged in suites accessed via a pointer to a table of function pointers. Some of these routines are also new in Photoshop 3.0 and may not be provided by other 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:

    #defineerrPlugInHostInsufficient -30900

All of the routines use Pascal calling conventions. A complete list can be found in **PIGeneral.h.** The two routines guaranteed to be present if defined in the plug-in interface record provide checking for command-period (or other requests to abort) and access to a host progress indicator:

TestAbort( )

- **pascal Boolean TestAbort ( );**
  The 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.

UpdateProgress( )

- **pascal void UpdateProgress (long done, long total);**
  The 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.

Photoshop 3.0 provides a routine to allow some plug-in types to pass events to Photoshop for processing. For example, when a plug-in receives a deactivate event for one of Photoshop's windows, it is considered polite if the plug-in passes this event on to Photoshop. This routine can also be used to allow Photoshop to do updates of its own windows by passing relevant update and null events to Photoshop. The calling sequence for this routine is:

ProcessEvent( )

- **pascal void ProcessEvent (EventRecord *event);**
  The parameter is actually a generic pointer and the data pointed to will depend on the platform the plug-in is running on.

A general host callback routine may or may not be present. It's functionality and calling sequence is host specific and is defined by the hostType field. The general functionality and calling sequence is identified by the signature supplied with the procedure pointer. Host proc's are used to support operations which are special to some host and would not apply to most other hosts. In the case of Photoshop's callback, for example, this is a place where some callback operations which may well not be supported in the long term but which are critical to getting certain features in Photoshop working as plug-ins are provided. *This callback is not generally useful on the Windows platform.*

The next general callback routine is used to display pixels in various image modes. It takes a structure describing a block of pixels to display:

DisplayPixels( )

- **pascal OSErr DisplayPixels (const PSPixelMap *source, const VRect *srcRect, int32 dstRow, int32 dstCol, unsigned32 platformContext);**

The parameters have the following interpretations:

- **source**
  The **PSPixelMap** containing the pixels to be displayed.

```
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;
```

  The fields in this structure are as follows:

  - **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.

- **bounds**
  The bounds for the pixel map.
- **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 pas plugInModeDuotone or plugInModeIndexedColor.
- **rowBytes**
  The offset from one row to the next of pixels.
- **colBytes**
  The offset from one column to the next of pixels.
- **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.
- **baseAddr**
  The address of the byte value for the first plane of the top left pixel.

- **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.

- **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.
  The PSPixelMasks structure is defined as follows:

```
typedef struct PSPixelMask
  {
  struct PSPixelMask *next;
  void *maskData;
  int32 rowBytes;
  int32 colBytes;
  int32 maskDescription;
  }
PSPixelMask;
```

- **next**
  A pointer to the next mask in the chain

- **maskData**
  A pointer to the mask data.

- **rowBytes**
- **colBytes**
  The row and column steps for the mask.

- **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
  ```

- **maskPhaseRow**
- **maskPhaseCol**
  maskPhaseRow and maskPhaseCol give the phase of the checkerboard with respect to the top left corner of the **PSPixelMap**.

- **srcRect**
  The rectangle within that **PSPixelMap** to be displayed.

- **dstRow**
- **dstCol**
  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 does not scale the pixels, so specifying the top left corner is sufficient to specify the destination.

- **platformContext**
  This parameter is not used on the Macintosh since the routine simply assumes that the target is the current port. On Windows, platformContext should be the target hDC, cast to an unsigned32. Under other platforms, this may specify a drawing context.

The routine will do the appropriate color space conversion and copybits 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.

GetPropertyProc( )

The GetProperty callback is available to filter and export modules. It allows these modules to get information about the document currently being processed.

- **pascal OSErr (*GetPropertyProc) (OSType signature, OSType key, int32 index, int32 * simpleProperty, Handle *complexProperty);**

The signature and key form a pair to identify the property of interest. Photoshop's signature is always '8BIM'. The key values are documented below and in PIProperties.h.

Properties like channel names and path names or data can be indexed. Indices generally start at 0.

Properties can consist either of an integer returned in simpleProperty or a handle returned in complexProperty. The type of property data is documented in PIProperties.h. In the case of a complex (i.e., handle based) property, the plug-in is responsible for disposing of the handle it is passed via the dispose call in the handle suite.

Properties involving strings -- e.g., channel names and path names -- are returned in a handle where the length of the handle determines the size of the string. There is no length byte nor is the string zero terminated.

Property keys

'nuch' (number of channels): This returns 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. This property is simple.

'nmch' (channel name): This returns 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. This property is complex.

'mode' (image mode): This returns the mode of the image using the constants defined in PIGeneral. This property is simple.

'nupa' (number of paths): This property returns the number of paths in the document. This property is simple.

'nmpa' (path name): This property returns the name of the indexed path. The paths are indexed starting with zero. This property is complex.

'path' (path contents): This property returns the contents of the indexed path in the format documented in the path resources documentation. LittleEndian platforms should note that the data is stored in BigEndian form. This property is complex.

'wkpa' (work path index): This property returns the index of the work path or -1 if there is no work path. This property is simple.

'clpa' (clipping path index): This property returns the index of the clipping path or -1 if there is no clipping path. This property is simple.

'tgpa' (target path index): This property returns the index of the target path or -1 if there is no target path. This property is simple.

This list is complete for Photoshop 3.0.1. Future versions of the program will almost certainly support more properties. We will also probably add a way to write to properties from plug-ins in some future version. There is no way to do so at this time.

AdvanceStateProc ( )

- **pascal OSErr (*AdvanceStateProc) (void);**

In a plug-in type specific manner, this callback allows the plug-in to drive Photoshop to update the buffers used for communicating between Photoshop and the plug-in without the plug-in actually returning from the selector call. It returns noErr if successful and a non-zero error code if something went wrong.

ColorServicesProc ( )

- **pascal OSErr (*ColorServicesProc) (ColorServicesInfo *info);**

```
typedef struct ColorServicesInfo
{
    int32 infoSize;
    int16 selector;
    int16 sourceSpace;
    int16 resultSpace;
    Boolean resultGamutInfoValid;
    Boolean resultInGamut;
    void *reservedSourceSpaceInfo;
    void *reservedResultSpaceInfo;
    int16 colorComponents[4];
    void *reserved;
    Str255 *pickerPrompt;
}
ColorServicesInfo;
```

The fields of this record are as follows:

- **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);
  ```

- **selector**
  This field selects the operation performed by the ColorServices callback. At present there are two operations available, choosing a color using the Photoshop color picker (actually, using the user's preferred color picker), and converting color values from one color space to another. The selectors for these are respectively:

  ```
  #define plugIncolorServicesChooseColor  0
  ```

```
#define plugIncolorServicesConvertColor 1
```

Available color spaces:

```
#define plugIncolorServicesRGBSpace     0
#define plugIncolorServicesHSBSpace     1
#define plugIncolorServicesCMYKSpace    2
#define plugIncolorServicesLabSpace     3
#define plugIncolorServicesGraySpace    4
#define plugIncolorServicesHSLSpace     5
#define plugIncolorServicesXYZSpace     6
```

- **sourceSpace**
  This field is used for to indicate the color space of the input color contained in colorComponents. For plugIncolorServicesChooseColor the input color is used as an initial value for the picker. For plugIncolorServicesConvertColor the input color will be converted from the color space indicated by sourceSpace to the one indicated by resultSpace.

- **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.

- **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.

- **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.

- **colorComponents**
  This array contains the actual color components of the input or output color. They will be included in the components array as they are listed in the color space name. So for plugIncolorServicesRGBSpace colorComponents[0] will contain the red (R) component, colorComponents[1] will contain the green (G) component, colorComponents[2] will contain the blue (B) component. Components not used in the input color space need not be filled in and components not used in the result color space are undefined.

- **pickerPrompt**
  This field 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.

- **reservedSourceSpaceInfo**
- **reservedResultSpaceInfo**
- **reserved**
  These three fields are reserved for future expansion and must be set to NULL. A parameter error will be returned if they are not.

Monitor Descriptions

A number of the plug-ins 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:

- **gamma**
  This field contains the monitor's gamma value or zero if the whole record is invalid.

- **redX**
- **redY**
- **greenX**
- **greenY**
- **blueX**
- **blueY**
  These fields specify the chromaticity coordinates of the monitor's phosphors.

- **whiteX**
- **whiteY**
  These fields specify the chromaticity coordinates of the monitor's white point.

- **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.

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 in order a 2 byte version number for the suite, a 2 byte count of the number of routines in the suite - routines can be added to the suite without incrementing the version number - and a series of **ProcPtr**'s for the routines. Before calling a callback defined in the suite, the plug-in needs to check the following conditions:

- The suite pointer must not be null.
- The suite version number must match the version number the plug-in wishes to use. (We do not expect to be changing version numbers with any degree of frequency.)
- The number of routines defined in the suite must be great enough to include the routine of interest.
- The pointer for the routine of interest must not be null.

If these conditions are not met and the plug-in does not want to work around the non-availability of the callback, the plug-in should put up an error dialog and then return to the host as if the user had canceled.

Buffer Suite

The buffer suite provides an alternative to the memory management functions available in previous versions of Photoshop's plug-in specification by providing 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 some mechanisms for interacting with this 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: (1) the memory is reserved throughout the execution of the plug-in and (2) 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. On Windows, Photoshop's memory scheme is designed such that it allocates just enough memory to not let Windows' virtual memory manager to kick in. If the plug-in allocates lots of memory using ***GlobalAlloc ( ),*** this scheme will be defeated and Photoshop will be double-swapping thereby degrading performance. Using the buffer suite, a plug-in can avoid doing some of the accounting for space to be reserved. This simplifies the prepare phase for acquire, filter, and format plug-ins. Unfortunately, export modules are expected to 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 really provide any help for export modules. But for other plug-ins, buffer allocations can be delayed until they are actually needed.

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

Version 1 was purely developmental. The routines in version 2 of the suite are:

AllocateBuffer( )

- **pascal OSErr AllocateBuffer (int32 size, BufferID *buffer);**
  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 - e.g., during the continue calls to filter and format plug-ins.

LockBuffer( )

- **pascal Ptr LockBuffer (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. It will optionally try to move the block to the high end of memory to avoid fragmentation. "moveHigh" parameter has no effect under MS-Windows.

UnlockBuffer( )

- **pascal void UnlockBuffer (BufferID buffer);**
  This is the corresponding routine to unlock a buffer. A buffer can be locked multiple times and only the final balancing unlock call will actually unlock it.

FreeBuffer( )

- **pascal void FreeBuffer (BufferID buffer);**
  This routine releases the storage associated with a buffer. Use of the buffer's ID after calling **FreeBuffer** will probably result in severe crashes.

BufferSpace( )

- **pascal int32 BufferSpace (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

Macintosh only.

The second suite of callback routine provides support for storing data with and retrieving data from a document. These routines essentially provide pseudo-resources which plug-ins 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. (**NOTE: Some sort of registry needs to be set up for resource types. No such registry yet exists. For now, please contact AppleLink: ADOBE.WISE or wise@mv.us.adobe.com to discuss registering a type. **)
The first fully functional version of the suite is version 3. The routines in that version are:

CountPIResources( )

- **pascal int16 CountPIResources (ResType ofType);**
  This routine returns a count of the number of resources of a given type.

GetPIResource( )

- **pascal Handle GetPIResource (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 host and should be treated as a read only handle.

DeletePIResource( )

- **pascal void DeletePIResource (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 renumber.

AddPIResource( )

- **pascal OSErr AddPIResource (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), this routine will return **memFullErr**.

Handle Suite

The use of handles in the pseudo-resource suite poses a problem for platforms other than the Macintosh where a direct equivalent may not exist. In those cases, Photoshop chooses a specific model for what it expects of a handle. The following suite of routines is used primarily for cross-platform support purposes since on the Macintosh, handles are handles. The one additional feature gained by using these routines rather than the Macintosh toolbox is that Photoshop will account for these handles in its VM space calculations. Hence, it is important to free any handles allocated using this suite by calling the free routine provided in this suite, etc..
Here are the routines in version 1 of the suite:

NewPIHandle ( )

- **pascal Handle NewPIHandle (int32 size);**
  This routine allocates a handle of the indicated size. It returns NULL if the handle could not be allocated.

DisposePIHandle ( )

- **pascal void DisposePIHandle (Handle h);**
  This routine disposes of the indicated handle.

GetPIHandleSize ( )

- **pascal int32 GetPIHandleSize (Handle h);**
  This routine returns the size of the indicated handle.

SetPIHandleSize ( )

- **pascal OSErr SetPIHandleSize (Handle h, int32 newSize);**
  This routine attempts to resize the indicated handle. It returns noErr if successful and an error code if unsuccessful.

LockPIHandle ( )

- **pascal Ptr LockPIHandle (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. This routine really only matters for cross platform implementations.

UnlockPIHandle ( )

- **pascal void UnlockPIHandle (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. This routine really only matters for cross platform implementations.

RecoverSpaceProc ( )

- **pascal 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) which you are supposed to dispose of using the DisposePIHandle callback but instead 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 stop reserving space for the handle.

General Notes

Macintosh

Global Variables

Most Macintosh development systems reference global variables by using negative offsets from register A5. If a plug-in were to try to use global variables in the standard way, its global variable space would overlap Adobe Photoshop's global variable space, usually resulting in a quick and fiery death.

The solution is to write the code in such a way as to not require global variables. In most cases, it is possible to replace global variables with additional procedure parameters. One case where this is impossible is with static data, which must be preserved between calls to the plug-in. Static data can be stored by allocating a handle, storing the static data in memory pointed to by the handle, and storing the handle in the data parameter, which Adobe Photoshop preserves between calls. This is the approach taken in the sample plug-in code with this kit since it allows the code to work equally well under any development environment.

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 the setting of optional compilation/link flags under most development environments if the resultant segment exceeds 32k.

About Boxes

All five kinds of plug-in should respond to a selector value of zero, which means display an about box. The plug-in actually has complete freedom to display any kind of about box it wishes, but to fit in smoothly with the Adobe Photoshop interface it should obey the following conventions:

1. It 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.
2. It 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.

If you have placed multiple plug-in modules in a single file, only one of them should display an about box, which should describe all of the modules. When Photoshop attempts to bring up the about box for a plug-in, it will make the about box call for all of the plug-ins in the same file.

Configuration

Photoshop plug-ins may assume 128K or larger ROMs, and System 6.0.2 or later. PiPL-only plug-ins (i.e., Photoshop 3.0 or later) may assume System 7. Keep in mind that Photoshop will run, and thus your plug-in may be called, on machines as old as the Mac Plus. Thus, plug-ins should not assume, and should check for if they require: 68020 or 68030 processors, math co-processors, 256K ROMs, and Color or 32-Bit QuickDraw. Photoshop 3.0 requires a 68020 or better, Color QuickDraw, and 32-Bit QuickDraw, so if you restrict your plug-in so that it only runs under Photoshop 3.0, you can assume these features are present.

Windows

Configuration

Photoshop plug-ins may assume Windows 3.1 in standard or enhanced mode and a 80386 processor.

About the Sample Plug-ins

Macintosh Version

The 6 sample plug-ins included with this kit can be built using MPW. They have been tested against MPW 3.3.1.
The kit 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 and such.
PIUtilities.c and PIUtilities.h contain various routines and macros to make it easier to use the host callbacks. The macros make various assumptions about how global variables are being handled. None of the routines worry about switching A5 worlds since the sample plug-ins do not use A5 worlds. If you do not follow the model for dealing with globals (basically not using them) used in the sample plug-in code, you will probably have to modify these files. Remember, it is VERY bad to call back to the host with the wrong A5 world!

Windows Version

The kit includes two sets of utilities: PIUtilities and Windows Utilities.

PIUtilities.c and PIUtilities.h contain various routines and macros to make it easier to use the host callbacks. The macros make variou assumptions about how global variables are being handled. If you do not follow the model for dealing with globals (basically not using them) used in the sample plug-in code, you will probably have to modify these files.
Winutils.c provides support for some Mac Toolbox functions used in PIUtilities.c, namely memory management functions (e.g NewHandle( ) etc.)
Structure packing for all records (i.e FilterRecord, FormatRecord, AcquireRecord, ExportRecord and AboutRecord) should be the default for the machine (this has changed for 32-bit plugins for speed reasons) however the Info structures (FilterInfo, FormatInfo etc.) must be packed to byte boundaries. This means 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, like say Symantec C++ or Borland C++, you have to modify the header files with appropiate pragmas. The Borland #pragmas still appear in the header files as they did in the 16-bit plugin kit, but are untested.

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. The most common problem is the use of "WORD wParam" for callback functions. The plugin examples use

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

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

The Windows kit also includes 2 executable utilties, namely, MacToDos.exe and CnvtPiPL.exe.
MacToDos.exe lets you convert Macintosh text files into PC text files. If you happen to pick up your plug-in kit from Adobe server, some source files may be in Macintosh format. You may have to convert them into PC format before compiling them.
CnvtPIPL.exe lets you convert PiPL resource in Macintosh format (ASCII format which conforms to the PiPL resource template) into Windows's PiPL format.
These two utilities are included under the Util sub-directory.

Acquisition Modules

Basics

The code resource and file type for acquisition modules is '8BAM'.

The AcquireRecord Structure

The stuff parameter contains a pointer to a structure of the following type:

```
typedef struct AcquireRecord
{
	int32			serialNumber;
	TestAbortProc	abortProc;
	ProgressProc		progressProc;
	int32			maxData;

	int16			imageMode;
	Point			imageSize;
	int16			depth;
	int16			planes;

	Fixed			imageHRes;
	Fixed			imageVRes;

	LookUpTable		redLUT;
	LookUpTable		greenLUT;
	LookUpTable		blueLUT;

	void *			data;

	Rect			theRect;
	int16			loPlane;
	int16			hiPlane;
	int16			colBytes;
	int32			rowBytes;
	int32			planeBytes;

	Str255			fileName;
	int16			vRefNum;
	Boolean			dirty;

	OSType			hostSig;
	ProcPtr			hostProc;

	int32			hostModes;
```

```
int16                   planeMap [16];

Boolean                 canTranspose;
Boolean                 needTranspose;

Handle              duotoneInfo;

int32                   diskSpace;

SpaceProc           spaceProc;

PlugInMonitormonitor;

void *              platformData;

BufferProcs *  bufferProcs;
ResourceProcs *   resourceProcs;

ProcessEventProc processEvent;

Boolean                 canReadBack;
Boolean                 wantReadBack;

Boolean                 acquireAgain;

Boolean                 canFinalize;

DisplayPixelsProc displayPixels;

HandleProcs * handleProcs;

Boolean             wantFinalize;

char            reserved1[3];

ColorServicesProc colorServices;

AdvanceStateProc advanceState;

char            reserved [216];

} AcquireRecord;
```

Record Fields

- **serialNumber**

This field contains Adobe Photoshop's serial number. Plug-in modules can use this value for copy protection, if desired.

- **abortProc**
  This field contains a pointer to the **TestAbort** callback documented in the general documentation.

- **progressProc**
  This field contains a pointer to the UpdateProgress callback documented in the general documentation. This procedure should only be called 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.

- **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 **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.

- **imageMode**
  The **acquireSelectorStart** 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.

- **imageSize**
  The **acquireSelectorStart** routine should set this field to inform Photoshop of the image's width (**imageSize.h**) and height (**imageSize.v**) in pixels.

- **depth**
  The **acquireSelectorStart** 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 except grayscale and RGB which also allow 16.

- **planes**
  The **acquireSelectorStart** 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. 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. The results would be unpredictable.

- **imageHRes**
- **imageVRes**
  The **acquireSelectorStart** 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.
  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.

- **redLUT**
- **greenLUT**
- **blueLUT**
  If an indexed color mode image is being returned, the **acquireSelectorStart** routine should return the image's color table in these fields.

- **duotoneInfo**
  If the plug-in is acquiring a duotone mode image, it should allocate a handle and return the duotone information here. The format of the information is the same as that provided by export modules, and should be treated as a black box by plug-ins.
  The plug-in is responsible for freeing the handle in its **acquireSelectorFinish** routine.

- **data**
  The **acquireSelectorContinue** routine should return a pointer to the image's data in this field. After all of the image has been returned, the **acquireSelectorContinue** should set this field to NULL.
  Note that the plug-in is responsible for freeing any memory pointed to by this field. This is a change from previous version's of Photoshop's plug-in interface.

- **theRect**
  The **acquireSelectorContinue** routine should set this field to the area being returned.

- **loPlane**
- **hiPlane**
  The **acquireSelectorContinue** routine 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.

- **colBytes**
  The **acquireSelectorContinue** routine 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.

- **rowBytes**
  The **acquireSelectorContinue** routine should set this field to the offset in bytes between rows of returned data.

- **planeBytes**
  The **acquireSelectorContinue** routine 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.

- **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 return 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. Note that attempts to index past the end of a planeMap will result in the identity map being used for the indexing.

- **fileName**
  By default, Photoshop opens newly acquired images as "Untitled-..." . File importing acquisition 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.

- **vRefNum**
  If the plug-in sets fileName, it should also set vRefNum to the file's volume reference number. Not applicable on MS-Windows.

- **dirty**
  By default, newly acquired images are marked as dirty, meaning that the user will be prompted to save the unsaved changes when closing the file. File importing acquisition modules should set this field to false to prevent this.

- **hostSig**
  The host program's signature. Photoshop's signature is **'8BIM'**.

- **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.

- **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 features (such as color scanning) if not supported by the host.

- **canTranspose**
  If the host supports transposing images during or after scanning, it should set this field to true. Photoshop always sets this field to true.

- **needTranspose**
  This field is 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 its **acquireSelectorStart** routine.
  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.

- **diskSpace**
  This field contains the number of free bytes on the host's scratch disk or disks. If the host does not use a scratch disk, it should set this field to -1.

- **spaceProc**

If not null, this field contains a pointer to a function with the following calling conventions:

- **pascal int32 SpaceProc (void);**

This function 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.

- **monitor**
  This field contains the monitor setup information for the host. See the general documentation for more details.

- **platformData**
  This field contains a pointer to platform specific data. Not used on the Macintosh.

- **bufferProcs**
  This field contains a pointer to the buffer suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **resourceProcs**
  This field contains a pointer to the pseudo-resource suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **processEvent**
  This field contains a pointer to the **ProcessEvent** callback documented in the general documentation. It contains null if the callback is not supported. This function is not useful on Windows.

- **canReadBack**
  If the 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.

- **wantReadBack**
  If the plug-in 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 simply be left untouched.

- **acquireAgain**

If the plug-in wishes to be called again to acquire another image, it should set this flag during the **acquireSelectorFinish** call. Host's that support multiple image acquisition should start the acquisition process again with a call to **acquireSelectorStart** to begin acquiring a new image. Plug-ins which do not want to put up a user interface for each acquisition should put up their interface during the **acquireSelectorPrepare** call. Plug-ins should not count on being called again just because they set this flag - i.e., **acquireSelectorFinish** should still do all of the necessary clean-up. With the addition of the finalize selector, plug-ins can now put up an interface that survives across multiple acquisitions.

- **canFinalize**
  If the host can make the finalize call, it should set this field to true.

- **displayPixels**
  This field contains a pointer to the DisplayPixels callback documented in the general documentation. It contains null if the callback is not supported.

- **handleProcs**
  This field contains a pointer to the handle suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **wantFinalize**
  This flag requests an acquireSelectorFinalize call if the host provides the newer protocol (**canFinalize**).

- **reserved1**
  This 3 byte field is used for alignment to a four-byte boundary.

- **colorServices**
  This field contains a pointer to the **ColorServices** callback documented in the general documentation. It contains null if the callback is not supported.

- **advanceState**
  The **advanceState** callback allows one to drive the interaction through the inner (**acquireSelectorContinue**) 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.

- **reserved**
  These are set to zero by the host for future expansion of the plug-in standard. Must not be used by plug-ins.

Calling Order
When the user invokes the plug-in by selecting its name from the Acquire submenu, Photoshop loads the plug-in's resource into memory and calls it with the following sequence of selector values (see the header file for their actual values):

(1) acquireSelectorPrepare
This allows the plug-in to adjust Photoshop's memory allocation algorithm. Before this call, Photoshop sets **maxData** to the maximum number of bytes it would be able to free up. The plug-in module then has the option of reducing this number during this call. Reducing the number can speed up operation, 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. Furthermore, in order to keep this amount of memory free, Photoshop is required to write any newly acquired image data to the virtual memory file as it is received.
If the plug-in knows that its memory requirements will be limited (if it can return the image data in strips, or if the maximum resolution image it can return is small), it should reduce **maxData** to its actual requirements during this call. This will allow small acquisitions to be performed entirely in RAM.

If, as is often the case, the plug-in only needs a small amount of memory, but will operate faster if given more, a tradeoff has to be made. One solution is to divide the **maxData** field by 2, thus allocating half the memory to Photoshop and half to the plug-in.
Another option is to reduce **maxData** to zero and then use the buffer and handle suites to allocate memory.

(2) acquireSelectorStart
This call lets Photoshop know 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, the plug-in module should set **imageMode**, **imageSize**, **depth**, **planes**, **imageHRes** and **imageVRes**. If an indexed color image is being returned, it should also set **redLUT**, **greenLUT** and **blueLUT**. If a duotone mode image is being returned, it should also set **duotoneInfo**. See the descriptions of the fields given above.

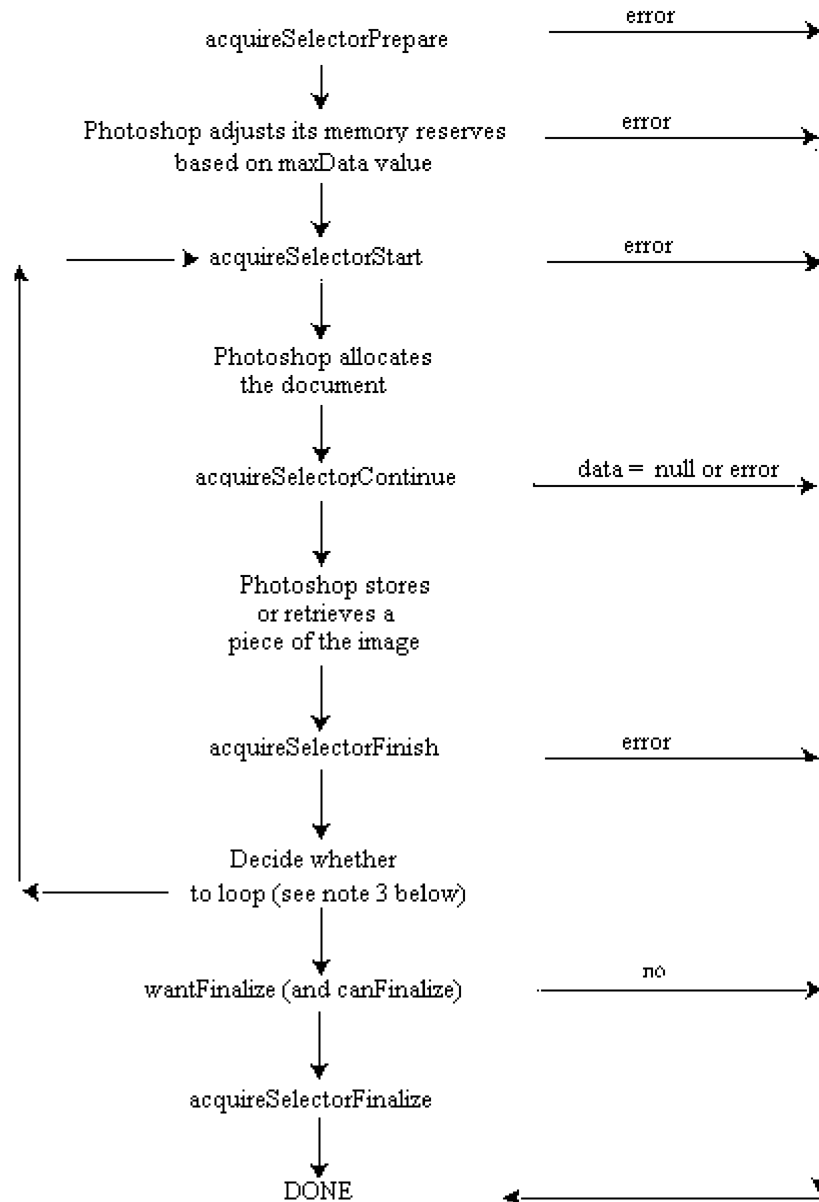
(3) acquireSelectorContinue
This call returns an area of the image to Photoshop. Photoshop will continue to call this routine until it either returns an error, or sets the **data** field to NULL.
The area of the image being returned is specified by **theRect** and by the **loPlane** and **hiPlane**. **data** contains a pointer to the actual data being returned. **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 (e.g., NTSC frame grabbers), the plug-in can simply set **theRect** to the entire image area. However, if you wish to be able to scan large images, the plug-in must use the **theRect** field to return the image pieces. There are no restrictions on how the pieces tile the image (i.e., 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 (or **GlobalAlloc( )** on Windows) or via the buffer suite. The plug-in is responsible for freeing this buffer in the **acquireSelectorFinish** call. (Note: this is a change from pre-version 3 interfaces, which freed the block for the plug-in!)

(4) acquireSelectorFinish
This call allows the 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 a command-period while processing the results of **acquireSelectorContinue** call, it will call the **acquireSelectorFinish** routine (be careful here, since normally the plug-in would be expecting another **acquireSelectorContinue** call).

(5) acquireSelectorFinalize
If the plug-in is using finalization (i.e., the host set **canFinalize** and the plug-in set **wantFinalize**), then thiscall will be made after all possible looping is complete.

State Machine

Photoshop plug-in acquire module's state machine



Notes:

1. If **acquireSelectorPrepare** succeeds -- i.e. , the result value is zero -- and **wantFinalize** is TRUE, then Photoshop guarantees that **acquireSelectorFinish** will be called.

2. If **acquireSelectorStart** succeeds then Photoshop guarantees that **acquireSelectorFinish** will be called.
3. Photoshop supports multiple document acquire which allows us to loop back to **acquireSelectorStart**. The simplest looping occurs after a successful acquisition sequence which ends with **acquireAgain** set TRUE. If the plug-in is using finalization (i.e., the host set **canFinalize** and the plug-in set **wantFinalize**), then we can also loop if **acquireAgain** is TRUE and the error code which terminated acquistion was > 0 or equalled **userCanceledErr**.
4. In the event of any error during acquisition, the document being acquired is discarded.
5. Hosts may choose to just treat **acquireAgain** as FALSE.
6. The plug-in can tell whether the host understands finalization by checking the **canFinalize** flag.
7. The **advanceState** callback allows one to drive the interaction through the inner (**acquireSelectorContinue**) loop without actually returning from the plug-in. If it returns an error, then the plug-in generally should treat this as an error **acquireSelectorContinue** continue and pass it on when it returns.

Error return values

The plug-in module may also 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
```

Sample Plug-in

DummyScan

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

Export Modules

Basics

The code resource and file type for export modules is **'8BEM'**.

The ExportRecord Structure

The **stuff** parameter contains a pointer to a structure of the following type:

```
typedef struct ExportRecord
{
        int32               serialNumber;
        TestAbortProc       abortProc;
        ProgressProc        progressProc;
        int32               maxData;

        int16               imageMode;
        Point               imageSize;
        int16               depth;
        int16               planes;

        Fixed               imageHRes;
        Fixed               imageVRes;

        LookUpTable         redLUT;
        LookUpTable         greenLUT;
        LookUpTable         blueLUT;

        Rect                theRect;
        int16               loPlane;
        int16               hiPlane;

        void *              data;
        int32               rowBytes;

        Str255              fileName;
        int16               vRefNum;
        Boolean             dirty;

        Rect                selectBBox;

        OSType              hostSig;
        ProcPtr         hostProc;

        Handle              duotoneInfo;
```

```
        int16               thePlane;

        PlugInMonitor       monitor;

        void *              platformData;

        BufferProcs    *    bufferProcs;
        ResourceProcs *     resourceProcs;

        ProcessEventProc    processEvent;

        DisplayPixelsProc   displayPixels;

        HandleProcs *       handleProcs;
        ColorServicesProc   colorServices;

        GetPropertyProc     getProperty;
        AdvanceStateProc    advanceState;

        int16               layerPlanes;
        int16               transparencyMask;
        int16               layerMasks;
        int16               invertedLayerMasks;
        int16               nonLayerPlanes;

        char                reserved [210];

    } ExportRecord;
```

Record Fields

- **serialNumber**
  This field contains Adobe Photoshop's serial number. Plug-in modules can use this value for copy protection, if desired.

- **abortProc**
  This field contains a pointer to the **TestAbort** callback documented in the general documentation.

- **progressProc**
  This field contains a pointer to the **UpdateProgress** callback documented in the general documentation. This procedure should only be called during the actual main operation of the plug-in, not during long operations during the preliminary user interface.

- **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 **exportSelectorPrepare** routine. The **exportSelectorContinue** routine

should process the image in pieces no larger than **maxData**, less the size of any large tables or scratch areas it has allocated.

- **imageMode**
  The mode of the image being exported (grayscale, RGB Color, etc.). See the header file for possible values. The **exportSelectorStart** should return an **exportBadMode** error if it is unable to process this mode of image.

- **imageSize**
  The image's width (**imageSize.h**) and height (**imageSize.v**) in pixels.

- **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.

- **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.

- **imageHRes**
- **imageVRes**
  The image's horizontal and vertical resolution in terms of pixels per inch. These are fixed point numbers (16 binary digits).

- **redLUT**
- **greenLUT**
- **blueLUT**
  If an indexed color or duotone mode image is being processed, these fields will contain its color table.

- **theRect**
  The **exportSelectorStart** and **exportSelectorContinue** routines should set this field to request a piece of the image for processing. It should be set to an empty rectangle when complete.

- **loPlane**
- **hiPlane**
  The **exportSelectorStart** and **exportSelectorContinue** routines should set these fields to the first and last planes to process next.

- **data**
  This field contains a pointer to the requested image data. If more than one plane has been requested (**loPlane** < > **hiPlane**), the data is interleaved.

- **rowBytes**
  The offset between rows for the requested image data.

- **fileName**

The name of the file the image was read from. File exporting modules should use this field as the default name for saving.

- **vRefNum**
  The volume reference number of the file the image was read from.

- **dirty**
  File exporting modules should clear this field to prevent the user being prompted to save any unsaved changes when the image is eventually closed.

- **selectBBox**
  The bounding box of the current selection. If there is no current selection, this is an empty rectangle.

- **hostSig**
  The host program's signature. Photoshop's signature is **'8BIM'**.

- **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.

- **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.

- **thePlane**
  Currently selected channel, or -1 if a composite color channel, or -2 if some other combination of channels.

- **monitor**
  This field contains the monitor setup information for the host. See the general documentation for more details.

- **platformData**
  This field contains a pointer to platform specific data. Not used on the Macintosh.

- **bufferProcs**
  This field contains a pointer to the buffer suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **resourceProcs**
  This field contains a pointer to the pseudo-resource suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **processEvent**

This field contains a pointer to the **ProcessEvent** callback documented in the general documentation. It contains null if the callback is not supported.

- **displayPixels**
  This field contains a pointer to the **DisplayPixels** callback documented in the general documentation. It contains null if the callback is not supported.

- **handleProcs**
  This field contains a pointer to the handle suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **colorServices**
  This field contains a pointer to the **ColorServices** callback documented in the general documentation. It contains null if the callback is not supported.

- **getProperty**
  This field contains a pointer to the property suite if it is supported by the host, otherwise null. (See **getProperty** documentation).

- **advanceState**
  The advanceState callback allows one 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. The structure is documented in the following fields:

- **layerPlanes**
  This field contains the number of planes of data possibly governed by a transparency mask.

- **transparencyMask**
  This field contains 1 or 0 indicating whether the data is governed by a transparency mask.

- **layerMasks**
  This field contains the number of layers masks (currently 1 or 0) for which 255 = fully opaque.

- **invertedLayerMasks**
  This field contains the number of layers masks (currently 1 or 0) for which 255 = fully transparent.

- **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

- **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.

Calling Order

When the user invokes the plug-in by selecting its name from the Export submenu, Photoshop loads the plug-in's resource into memory and calls it with the following sequence of selector values (see the header file for their actual values):

(1) exportSelectorPrepare

This allows the plug-in to adjust Photoshop's memory allocation algorithm. Before this call, Photoshop sets **maxData** to the maximum number of bytes it would be able to free up. The plug-in module then has the option of reducing this number during this call. Reducing the number can speed up operation, 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 the plug-in knows that its memory requirements will be limited (if it can process the image data in strips, or if the maximum resolution image it can process is small), it should reduce **maxData** to its actual requirements during this call. This will allow small exports to be performed entirely in RAM.

If, as is often the case, the plug-in only needs a small amount of memory, but will operate faster if given more, a tradeoff has to be made. One solution is to divide the **maxData** field by 2, thus allocating half the memory to Photoshop and half to the plug-in.

(2) exportSelectorStart

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

During this call, the plug-in 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 (i.e., horizontal and vertical strips are allowed as are a grid of tiles).

(3) exportSelectorContinue

During this routine, the plug-in should process the image data pointed to by data. It should then adjust **theRect**, **loPlane** and **hiPlane** to let Photoshop know what area of the image it wishes to process next. If the entire image has been processed, it should set **theRect** to an empty rectangle.

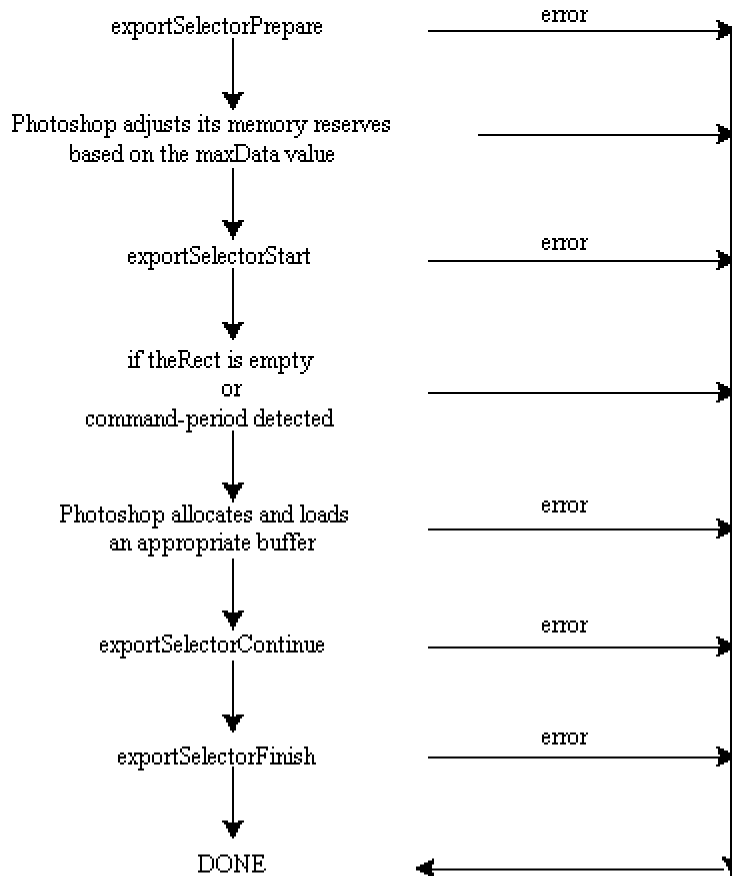
The requested image data is pointed to by data. If more than one plane has been requested (**loPlane** < > **hiPlane**), the data is interleaved. The offset from one row to the next is indicated by rowBytes. This is not necessarily equal to the width of **theRect** - there may be additional pad bytes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

(4) exportSelectorFinish

This call allows the plug-in 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 a command-period between calls to the **exportSelectorContinue** routine, it will call the **exportSelectorFinish** routine (be careful here, since normally the plug-in would be expecting another **exportSelectorContinue** call).

State Machine



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 also 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

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. Not applicable for the Windows platform.

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.).

Filter Modules

Basics

The code resource and file type for filter modules is **'8BFM'**.

The FilterRecord Structure

The stuff parameter contains a pointer to a structure of the following type:

```
typedef char FilterColor [4];

typedef struct FilterRecord
{
    int32            serialNumber;

    TestAbortProc    abortProc;
    ProgressProc     progressProc;

    Handle           parameters;

    Point            imageSize;

    int16            planes;

    Rect             filterRect;

    RGBColor         background;
    RGBColor         foreground;

    int32            maxSpace;
    int32            bufferSpace;

    Rect             inRect;

    int16            inLoPlane;
    int16            inHiPlane;

    Rect             outRect;

    int16            outLoPlane;
    int16            outHiPlane;

    Ptr              inData;
    int32            inRowBytes;

    Ptr              outData;
```

```
int32                   outRowBytes;

Boolean                 isFloating;

Boolean                 haveMask;
Boolean                 autoMask;

Rect                    maskRect;

Ptr                     maskData;
int32                   maskRowBytes;

FilterColor             backColor;
FilterColor             foreColor;

OSType                  hostSig;
ProcPtr             hostProc;

int16                   imageMode;

Fixed                   imageHRes;
Fixed                   imageVRes;

Point                   floatCoord;
Point                   wholeSize;

PlugInMonitor           monitor;

void *                  platformData;

BufferProcs *           bufferProcs;
ResourceProcs *         resourceProcs;

ProcessEventProc        processEvent;

DisplayPixelsProc       displayPixels;

HandleProcs *           handleProcs;

Boolean                 supportsDummyPlanes;

Boolean                 supportsAlternateLayouts;

int16                   wantLayout;

int16                   filterCase;
```

```
int16                   dummyPlaneValue;

void *                  premiereHook;

AdvanceStateProc        advanceState;

Boolean                 supportsAbsolute;

Boolean                 wantsAbsolute;

GetPropertyProc         getProperty;

Boolean                 cannotUndo;

Boolean                 supportsPadding;

int16                   inputPadding;

int16                   outputPadding;

int16                   maskPadding;

char                    samplingSupport;

char                    reservedByte;

Fixed                   inputRate;
Fixed                   maskRate;

ColorServicesProc       colorServices;

int16                   inLayerPlanes;
int16                   inTransparencyMask;
int16                   inLayerMasks;
int16                   inInvertedLayerMasks;
int16                   inNonLayerPlanes;

int16                   outLayerPlanes;
int16                   outTransparencyMask;
int16                   outLayerMasks;
int16                   outInvertedLayerMasks;
int16                   outNonLayerPlanes;

int16                   absLayerPlanes;
int16                   absTransparencyMask;
```

```
        int16               absLayerMasks;
        int16               absInvertedLayerMasks;
        int16               absNonLayerPlanes;


        int16               inPreDummyPlanes;
        int16               inPostDummyPlanes;

        int16               outPreDummyPlanes
        int16               outPostDummyPlanes;


        int32               inColumnBytes;
        int32               inPlaneBytes

        int32               outColumnBytes;
        int32               outPlaneBytes;

        char                reserved [134];
    } FilterRecord;
```

Record Fields

- **serialNumber**
  This field contains Adobe Photoshop's serial number. Plug-in modules can use this value for copy protection, if desired.

- **abortProc**
  This field contains a pointer to the **TestAbort** callback documented in the general documentation.

- **progressProc**
  This field contains a pointer to the **UpdateProgress** callback documented in the general documentation. 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.

- **parameters**
  Photoshop initializes this handle to NULL at startup. If a plug-in filter has any parameters that the user can set, it should allocate a relocatable block in the **filterSelectorParameters** routine, store the parameters in the block, and store the block's handle in this field.

- **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.

- **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.

- **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.

- **background**
- **foreground**
  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**.)

- **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**).

- **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.

- **inRect**
  The plug-in should set this field in its **filterSelectorStart** and **filterSelectorContinue** routines 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.

- **inLoPlane**
- **inHiPlane**
  The **filterSelectorStart** and **filterSelectorContinue** routines should set these fields to the first and last input planes to process next.

- **outRect**
  The plug-in should set this field in its **filterSelectorStart** and **filterSelectorContinue** routines 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.

- **outLoPlane**
- **outHiPlane**
  The **filterSelectorStart** and **filterSelectorContinue** routines should set these fields to the first and last output planes to process next.

- **inData**

This field contains a pointer to the requested input image data. If more than one plane has been requested (**inLoPlane** < > **inHiPlane**), the data is interleaved.

- **inRowBytes**
  The offset between rows of the input image data. (There may or may not be pad bytes at the end of each row.)

- **outData**
  This field contains a pointer to the requested output image data. If more than one plane has been requested (**outLoPlane** < > **outHiPlane**), the data is interleaved.

- **outRowBytes**
  The offset between rows of the output image data. (There may or may not be pad bytes at the end of each row.)

- **isFloating**
  This field is set true if and only if the selection is floating.

- **haveMask**
  This field is set true if and only if a non-rectangular area has been selected.

- **autoMask**
  By default, Photoshop automatically masks any changes to the area actually selected. If **isFloating** is false, and **haveMask** is true, the plug-in can turn off this feature by setting this field to false. It can then perform its own masking.

- **maskRect**
  If **haveMask** is true, and the plug-in needs access to the selection mask, it should set this field in its **filterSelectorStart** and **filterSelectorContinue** routines 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.

- **maskData**
  A pointer to the requested mask data.

- **maskRowBytes**
  The offset between rows of the mask data.

- **backColor**
- **foreColor**
  The current background and foreground colors, in the color space native to the image.

- **hostSig**
  The host program's signature. Photoshop's signature is '**8BIM**'.

- **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.

- **imageMode**
  The mode of the image being filtered (Gray Scale, RGB Color, etc.). See the header file for possible values. The **filterSelectorStart** should return a **filterBadMode** error if it is unable to process this mode of image.

- **imageHRes**
- **imageVRes**
  The image's horizontal and vertical resolution in terms of pixels per inch. These are fixed point numbers (16 binary digits).

- **floatCoord**
  If **isFloating** is true, the coordinate of the top-left corner of the floating selection in the main image's coordinate space.

- **wholeSize**
  If **isFloating** is true, the size in pixels of the entire main image.

- **monitor**
  This field contains the monitor setup information for the host. See the general documentation for more details.

- **platformData**
  This field contains a pointer to platform specific data. Not used on the Macintosh.

- **bufferProcs**
  This field contains a pointer to the buffer suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **resourceProcs**
  This field contains a pointer to the pseudo-resource suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **processEvent**
  This field contains a pointer to the **ProcessEvent** callback documented in the general documentation. It contains null if the callback is not supported.

- **displayPixels**
  This field contains a pointer to the **DisplayPixels** callback documented in the general documentation. It contains null if the callback is not supported.

- **handleProcs**

This field contains a pointer to the handle suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **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.

- **supportsAlternateLayouts**
  Does the host support data layouts other than rows of columns of planes? This field is set by the host to indicate whether it respects the **wantLayout** field.

- **wantLayout**
  The desired layout for the data. See **PIGeneral.h**. The host only looks at this field if it has also set **supportsAlternateLayouts**.

- **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.

- **dummyPlaneValue**
  The value to store into any dummy planes. 0..255 = specific value. -1 = leave undefined (i.e., random)

- **premiereHook**
  See the Adobe Premiere™ Plug-in Developer's Kit.

- **advanceState**
  See above.

- **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.

- **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.

- **getProperty**
  See **getProperty** in the general documentation section.

- **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.

- **inputPadding**
- **outputPadding**

- **maskPadding**
  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.

- **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*.

- **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 * inputRect). 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.

- **maskRate**
  Like inputRate, but as applied to the mask data.

- **inLayerPlanes**
- **inTransparencyMask**
- **inLayerMasks**
- **inInvertedLayerMasks**
- **inNonLayerPlanes**
  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.

- **outLayerPlanes**
- **outTransparencyMask**
- **outLayerMasks**
- **outInvertedLayerMasks**
- **outNonLayerPlanes**
  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.

- **absLayerPlanes**
- **absTransparencyMask**
- **absLayerMasks**
- **absInvertedLayerMasks**
- **absNonLayerPlanes**

The structure of the input data when wantsAbsolute is true.

- **inPreDummyPlanes**
- **inPostDummyPlanes**
  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.

- **outPreDummyPlanes**
- **outPostDummyPlanes**
  Like inPreDummyPlanes & inPostDummyPlanes except it applies to the output data.

- **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, the plug-in should assume that the host has not set it.

- **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, the plug-in should assume that the host has not set it.

- **outColumnBytes**
- **outPlaneBytes**
  The output equivalent of the previous two fields.

- **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.

Calling Order

When the user invokes the plug-in by selecting its name from the Filter submenu, Photoshop loads the plug-in's resource into memory and calls it with the following sequence of selector values (see the header file for their actual values):

(1) 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. Photoshop has a feature the repeats the most recent filtering operation using the current parameters, in which case this call is skipped.

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[a] 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.

Ideally, 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.
One other feature which can be put into plug-ins with respect to parameters is to store a default parameter handle in the plug-in's resource fork. This way, you can save preference settings for the plug-in across invocations of the host.

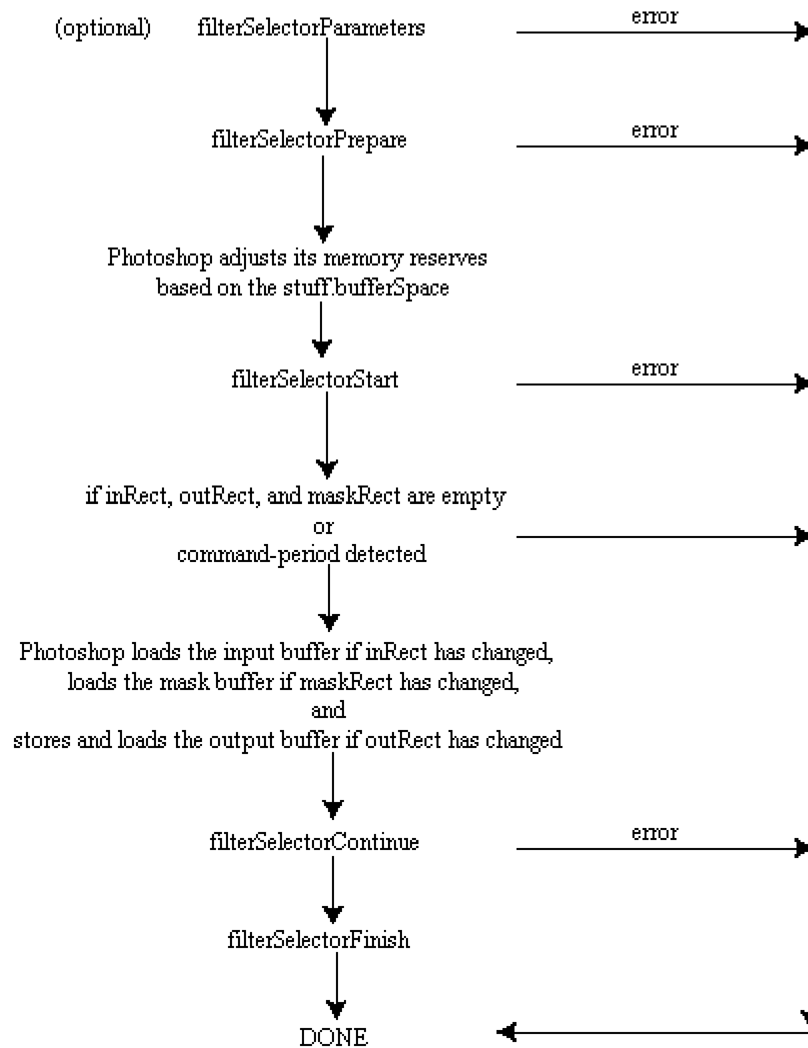
(2) filterSelectorPrepare
If the plug-in is planning on allocating any large (>= about 32K) buffers or tables, it should set the bufferSpace field to the number of bytes it is planning to allocate. Photoshop will then try to free up that amount of memory before calling the plug-in's **filterSelectorStart** routine. Alternatively, one can set this field to zero and use the buffer and handle suites if they are available.
The fields such as **imageSize**, **planes**, **filterRect**, etc. have now been defined, and can be used in computing the buffer size requirements.

(3) filterSelectorStart
The 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, the plug-in should process the image in pieces to minimize memory requirements. Unless there is a lot of startup/shutdown overhead on each call (e.g., talking to an external DSP), tiling the image with rectangles measuring 64x64 to 128x128 seems to work fairly well.

(4) filterSelectorContinue
This routine is repeatedly called as long as at least one of the **inRect** , **outRect**, or **maskRect** fields is non-empty.
This routine 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.

(5) filterSelectorFinish
This call allows the plug-in to clean up after a filtering operation. This call is made if and only if the **filterSelectorStart** routine returns without error (even if the **filterSelectorContinue** routine returns an error).
If Photoshop detects a command-period (i.e. user presses the Escape Key on 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).

State Machine



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 **filterSelectorFinish** 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 also 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

Sample Plug-in

Dissolve
: is a sample filter plug-in which demonstrates layers.

Image Format Modules

Basics

The code resource and file type for acquisition modules is '**8BIF**'.

The FormatRecord Structure

The **stuff** parameter contains a pointer to a structure of the following type:

```
typedef struct FormatRecord
	{

	int32			serialNumber;
	TestAbortProc		abortProc;
	ProgressProc		progressProc;
	int32			maxData;

	int32			minDataBytes;
	int32			maxDataBytes;
	int32			minRsrcBytes;
	int32			maxRsrcBytes;

	int32			dataFork;
	int32			rsrcFork;

	int16			imageMode;
	Point			imageSize;
	int16			depth;
	int16			planes;

	Fixed			imageHRes;
	Fixed			imageVRes;

	LookUpTable		redLUT;
	LookUpTable		greenLUT;
	LookUpTable		blueLUT;

	void *			data;

	Rect			theRect;
	int16			loPlane;
	int16			hiPlane;
	int16			colBytes;
	int32			rowBytes;
	int32			planeBytes;
```

```
	int16			planeMap [16];

	Boolean			canTranspose;
	Boolean			needTranspose;

	OSType			hostSig;
	ProcPtr		hostProc;

	int32			hostModes;

	Handle			revertInfo;

	NewPIHandleProc		newHandleProc;
	DisposePIHandleProc	disposeHandleProc;

	Handle			imageRsrcData;
	int32			imageRsrcSize;

	PlugInMonitor		monitor;

	void *			platformData;

	BufferProcs *		bufferProcs;
	ResourceProcs *		resourceProcs;

	ProcessEventProc	processEvent;

	DisplayPixelsProc	displayPixels;

	HandleProcs *		handleProcs;

	OSType			fileType;

	ColorServicesProc	colorServices

	AdvanceStateProc	advanceState;

	char			reserved [236]; /* Set to zero */

} FormatRecord;
```

Image Resources

Photoshop documents can have other properties associated with them besides pixel data. For example, we save page setup information and pen tool paths. Photoshop supports the concept of a block of data known as the image resources for a file. Image formats 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).

Record Fields

- **serialNumber**
  This field contains Adobe Photoshop's serial number. Plug-in modules can use this value for copy protection, if desired.

- **abortProc**
  This field contains a pointer to the **TestAbort** callback documented in the general documentation.

- **progressProc**
  This field contains a pointer to the **UpdateProgress** callback documented in the general documentation. This procedure should only be called during the actual main operation of the plug-in, not during long operations during the preliminary user interface.

- **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.

- **minDataBytes**
- **maxDataBytes**
  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.

- **minRsrcBytes**
- **maxRsrcBytes**
  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.

- **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.

- **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.

- **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**.

- **imageSize**
  The **formatSelectorReadStart** routine should set this field to inform Photoshop of the image's width (**imageSize.h**) and height (**imageSize.v**) in pixels. Photoshop will set this field before it calls **formatSelectorOptionsStart**, **formatSelectorEstimateStart**, or **formatSelectorWriteStart**.

- **depth**
  The **formatSelectorReadStart** routine should set this field to inform Photoshop of the image's resolution in bits per pixel per plane. The only valid settings are 1 for bitmap mode images, and 8 for all other modes. Photoshop will set this field before it calls **formatSelectorOptionsStart**, **formatSelectorEstimateStart**, or **formatSelectorWriteStart**.

- **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.

- **imageHRes**
- **imageVRes**
  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.

- **redLUT**
- **greenLUT**
- **blueLUT**
  If an indexed color mode image is being returned, the formatSelectorReadStart routine should return the image's color table in these fields. If an indexed color document is being written, Photoshop will set these fields before it calls formatSelectorOptionsStart, formatSelectorEstimateStart, or formatSelectorWriteStart.

- **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.

- **theRect**
  The plug-in should set this to the area of the image covered by the buffer specified in data.

- **loPlane**
- **hiPlane**

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.

- **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.

- **rowBytes**
  The start and continue routines should set this field to the offset in bytes between rows of data in the buffer.

- **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**
  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.

- **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.

- **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.

- **hostSig**
  The host program's signature. Photoshop's signature is '**8BIM**'.

- **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.

- **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.

- **revertInfo**
  This field is set to NULL by Photoshop when a format for a file is first created. If this field is defined on a **formatSelectorReadStart** call, then treat the call as a revert and don't query the user. If it is null on

the **formatSelectorReadStart** call, then query the user as appropriate and set up this field to store a handle containing the information necessary to read the file without querying the user for additional parameters (essential for reverting the file) and if possible to write the file without querying the user. The contents of this field are sticky to a document and will be duplicated when we duplicate the image format information for a document. On all **formatSelectorOptions** calls, leave **revertInfo** containing enough information to revert the document.
Photoshop will dispose of this field when it disposes of the document, hence, the plug-in must call on Photoshop to allocate the data as well using the following callbacks or the callbacks provided in the handle suite.

- **newHandleProc**
  This is the same as the **NewPIHandle** callback described in the general documentation. This field existed before the handle suite was defined.

- **disposeHandleProc**
  This is the same as the **DisposePIHandle** callback described in the general documentation. This field existed before the handle suite was defined.

- **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 **formatSelectorContinue** and the first time it is non-NULL parses the handle as a block of image resource data for the current document.

- **imageRsrcSize**
  This is the size of the handle in **imageRsrcData**. It is really only relevant during the estimate sequence when it is provided instead of the actual resource data.

- **monitor**
  This field contains the monitor setup information for the host. See the general documentation for more details.

- **platformData**
  This field contains a pointer to platform specific data. Not used on the Macintosh.

- **bufferProcs**
  This field contains a pointer to the buffer suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **resourceProcs**
  This field contains a pointer to the pseudo-resource suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **processEvent**
  This field contains a pointer to the **ProcessEvent** callback documented in the general documentation. It contains null if the callback is not supported.

- **displayPixels**
  This field contains a pointer to the **DisplayPixels** callback documented in the general documentation. It contains null if the callback is not supported.

- **handleProcs**
  This field contains a pointer to the handle suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **fileType**
  This field contains the file type for filtering.

- **colorServices**
  This field contains a pointer to the **ColorServices** callback documented in the general documentation. It contains null if the callback is not supported.

- **advanceState**
  The **advanceState** callback allows one 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.

- **reserved**
  Set to zero by the host for future expansion of the plug-in standard. Must not be used by plug-ins.

Calling Sequences

Image format plug-ins actually need to support a variety of selector calling sequences to support various pieces of the process of reading and writing files.

One sequence is used to read image files. It works much like acquire modules do and it should be relatively easy to convert acquire modules to read-only image formats.

Three sequences are used when writing a file. The first should be used to request save options from the user. It will only be used when first saving a file in a particular format. The second estimates the file size so that the host can decide whether there is enough disk space available. The last sequence actually writes the file.
All of these sequences follow the same general pattern:

(1) prepare
The prepare call is made to allow the plug-in to adjust Photoshop's memory allocation algorithm. Before this call, Photoshop sets maxData to the maximum number of bytes it would be able to free up. The plug-in module then has the option of reducing this number during this call. Reducing the number can speed up operation, 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. A smaller number will allow Photoshop to keep more image data in memory.
If the plug-in knows that its memory requirements will be limited (if it can return the image data in pieces, or if the biggest image it can return is small), it should reduce maxData to its actual requirements during this call. This will allow small acquisitions to be performed entirely in RAM.
If, as is often the case, the plug-in only needs a small amount of memory, but will operate faster if given more, a tradeoff has to be made. One solution is to divide **maxData** by 2, thus allocating half the memory to Photoshop and half to the plug-in.
If the plug-in can use the buffer and handle suites for all its memory allocation, this is even better. In this event, the plug-in should simply set **maxData** to zero.

(2) start
The start call allows the plug-in to begin its interaction with the host.
For formatSelectorReadStart, the plug-in should set **imageMode**, **imageSize**, **depth**, **planes**, **imageHRes** and **imageVRes**. If an indexed color image is being returned, it should also set **redLUT**, **greenLUT** and **blueLUT**. If the plug-in has a block of image resources it wishes processed it should set **imageRsrcData** to be a handle to the resource data.
For both reading and writing, the plug-in should set up the first input or output buffer as appropriate. The area of the image being returned or requested is specified by **theRect**, **loPlane**, and **hiPlane**. **data** contains a pointer to the actual pixels. **colBytes**, **rowBytes**, **planeBytes**, and **planeMap** specify the organization of the data.
For **formatSelectorReadStart** calls, the pixel data block should be filled in with the data to save. For **formatSelectorOptionsContinue**, **formatSelectorEstimateContinue**, and **formatSelectorWriteContinue** calls, the data block will be filled in with the requested pixel data at the beginning of the next call to the plug-in.
Photoshop is very flexible in the format in which image data can be transferred. For example, to return or request 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**). If instead, you wish to return or request 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.
The actual pixel data is stored in the block of memory pointed to by the contents of data. Most plug-ins will either use a **NewPtr** call or will allocate the memory using the buffer suite. The plug-in is responsible for freeing this memory during the finish call.

(3) continue
This call is used to process a sequence of areas within the image. The selected routine 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 or requested. The host will keep calling the continue routine until data is NULL.

(4) finish
The finish selector allows the plug-in to clean-up from the operation just performed. This call is made if and only if the start call returns without error (even if one of the continue calls results in an error.)
Most plug-ins will at least need to free the buffer used to return or request pixel data if this has not been done previously.

If Photoshop detects a command-period while processing the results of a continue call, it will call the finish routine. Be careful here, since normally the plug-in would be expecting another continue call. This is why it is frequently best to do all of one's clean-up in the finish call.

Error return values

The plug-in module may also 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	

Sample Plug-in

Sample Format

is a sample format module.

Document File Formats

Image Resource Block

The image resource data block stored in various files by Photoshop 3.0 is used to store non-pixel data which may be associated with an image such as pen tool paths. It is referred to as resource data because it holds data which would formerly have been stored in the resource fork of the file on the Macintosh.

It consists of successive blocks of data in the following format:

1. resource type (4 bytes, most often '**8BIM**')
2. resource ID (2 bytes)
3. resource name (a Pascal format string padded to make the size even)
4. resource size (4 bytes)
5. resource data (**resource size** bytes plus padding to make the size even)

Path Resource Format

Photoshop stores the paths saved with an image in the resource fork of the image file or in the image resource block. This document describes how to interpret and modify those paths.

(1) Photoshop stores its paths as resources of type '**8BIM**' with IDs in the range 2000 through 2998. Photoshop stores other information using resources of type '**8BIM**' so it is important to pay attention to the IDs. The name of the resource is the name given to the path when it was saved.

(2) 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.

Items 3 and 4 describe the path resource format in detail. The path format returned by GetProperty ( ) call is identical to what is described below (Please refer to Paths To Illustrator Sample).

(3) All points used in defining a path are stored as a pair of 32-bit components, vertical component first. The two components are fixed point numbers with 8 bits before the binary point and 24 bits after the binary point. We insist on leaving three guard bits 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. We include the lower bound but not the upper bound. We use such a limited range because we express the points 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 a LittleEndian machine (Intel platform), the byte order is reversed. You should swap the bytes before accessing it as int32.

(4) The data in a path resource consists of a sequence of 26 byte records.

A. The first two bytes (bytes 0 and 1) of each record are a 16-bit value which indicates the kind of data contained in the rest of the record. On a LittleEndian Machine (Intel platform), you should swap the bytes before accessing it as a short (int16).

B. If the kind value is 0, 1, or 2, then this record is part of the description of a closed subpath within the compound path.

1. If the kind value is 0, then bytes 2 and 3 of the record contain the length of the closed subpath. Such a record is then followed by records describing the knots of the subpath. This must be the first record in the subpath description.

2. If the kind value is 1 or 2, then the remaining 24 bytes of the record represent three points in the above format giving the control point for the Bezier segment preceding the knot, the anchor point for the knot, and the control point for the Bezier segment leaving the knot in that order. If the kind value is 1, the control points are linked; i.e., 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. If the kind value is 2, then this is a knot for which the control points are not linked.

C. If the kind value is 3, 4, or 5, then this record is part of the description of an open subpath within the compound path.

1. If the kind value is 3, then this is a path length record just like kind value 0.

2. If the kind value is 4, then this record contains the data for a knot with linked controls on the open subpath.

3. If the kind value is 5, then this record contains the data for a knot with non-linked control on the open subpath.

D. Further kind values may be added in the future. Since Photoshop will ignore records for which it does not understand the kind value, this is a relatively easy format to extend.

Photoshop 3.0

The Macintosh file type code is '**8BPS'**. The DOS extension is '.**PSD'**. All information is stored in big-endian byte order, so little-endian machines will have to swap bytes when reading and writing.

The 'File Info' in Photoshop 3.0 is stored in numerous places in a given file for various formats:

On the Mac the information is stored in the resource fork for any file format:

The keywords are stored in a 'KeyW' resource and the caption is stored in a 'TEXT' resource. These resources are referenced by the 'pnot' resource. This information is readable by Aldus Fetch. For more information on the format of these resources see:

Inside Macintosh: QuickTime Components
    and
Aldus Fetch Awareness Developer's Toolkit
  (206) 628-5693

All of the data from File Info is stored in an 'ANPA 1000' 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 see:

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

On all platforms, the data is also stored in the data fork of the file. For file formats that can support Photoshop's image resources the data is stored as an image resource '1028' (kCaptionID) in IPTC-NAA record 2 format.

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)

kCaptionID image resource

For old Photoshop files, the caption data is read from the image resource '1008' (kOldCaptionID) or '1020' (kPrintCaptionID) (it cannot appear in both). This data is appended to the caption data.

'pnot' resource related data (keywords and caption) (Mac only)

'ANPA' resource (Mac 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.

Whenever writing a file and skipping bytes, write zeros.

Whenever reading one of the length delimited sections, use the length to decide whether you should stop reading.

When writing one of these sections, it is usually a good idea to write the entire section as Photoshop may endeavor to read the whole thing.

The areas not stored for a layer mask are set to 255.

The areas not stored for a transparency mask are fully transparent.

The following sections describe the information stored in the file, in order.

1. Signature (4 bytes)
Always equal to '8BPS' for this format. Do not try to read the file if the signature does not match this value.

2. Version (2 bytes)
Always equal to 1 for this format. Do not try to read the file if the version number does not match this value.

3. Reserved (6 bytes)
Readers should ignore these bytes, and writers should write zeros.

4. Channels (2 bytes)

The number of channels in the image, including any alpha channels. Supported range is 1 to 24.

5. Rows (4 bytes)
The height of the image in pixels. Supported range is 1 to 30000.

6. Columns (4 bytes)
The width of the image in pixels. Supported range is 1 to 30000.

7. Depth (2 bytes)
The number of bits per channel. Supported values are 1, 8 and 16.

8. Image Mode (2 bytes)

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.

9. Color Data (4 bytes length + variable)
Contains the required data to define the color mode.

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

For duotone images, the mode data will contain the duotone specification, the format of which is not documented. Non-Photoshop readers can treat the duotone image as a grayscale image, and keep the duotone specification around as a black box for use when saving the file.

For all other modes, the byte count is zero.

10. Image Resources (4 bytes length + variable)
Successive blocks of data in the following format:

1. Resource Type (4 bytes)

2. Resource ID (2 bytes)

3. Resource Name (a Pascal format string padded to make the size even)

4. Resource Size (4 bytes)

5. Resource Data (Resource Size bytes plus padding to make the size even)

If the resource type is '**8BIM'** then:

If the resource ID is 2999, then this resources contains a Pascal-style string containing the name of the clipping path to use with this image when saving it as an EPS file.

If the resource ID is in the range 2000 - 2998 then the resource is a path resource. The name of the resource is the name given to the path when it was saved.

All points used in defining a path are stored as a pair of 32-bit components, vertical component first. The two components are fixed point numbers with 8 bits before the binary point and 24 bits after the binary point. We insist on leaving three guard bits 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. We include the lower bound but not the upper bound. We use such a limited range because we express the points 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.

A path resource consists of a sequence of 26 byte records as follows:

A. The first two bytes (bytes 0 and 1) of each record are a 16-bit value which indicates the kind of data contained in the rest of the record.

B. If the kind value is 0, 1, or 2, then this record is part of the description of a closed subpath within the compound path.

> **1.** If the kind value is 0, then bytes 2 and 3 of the record contain the length of the closed subpath. Such a record is then followed by records describing the knots of the subpath. This must be the first record in the subpath description.
>
> **2.** If the kind value is 1 or 2, then the remaining 24 bytes of the record represent three points in the above format giving the control point for the Bezier segment preceding the knot, the anchor point for the knot, and the control point for the Bezier segment leaving the knot in that order. If the kind value is 1, the control points are linked; i.e., 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. If the kind value is 2, then this is a knot for which the control points are not linked.

C. If the kind value is 3, 4, or 5, then this record is part of the description of an open subpath within the compound path.

> **1.** If the kind value is 3, then this is a path length record just like kind value 0.
>
> **2.** If the kind value is 4, then this record contains the data for a knot with linked controls on the open subpath.
>
> **3.** If the kind value is 5, then this record contains the data for a knot with non-linked control on the open subpath.

D. Further kind values may be added in the future. Since Photoshop will ignore records for which it does not understand the kind value, this is a relatively easy format to extend.

11. Misc Size (4 byte length + variable)

1. Layers Size (4 byte length)

Rounded to a multiple of 2.

Successive blocks of data in the following format:

2. Layer Count (2 bytes)

If negative then the first alpha channel is the merged transparency
channel and the actual layer count is the abolute value of the number.

3. for each layer:

1. Layer Top (4 bytes)

2. Layer Left (4 bytes)

3. Layer Bottom (4 bytes)

4. Layer Right (4 bytes)

The above describe the rectangle containing the contents of the layer.

5. Layer Channels (2 bytes)

The number of channels in the layer.

for each channel:

1. Channel ID (2 bytes)

0 = red, 1 = green, etc.
-1 = transparency mask
-2 = user supplied layer mask

2. Channel Data Length (4 bytes)

The length in bytes of the data for the channel.
See **Channel Data** below.

6. Blend Mode Signature (4 bytes)

always equal to '8BIM'.

7. Blend Mode Key (4 bytes)

'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

8. Opacity (1 byte)

0 = transparent .. 255 = opaque.

9. Clipping (1 byte)

0 = base,
1 = non-base.
In the future this may be extended to allow deeper nesting.

10. Flags (1 byte)

bit 0: transparency protected
bit 1: visible
bit 2: obsolete

11. Zero (1 byte)

12. Extra Data Size (4 bytes)

Extra Data Size bytes of data as follows:

1. Layer Mask Data Size (4 bytes)

Layer Mask Data Size bytes as follows:

A. Top (4 bytes)

B. Left (4 bytes)

C. Bottom (4 bytes)

D. Right (4 bytes)

E. Default Color (1 byte)

0 or 255.

F. Flags (1 byte)

bit 0: position relative to layer
bit 1: layer mask disabled
bit 2: invert layer mask when blending

G. Pad (2 bytes)

Zeros.

2. Layer Blending Ranges Length (4 bytes)

Layer Blending Ranges Length bytes as follows:

A. Grayscale Source Range (4 bytes)

Present but irrelevant for Lab & Grayscale.

Contains 2 black values followed by 2 white values.

B. Grayscale Destination Range (4 bytes)

C. First Channel Source Range (4 bytes)

D. First Channel Destination Range (4 bytes)

above repeated for remaining channels.

3. Layer Name (1 byte + variable)

Pascal style string containing the layer name and sufficient zeros to pad to a multiple of 4.

4. for each layer:

A. Channel Data (variable)

for each channel in the layer :

1. Compression (2 bytes)

0 = Raw Data,
1 = RLE compressed.

2. Image Data (variable)

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.

At this point, if the data since the **Layers Size** is odd, a pad byte will be inserted.

5. Layer Mask Alpha Size (4 bytes)

The next Layer Mask Alpha Size bytes have the following structure.

1. Overlay Color Space (2 bytes)

2. Color Components (8 bytes)

4 * 2 byte color components

3. Opacity (2 bytes)

0 = transparent,
100 = opaque.

4. Kind (1 byte)

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.

5. Pad (1 byte)

Zero.

12. Compression (2 bytes)

0 = Raw Data,
1 = RLE compressed.

13. Image Data (variable)

Image data is stored in planar order, e.g. all the red data, 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.

EPS

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> <blockÊsize> <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.

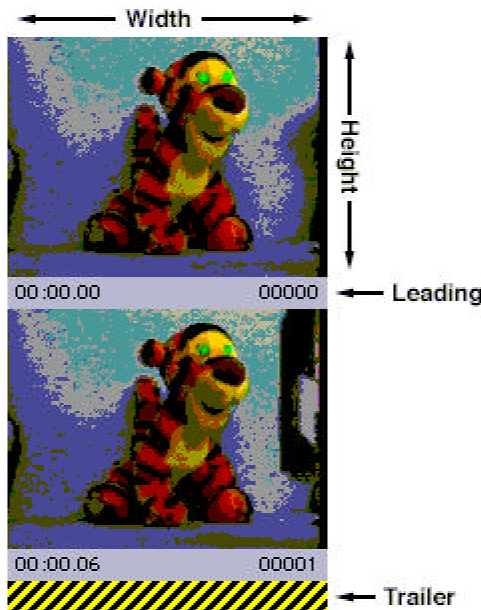
<binary/hex> is 1 if the data is in binary format, and 2 if the data is in hex format.

<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

Adobe Premiere 2.0 introduced a new file type: the Filmstrip. Premiere allows any video clip to be exported as a filmstrip. Adobe Photoshop 3.0 supports the filmstrip file type to allow each frame to be individually painted. The format for the filmstrip file is fairly simple, and is described below:

A Filmstrip consists of a sequence of equal sized 32-bit deep images, as shown in the picture below. The channel order in the file is Red, Green, Blue, Alpha. Between the frames is an arbitrarily sized leading area, in which any type of information may be embedded. Premiere puts the timecode and frame number for the frame in this area. This area is ignored by Premiere when the file is read, so the user is free to draw in this area. Following all the frames is a 16 row Trailer area the same width as the images. 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;
```

The fields are defined as follows:

- **signature**
  This field must be set to the code 'Rand' and is used to verify the validity of the record.

- **numFrames**
  This is the total number of frames in the file.

- **packing**
  This is the packing method used, currently only a value of 0 is defined, for no packing.

- **width**
  The width of each image, in pixels.

- **height**
  The height of each image, in pixels.

- **leading**
  The height of the leading areas, in pixels.

- **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, 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. Then fill in and write the FilmStrip record to the file.

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

TIFF

The same "Image Resources" information is stored in TIFF files under tag number 34377 as is stored in Photoshop 3.0 files (see Image Resource Block and Image Resources in the preceding sections).

The following table describes the standard TIFF tags and tag values that Photoshop 3.0 is able to read and write.

Tag	Reads	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 16	1 to 16
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

Load File Formats

Introduction

Many of Photoshop's image processing operations are controlled by dialogs that allow the saving of dialog settings into a file. These files can be loaded into the dialog at a later time, even for use in a different image. 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, there is no information stored in the resource forks of any 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), i.e. the reverse of the standard way this is done on Intel platforms (little-endian).

Arbitrary Map

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

The Macintosh file type code is '**8BLT**'. The Windows file name extension is "**.AMP**".

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 grayscale 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

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

The Macintosh file type code is '**8BBR**'. The Windows file name extension is "**.ABR**".

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

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.

The Macintosh file type code is '**8BCT**'. The Windows file name extension is **".ACT"**.

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

256 RGB colors are written one at a time, 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

Colors files are loaded and saved in Photoshop's Colors palette.

The Macintosh file type code is '**8BCO**' The Windows file name extension is **".ACO"**.

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 (2 bytes)
A short integer indicating the color space the color is in, referred to below as the space ID.

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.

RGB colors have a space ID of 0. 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).

HSB colors have a space ID of 1. 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 HSVColor data structure (e.g. pure red is defined as 0, 65535, 65535).

CMYK colors have a space ID of 2. 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).

Lab colors have a space ID of 7. 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).

Grayscale colors have a space ID of 8. 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. However, the following list gives the color space IDs currently defined by Photoshop for some custom color spaces:

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

Command Buttons

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.

The Macintosh file type code is '**8BFK**'. The file name extension is '**.ACM**'.

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 numnbers 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

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.)

The Macintosh file type code is '**8BSC**'. The Windows file name extension is **".CRV"**.

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

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

The Macintosh file type code is '**8BDT**'. The Windows file name extension is **".ADO"**.

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)
> 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 1 overprint color. For tritones, there are 4 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.

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

Halftone Screens

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

The Macintosh file type code is '**8BHS**'. The Windows file name extension is "**.AHS**".

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

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

The Macintosh file type code is '**8BHA**'. The Windows file name extension is **".HSS"**.

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

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

The Macintosh file type code is '**8BIC**'. The Windows file name extension is "**.API**".

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

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

The Macintosh file type code is '**8BCK**'. The Windows file name extension is "**.ACF"**.

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

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

The Macintosh file type code is '**8BLS**'. The Windows file name extension is "**.ALV**".

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)
Twenty-nine sets of levels. Each set of levels consists of 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

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

The Macintosh file type code is '**8BMS**'. The Windows file name extension is "**.AMS**".

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

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

The Macintosh file type code is **'8BXT'**. The file name extension is '**.AXT'**.

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

2. Color Space (2 bytes)
A short integer indicatin 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

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

The Macintosh file type code is '**8BSR**'. The file name extension is '**.ASR**'.

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

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

The Macintosh file type code is '**8BSV**'. The file name extension is '**.ASV'**.

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

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

The Macintosh file type code is '**8BSS**'. The Windows file name extension is "**.ASP**".

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

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

The Macintosh file type code is '**8BST**'. The Windows file name extension is **".AST"**.

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, we expect the file to have 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)
If the flag indicates the table is present, then the 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.

(i.e., to test the bit at bitIndex we use table [bitIndex >> 3] & (0x0080 >> (bitIndex & 0x07))) != 0. bitIndex itself is calculated in the same way one 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

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.

The Macintosh file type code is '**8BTF**'. The Windows file name extension is **".ATF"**.

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.