Adobe Photoshop - Software Development Kit Documentation - Macintosh Version.

The following Photoshop Macintosh version SDK documents have been consolidated into this single document. The entire Mac SDK including code samples is also available on a Macintosh formatted floppy disk through the Adobe Developers Association or through various on-line services.
(Note: This document is 63 pages long).

Refer to the PI_SDK_UNIX _2.5.2.ps addendum for supplemental information on developing Plug-ins for the UNIX versions of Photoshop.

Contents:

About the Sample Plug-ins

The sample plug-ins included with this kit can be built using either MPW C or Think C. They have been tested against MPW C 3.2.4 and Think C 5.0.

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 new header files for the respective types of plug-in modules.

The kit includes 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!

The kit includes code and resources for 5 plug-ins:

`DummyScan` is a sample acquire module. This is a new version of `DummyScan` which demonstrates the multiple acquire and read back facilities in 2.5.

`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.

`Dissolve` is a sample filter plug-in.

`SimpleFormat` is a sample format module.

Writing Plug-in Modules for Adobe Photoshop™

Introduction

Plug-in modules allow either Adobe Systems, Inc. or third-party developers to extend Adobe Photoshop, without actually modifying the base application. Adobe Photoshop version 2.5 supports five kinds of plug-in modules:

• Acquisition modules, which open an image in a new window. Acquisition modules car 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.

• Export modules, which output an existing image. Export modules can be used to print printers that do not have chooser-level driver support, or to save images in unsupported compressed file formats. These modules are accessed through the Export sub-menu.

• Filter modules, which modify a selected area of an existed image. These modules are inserted into the Filter menu.

• File format modules which provide support for additional image formats. These appea in the format pop-up in the Open As and Save As dialogs.

• Extension modules which patch some or all of Photoshop's core code to provide enhancements such as pressure pen support, access to additional memory, or acceleration. Documentation on extension modules is not included as part of this kit and available only through the Adobe Charged developers program.

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 w 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 completel different from that used by Silicon Beach. The differences were required primarily to support color images and Adobe Photoshop's virtual memory scheme.

Installation

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 t user's Photoshop preferences file.

When Adobe Photoshop starts executing, it searches the files in these folders, looking fo plug-in modules. When it finds a plug-in, it checks its version number, and if the version supported, it adds the plug-in's name to the appropriate menu or to the list of extensions be executed.

Each kind of plug-in module has its own 4-character code. For example, acquisition modules have the code `'8BAM'`. Adobe Photoshop searches for acquisition modules by

examining the resource fork of all files in the preferences folder that have file type `'8BAM'`,
for resources of type `'8BAM'`.

When it finds a resource of the correct type, is looks in the same file for a `'PiMI'` resource with the same resource ID. 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. Much of the information included here is new in 2.5 and is therefore optional. 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.

The structure of the general information is as follows:

1. major version (2 bytes)
   The major version number for the interface used by the plug-in. This field is required.

2. minor version (2 bytes)
   The minor version number for the interface used by the plug-in. This field is required.

3. priority (2 bytes)
   The priority which should be associated with this plug-in when it loads. Currently, this only used for extension modules.

4. general info size (2 bytes)
   The size of the general plug-in information in this resource.

5. type specific info size (2 bytes)
   The size of the type-specific plug-in information in this resource.

6. supported modes (2 bytes)
   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.).

7. required host proc (4 bytes)
   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 shoulc 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 fil 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 c do maintenance on their plug-in folders.

Execution

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 ir memory, locks it, and calls the routine starting at the first byte of the resource. It uses the following calling conventions:

```
pascal void PlugIn (short selector,
                    Ptr stuff,
                    long *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 depenc 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 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 shoulc 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 simply return a positive value and n report an error.) A negative value also means that execution of the plug-in should stop, b that the host should display its standard error dialog describing the error. Each plug-in ty has at least one plug-in specific error code (see the header files for details) which can be returned here. Standard Macintosh OS error codes such as `memFullErr` (-108) can also b used.

Callback Routines

A number of fields in the various plug-in "stuff" structures are callbacks to the host progra 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 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 2.5 or 2.5.1 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 wi be null. All of the routines use Pascal calling conventions. A complete list can be found i `PIGeneral.h`.

The two routines guaranteed to be present if defined in the plug-in interface record provic checking for command-period (or other requests to abort) and access to a host progress indicator:

• `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 b aborted. As a side effect, this changes to cursor to a watch and moves the watch hands periodically.

• `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 tc 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 2.5 provides a routine to allow some plug-in types to pass events to Photoshc 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 sequenc for this routine is:

• `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. 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 th case of Photoshop's callback, for example, this is a place where some callback operatior 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.

The final general callback routine is used to display pixels in various image modes. It is n supported in Photoshop 2.5 for the Macintosh but is supported in Photoshop 2.5.1 for Macintosh and Photoshop 2.5 for Windows. It takes a structure describing a block of pixe to display:

```
typedef struct PSPixelMap
    {
    int32 version;
    VRect bounds;
    int32 imageMode;
    int32 rowBytes;
    int32 colBytes;
    int32 planeBytes;
    void *baseAddr;
    }
PSPixelMap;
```

(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`.)

The fields in this record are as follows:

- `version`
  The version number for this structure. The current version number is version 0. Future versions of Photoshop may support additional parameters and will support higher versio numbers for `PSPixelMap`'s.

- `bounds`
  The bounds for the pixel map.

- `imageMode`
  The mode for the image data. The supported modes in Photoshop 2.5 are grayscale, RGB, CMYK, and Lab.

- `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.

The display pixels callback has the following calling sequence:

```
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.

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

- `dstRow` and `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. 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.

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. (W 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 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 i 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 buffe 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 til it keeps in memory.

Previous versions of the plug-in specification provide some mechanisms for interacting w 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.

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 requeste 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:

• `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` 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.

• `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 t avoid fragmentation.

• `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.

• `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.

• `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

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 ye exists. For now, please contact AppleLink: ADOBE.MARKH or mhamburg@adobe.com t discuss registering a type. **)

The first fully functional version of the suite is version 3. The routines in that version are:

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

• `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.

• `pascal void DeletePIResource (ResType ofType, int16 index);`
This routine deletes the resource that would have been returned by `GetPIResource`. Nc that since resources are identified by index rather than ID, this will cause subsequent resources to renumber.

• `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 2.5.1 is 1000), this routine will return `memFullErr`.

Handle Suite

The use of handles in the pseudo-resource suite poses a problem for platforms other tha the Macintosh where a direct equivalent may not exist. In those cases, Photoshop choos 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 handle 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:

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

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

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

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

- `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; in fact, it is unavailable in any version of Photoshop 2.5 o 2.5.1 for the Macintosh.

- `pascal void UnlockPIHandle (Handle h);`
  This routine unlocks the handle. Unlike the routines for buffers, the lock and unlock ca 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; in fact, it is unavailable in any version of Photoshop 2.5 or 2.5.1 for the Macintosh.

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 Setu dialog. The fields are as follow:

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

- `Fixed redX, redY, greenX, greenY, blueX, blueY;`
  These fields specify the chromaticity coordinates of the monitor's phosphors.

- `Fixed whiteX, whiteY;`
  These fields specify the chromaticity coordinates of the monitor's white point.

- `Fixed ambient;`
  This field specifies the relative amount of ambient light in the room. Zero means a relatively dark room, 0.5 means an average room, and 1.0 means a bright room.

Global Variables

Most Macintosh development systems reference global variables by using negative offse 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.

However, it is possible to use global variables in a plug-in module if done carefully. Thin C has a feature where is can use register A4 to access its global variables. See Think's documentation of details. Or if you are using MPW's C or Pascal compiler, Macintosh Technical Note 256 describes how to use global variables by setting up A5 yourself, and restoring it before exiting. Special care is required to make use of global variables in RO callback routines, such as userItem drawing procedures and dialog filter procedures. Se the example code for details.

Another solution is to write the code in such a way as not to require global variables. In most cases, it is possible to replace global variables with additional procedure paramete 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 dat 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 plu in code with this kit since it allows the code to work equally well under Think C and MPW

A special note should be made for extension plug-ins since they patch Photoshop's "bottleneck" table and their patches may need access to global variables. Since Photoshop does not pass a generic data parameter to the plug-in, extension modules cannot hang their global variables off this parameter as other modules can. Think C's A4 global support works because it keeps the global variables as part of the code resource. similar solution would have to be invented for other development environments.

Segmentation

Macintosh 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 w

The simplest way to solve this is to link all the plug-in's code into a single segment. This usually results in a limit of 32K of executable code for a plug-in, since most development systems use short branch instructions when making intra-segment routine calls.

If you are writing a large plug-in with more than 32K of executable code, one option is to use Think C, which supports A4-based jump tables, allowing multi-segment plug-ins. Se Think's documentation for details. Another options of this to use a recent version of MPW which supports branch islands.

About Boxes

All five kinds of plug-in should respond to a selector value of zero, which means display a 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:

- 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.

- It should not have an OK button, but should instead respond to a click anywhere in its dialog.

- 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 displ 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 ir the same file.

Configuration

Photoshop plug-ins may assume 128K or larger ROMs, and System 6.0.2 or later.

Keep in mind that Photoshop will run, and thus your plug-in may be called, on machines 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-l QuickDraw. Photoshop 2.5 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 2.5, you ca assume these features are present.

Acquisition Module Interface for Adobe Photoshop™

Introduction

This document describes version 4 of Adobe Photoshop's acquisition module interface. Plug-ins based on the following specifications should contain a 4 in their `'PiMI'` resource. There is no type specific info to put in the `'PiMI'` resource.

If the plug-in does not use the fields after the version 4 fields comment, it may choose to label itself a version 3 plug-in, to remain compatible with older versions of host software.

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

Record 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;

	/* Version 4 field follow */

	OSType			hostSig;
	ProcPtr			hostProc;

	int32			hostModes;

	int16			planeMap [16];

	Boolean			canTranspose;
	Boolean			needTranspose;

	Handle			duotoneInfo;

	int32			diskSpace;

	SpaceProc		spaceProc;

	PlugInMonitor		monitor;

	void *			platformData;

	BufferProcs *		bufferProcs;
	ResourceProcs *		resourceProcs;

	ProcessEventProc	processEvent;

	Boolean			canReadBack;
	Boolean			wantReadBack;
```

```
Boolean             acquireAgain;
Boolean             reservedFlag;
DisplayPixelsProc   displayPixels;
HandleProcs *       handleProcs;
char                reserved [228];
} AcquireRecord;
```

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 below.

(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, tc 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 car 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 dat: 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 via the buffer suite. The plug-in is responsible for freeing this buffer in the `acquireSelectorFinish` call. (Note: this is a change from pre-versior 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).

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 genera 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. 16-bit grayscale and 48-bit RGB mode can be specified either using the constants plugInModeGrayScale and plugInModeRGBColor together with a depth of 16 or using the constants plugInModeGray16 or plugInModeRGB48 (again with a depth of 16).

- `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, 8 or 16 for RGB and grayscale mode images, and 8 for all other modes.

- `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.

- `imageHRes` and `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`, and `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` and `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.

- `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.

- `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. I 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. In particular, through the bit positions for 16-bit grayscale and 48-bit RGB, the plug-in can determine whether the host supports images with 16 bits per component.

- `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.

- `canReadBack`
  If the host supports acquire modules reading back image data for further processing, it should set this field to true. Photoshop 2.5 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.

- `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).

- `reservedFlag` and `reserved`

These are set to zero by the host for future expansion of the plug-in standard. Must no be used by plug-ins.

Supporting Older Plug-in Versions

This section describes how hosts can be compatible with older versions of the plug-in specification. It is of interest only to authors of host software.

(1) If the version is less than 3, do not use `acquireSelectorPrepare`.

(2) If the version is less than 3, the host is required to dispose of the returned data (using `DisposPtr`) instead of the plug-in.

All other changes are backward compatible.

Export Module Interface for Adobe Photoshop™

Introduction

This document describes version 4 of Adobe Photoshop's export module interface. Plug-ins based on the following specifications should contain a 4 in their `'PiMI'` resource. There is nc type specific info to put in this resource.

If the plug-in does not use the fields after the version 4 fields comment, it may choose to label itself a version 3 plug-in, to remain compatible with older versions of host software.

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

Record 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;

char               reserved [232];

} ExportRecord;
```

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 int 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).

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 genera 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 and imageVRes

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

- redLUT, greenLUT, and 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 and 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).

- `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.

Supporting Older Plug-in Versions

This section describes how hosts can be compatible with older versions of the plug-in specification. It is of interest only to authors of host software.

(1) If the version is less than 3, do not use the `exportSelectorPrepare` selector.

(2) If the version is less than 4, Duotone mode images are exported as single channel Indexed Color mode images.

All other changes are backward compatible.

Filter Module Interface for Adobe Photoshop™

Introduction

This document describes version 4 of Adobe Photoshop's filter module interface. Plug-ins based on the following specifications should contain a 4 in their `'PiMI'` resource.

If the plug-in does not use the fields after the version 4 fields comment, it may choose to label itself a version 3.1 plug-in, to remain compatible with older versions of host software.

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

The resource name of the `'PiMI'` resource is used as the filter category name by Photoshop. If this resource has no name, the filter is placed in the "Other" filter category.

Filters optionally support a one byte flag in their type specific `'PiMI'` info which indicates whether the host should copy the source to the destination before running the plug-in. Plug-in filters are not required to write to all of their destination and hence the host must ordinarily copy the data. This flag allows one to turn that behavior off. A non-zero or absent value means that the host should copy. A zero means that it should not perform this copy.

Record 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;

    /* Version 4 fields follow */

    FilterColor         backColor;
    FilterColor         foreColor;

    OSType              hostSig;
    ProcPtr             hostProc;

    int16               imageMode;

    Fixed               imageHRes;
    Fixed               imageVRes;

    Point               floatCoord;
```

```
Point              wholeSize;

PlugInMonitor      monitor;

/* Photoshop 2.5 extensions */

void *             platformData;

BufferProcs *      bufferProcs;
ResourceProcs *    resourceProcs;

ProcessEventProc   processEvent;

DisplayPixelsProc  displayPixels;

HandleProcs *      handleProcs;

char               reserved [232];

} FilterRecord;
```

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 use input during play back. Adobe Premiere™ actually uses this feature. To be compatible with this feature, all parameters must be saved in a relocatable block whose handle is stored in the parameters field.

Ideally, the parameter block should contain the following information:

• A signature so that the plug-in can do a quick confirmation that this is, in fact, one of its parameter blocks.
• A version number so that the plug-in can evolve without requiring a new signature.

• 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 in Photoshop 2.5.

(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 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).

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 genera 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` and `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` and `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` and `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 thi field in its `filterSelectorStart` and `filterSelectorContinue` routines to request access to ar 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` and `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` and `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).

- `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.

Supporting Older Plug-in Versions

This section describes how hosts can be compatible with older versions of the plug-in specification. It is of interest only to authors of host software.

(1) If the version is less than 3.1, do not call the filter with CMYK mode images.

(2) If the version is less than 4, the plug-in is required to guess the imageMode based on the planes field, so alpha channels should not be included in the planes field, nor should HSB or HSL color images be filtered.

All other changes are backward compatible.

Image Format Module Interface for Adobe Photoshop™

Introduction

This document describes version 1 of Adobe Photoshop's image format plug-in module interface. Plug-ins based on the following specifications should contain a 1 in their `'PiMI'` resource.

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

`'PiMI'` Resource Structure

The `PiMI` resource contains information which is vital to Photoshop's ability to determine when to apply particular image format plug-ins. In addition to the general information documented in the general documentation, the type specific plug-in information should contain the following:

1. type count (2 bytes)
   The number of creator/file type pairs supported.

2. extension count (2 bytes)
   The number of three character file extensions supported.

3. can read (1 byte)
   Flag indicating whether this format module can be used to read files.

4. reads all types (1 byte)
   Flag indicating whether we should just ignore the list of creator/file type pairs at the end of the resource and just accept all files. Generally, you will not want to set this flag.

5. can write (1 byte)
   Flag indicating whether this format module can be used to write files.

6. can write if read (1 byte)
   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.

7. saves image resources (1 byte)
   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.

8. pad byte (1 byte)
   Padding. Reserved for future use. This should be set to 0 (false).

9. maximum channels (16 bytes)
   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 maxChannels [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.

10. maximum rows (2 bytes)
    The maximum number of rows that can be in an image saved in this format. Photoshop will use this field to screen out ineligible formats.

11. maximum columns (2 bytes)
The maximum number of columns that can be in an image saved in this format. Photoshop will use this field to screen out ineligible formats.

12. file type (4 bytes)
Macintosh file type to use when creating a file for this format.

13. creator type (4 bytes)
Macintosh creator type to use when creating a file for this format.

14. supported file type/creator pairs (8 bytes * type count)
A list of type count entries of file type and creator pairs which should be recognized as readable by this plug-in. A value of ‘   ’ indicates a “Don’t Care”.

15. supported 3 character extensions (4 bytes * extension count)
A list of extension count entries where the first three characters of each entry gives a possible DOS file name extension used to identify this file type. If this field contains any extensions, the first extension is taken as the default when creating files under Windows.

Interface Record 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;

char                    reserved [248];

} 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. In Photoshop 2.0, this information was stored in the resource fork of all of the file formats supported. This has the unfortunate consequence of making these files non-portable across platforms. Hence, Photoshop 2.5 supports the concept of a block of data known as the image resources for a file. The block will contain all of the information Photoshop would otherwise put in the resource fork. Image formats can store and retrieve this information if the file format definitior allows for a place to put such an arbitrary block of data (e.g., a TIFF tag or a PicComment).

If the format supports doing this, it should indicate so in its `'PiMI'` resource. If it does so, then on reads, it will be expected to supply a handle containing the block of data; on writes, it will be supplied with such a handle.

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. Mos 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 anothe continue call. This is why it is frequently best to do all of one's clean-up in the finish call.

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 genera 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` and `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` and `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.

- `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.

- `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`.

- `imageHRes` and `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`, and `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` and `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. I 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. The plug-in should use 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. 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).

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

Adobe Photoshop™ 2.0 and 2.5 EPS File Format

Photoshop 2.5.1 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 follo 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 rounde version of the high resolution bounding box for compatibility.

Photoshop 2.5 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 2.0 and 2.5 include 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 EP files, so Photoshop can read their files.

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

The comment is:

```
%ImageData: <columns> <rows> <depth> <mode> <pad channels> <block size> <binary/he
"<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 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> + 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.)

Adobe Photoshop™ 2.5 Image Resource Block Format

The image resource data block stored in various files by Photoshop 2.5 is used to store non-pixel dat may be associated with an image such as pen tool paths. It is referred to as resource data because data which would otherwise be 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)

The resolution and path information is stored using the same resource id and format as version 2.0 f used in the Macintosh resource fork.

Adobe Photoshop™ 2.0 and 2.5 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 tl 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 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.

(3) All points used in defining a path are stored as a pair of 32-bit components, vertical component first. ' two components are fixed point numbers with 8 bits before the binary point and 24 bits after the binary po We insist on leaving three guard bits in the points to eliminate most concerns over arithmetic overflow. He the range for each component is 0xF0000000 to 0x0FFFFFFF representing a range of –16 to 16. We inc 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 horiz component is given with respect to the image width. <0,0> represents the top-left corner of the image; <1, (<0x01000000,0x01000000>) represents the bottom-right.

(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.

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. Su 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 ab format giving the control point for the Bezier segment preceding the knot, the anchor point for the knot, ar the control point for the Bezier segment leaving the knot in that order. If the kind value is 1, the control pc are linked; i.e., editing one point edits the other one to preserve collinearity. Knots should only be marked having linked controls if their control points are collinear with their anchor. If the kind value is 2, then this is 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 ope subpath.

D. Further kind values may be added in the future. Since Photoshop will ignore records for which it does understand the kind value, this is a relatively easy format to extend.

Adobe Photoshop™ 2.0 File Format

Photoshop 2.0 format was designed to be fast and easy to read and write. Thus it is a very simple, uncompressed format.

The file type code is `'8BIM'`.

The file's data fork contains raw image data, in non-interleaved order. No pad bytes are used, except i Bitmap mode (where each line is padded to an even number of bytes).

The rest of the information for the file is stored in resources of type `'8BIM'` in the file's resource fork. M of the resources are optional, and if omitted, Photoshop will use reasonable default values.

Resource #1000: (Required)

This resource is 10 bytes long, containing 5 short integers. In order, they are: channels, rows, columns, depth, and mode.

"Channels" is the number of channels in the image, including alpha channels. (Range: 1 to 16)

"Rows" is the height of the image in pixels. "Columns" is the width of the image in pixels. (Range: 1 to 30000).

"Depth" is number of bits per pixel per channel. This is 1 for Bitmap mode images, and 8 for all other modes.

"Mode" indicates the mode of the image. Defined values are: 0 = Bitmap, 1 = Gray Scale, 2 = Indexed Color, 3 = RGB Color, 4 = CMYK Color, 5 = HSL Color, 6 = HSB Color, 7 = Multichannel, 8 = Duotone.

Resource #1001: (Optional)

The standard Macintosh 120 byte print record.

Resource #1003: (Required for Indexed Color Images)

Contains the indexed color table. It is 768 bytes long. The order is 256 red values, 256 green values, 256 blue values.

Resource #1005: (Optional)

Resolution information. Format is:

```
typedef struct ResolutionInfo
      {

      Fixed hRes;
      int16 hResUnit;
      int16 widthUnit;

      Fixed vRes;
      int16 vResUnit;
      int16 heightUnit;

      } ResolutionInfo;
```

`hRes` and `vRes` are always stored in units of pixels/inch. `hResUnit` and `vResUnit` are 1 if the resolutic should be displayed in pixel/inch, or 2 if it should be displayed in units of pixels/cm. `widthUnit` and `heightUnit` are 1 for inches, 2 for cm, 3 for points, 4 for picas, or 5 for columns.

Resource #1008: (Optional)

The caption. A Pascal format string with maximum length of 255.

Resource #1018: (Required for Duotone mode images)

Duotone specification. Internal format is not documented, treat as a black box.

Resources #2000-2999: (Optional)

Path resources. See separate documentation.

Adobe Photoshop™ 2.5 File Format

Photoshop 2.5 format was designed to be fast and easy to read and write, while solving a couple of problems with Photoshop 2.0 format:

(1) It is data fork only, thus cross-platform.

(2) It uses optional RLE compression, so simple images and mask channels can be saved much mor compactly.

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

The following sections describe the information stored in the file's data fork, 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

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

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 16 for Photoshop 2.5.

5. Rows (4 bytes)

The height of the image in pixels. Supported range is 1 to 30000 for Photoshop 2.5.

6. Columns (4 bytes)

The width of the image in pixels. Supported range is 1 to 30000 for Photoshop 2.5.

7. Depth (2 bytes)

The number of bits per channel. Supported values are 1 and 8 for Photoshop 2.5.

8. Mode (2 bytes)

The color mode of the file. Supported values are: Bitmap = 0, Grayscale = 1, Indexed Color = 2, RGI = 3, CMYK Color = 4, Multichannel = 7, Duotone = 8, Lab Color = 9.

9. Mode Data (4 byte 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 tak 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 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 byte length + variable)

Contains a Photoshop 2.5 image resources block, the format of which is documented separately. Information contained in this block includes the image's resolution, and pen tool paths.

11. Reserved Data (4 byte length + variable)

Reserved for future use. Readers should skip this, and writers should write a zero length block.

12. Compression (2 bytes)

Two values are currently define: 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 store 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 line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine PackBits, and the TIFF standard.

Many of Photoshop's image processing operations are controlled by dialogs that allow the saving of dia 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 th 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).

Adobe Photoshop™ 2.5
Arbitrary Map Load File Format

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

The Macintosh file type code is `'8BLT'`. The 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 las byte to 255 in the image data. A "null" table that has no effect on an image is a linear table of bytes fro 0 to 255.

If there is one table in the file, Photoshop 2.5 applies it to the master composite channel, if the image h one, or to the single active channel if there is only one. If there is no composite channel, but more thar 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 i treated as a master and the remainder are applied to the image channels in turn (i.e. the second map i associated with the first channel, the third map with the second channel, etc.)

Photoshop 2.5 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.

Adobe Photoshop™ 2.5
Brushes Load File Format

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

The file type code is `'8BBR'`. The 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 us 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 brus 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 r anti-aliasing. (Note that brushes with sampled data size either taller or wider than 32 pixels will nc 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 (8 bytes): a rectangle, exactly repeating the previous bounds entry

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 compress

b. data (variable): the brush tip image data is a single plane of grayscale data, stored in scanl 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 t 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 PackBit and the TIFF standard.

Adobe Photoshop™ 2.5
Color Table Load File Format

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 file type code is `'8BCT'`. The 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 byt 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.

Adobe Photoshop™ 2.5
Colors Load File Format

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

The file type code is `'8BCO'`. The 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 fc 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 re green, and blue components. They are full unsigned 16-bit values as in Apple's RGBColor data structu (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 hu saturation, and brightness components. They are full unsigned 16-bit values as in Apple's HSVColor da 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 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 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 a streamed to and from load files. The details of a custom color's color data fields are not public and sho 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

Adobe Photoshop™ 2.5
Curves Load File Format

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 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 19.

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 rang 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 f 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 u to nineteen curves, is when the original image is indexed color. In this case three curves are created fo 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 imag any alpha channel that is active corresponds to its channel number plus three (e.g. if channel two is act 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 us on that channel. This allows easy application of a single file to both RGB and Grayscale images.

Note that Photoshop 2.5 can write Curves files that Photoshop 2 will not be able to read, because Photoshop 2.5's active channel support is different from Photoshop 2.0's, and there could be more acti channels in a Curves dialog than 2.0 supported. Photoshop 2.5 will always write at least five curves to 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.

Adobe Photoshop™ 2.5
Duotone Options Load File Format

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

The Macintosh file type code is `'8BDT'`. The 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, 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 w 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 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 fi 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 2.5

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 2.5.

7. Overprint Colors (11 * 10 bytes)

Eleven ink colors, regardless of the number of plates. The number of defined overprints depends on th number of plates, Count. For monotones, there are no overprint colors. For duotones, there is 1 overp 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.

Adobe Photoshop™ 2.5
Function Keys Load File Format

Function Keys (FKeys) settings files are loaded and saved in Photoshop's Function Keys Preferences dialog.

The Macintosh file type code is `'8BFK'`. There is currently no file name extension assigned to these fil as the function keys feature is not supported by Photoshop on Windows.

1. Version (2 bytes)

Equal to 1, written as a short integer.

2. Command Strings (variable)

The remainder of the file consists of 30 Pascal-style strings, with no pad bytes between strings. The firs fifteen correspond to the function keys without the shift key. The strings reference the text of Photosho menu items. The command strings cannot be longer than 63 characters each (64 bytes counting the initial length byte).

Adobe Photoshop™ 2.5
Halftone Screens Load File Format

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

The Macintosh file type code is `'8BHS'`. The 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. Values range from 1000 to 999999, representing 1.0 t 999.999 lpi.

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 the screen, ranging from -1800000 to 1800000 (representing -180.0000° to 180.0000°).

d. shape code (2 bytes)
A code representing the shape of the halftone dots in this screen. Round = 1, Diamond = 2, Ellipse 3, Line = 4, Square = 5, Cross = 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 belo

e. miscellaneous (4 bytes)
For compatability, this should be set to 0. It is not used by Photoshop 2.5.

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. T 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 co for those screens that have custom functions provides enough information to separate the various functions and assign them.

Adobe Photoshop™ 2.5
Hue/Saturation Load File Format

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

The Macintosh file type code is `'8BHA'`. The 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 write 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 firs 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 zer 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 CM 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 othe

two values are ignored (Photoshop sets them to zero). (The values that are used range from -90 to in the user interface.)

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

Adobe Photoshop™ 2.5
Ink Colors Setup Load File Format

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

The Macintosh file type code is `'8BIC'`. The 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 a specified in the order Cyan, Magenta, Yellow, Magenta-Yellow (Red), Cyan-Yellow (Green), Cyan-Mage (Blue), Cyan-Magenta-Yellow, followed by the White and Black points. Each triple is written in the order (range: 0 to 10000, representing 0.0 to 1.0000), y (range: 1 to 10000, representing 0.0001 to 1.0000), (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 rang 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%).

Adobe Photoshop™ 2.5
Custom Kernel Load File Format

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

The Macintosh file type code is `'8BCK'`. The 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 imag pixel. The values progress through horizontal offsets first, e.g. the first five values all represent a vertica 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.

Adobe Photoshop™ 2.5 Levels Load File Format

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

The Macintosh file type code is `'8BLS'`. The file name extension is `'.ALV'`.

There are two versions of this file format. Photoshop 2.5 reads both but only writes version 2.

1. Version (2 bytes)

Equal to 2, written as a short integer.

2. Levels Records (190 bytes)

Nineteen sets of levels. Each set of levels consists of five short integers, in ten bytes. The first number a set is the input floor setting, and must range from 0 to 253. The second number is the input ceiling, a must range from 2 to 255. Third is the output value to which the input floor willbe matched. It can rang from 0 to 255. Fourth is the ceiling output, also ranging from 0 to 255. The fifth value is the gamma to 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 RGB document). The remaining sets apply to the active channels individually-- the second set applies t channel one (if it is an active channel), the third set to channel two, etc., up until the seventeenth set, which applies to channel sixteen. The exception to this, and the reason there are nineteen sets of leve is when the original image is indexed color. In this case three sets of levels are created for the red, gre and blue portions of the image's color table, and they replace the levels that represent the first channel 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 is corresponds to set number 5).

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

Adobe Photoshop™ 2.5
Monitor Setup Load File Format

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 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 t 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 come red x, then red y; then green x, etc. The x values range from 0 to 10000 (representing 0.0 to 1.0000); t y values range from 1 to 10000 (representing 0.0001 to 1.0000).

Adobe Photoshop™ 2.5
Separation Setup Load File Format

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 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 Ty[p]e is set to UCR.

Adobe Photoshop™ 2.5
Separation Tables File Format

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

The Macintosh file type code is `'8BST'`. The file name extension is `'.AST'`.

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. Th 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.

Adobe Photoshop™ 2.5
Transfer Function Load File Format

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 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 valu 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 t user interface.