

Adobe Premiere™ Plug-In Tool kit

v1.0

written by Randy Ubillos
©1992 Adobe Systems, Inc.

Writing Premiere™ Plug-Ins

This document is designed for those people wishing to write their own plug-in special effects and filters for use with Adobe Premiere™. Premiere plug-ins are separate files with embedded code resources which are called to perform the useful work of the filter or effect. Filters and Effects use slightly different calling parameters, but both are similar in nature to Adobe Photoshop™ Plug-ins.

The plug-in folder is scanned by Premiere at startup time for files of type 'SPFX', 'VFLT' and 'AFLT'. Any files matching these types are further examined to determine if they are valid. Valid plug-ins are added to the appropriate internal lists and displayed to the user.

The plug-in code must be a stand-alone code module. When called, the current resource fork will be the plug-in's, and the code module is free to load in whatever resources it needs. If the code module needs to store defaults beyond those which go in the `specsHandle` field, it can modify its own resource fork for later retrieval of parameters. Whenever possible, it is very desirable to design any dialogs so that the user is given direct visual feedback while any settings are being modified.

Special Effects

An effect is passed two 32-bit deep GWorlds containing the source images and a third GWorld for the storage of the result. The total duration of the effect and the current frame number are used to calculate what the current frame should look like. The effect code combines the two source images into the destination image and returns. Premiere handles all of the overhead for retrieving the frames and storing them.

Premiere also handles the generation and display of the animated effect icons in the F/X window. Premiere does this by calling the `esExecute` routine several times with icon sized images. The resultant effect icons are compressed and stored in 'ICNc' resources in the plug-in's resource fork. If the code for an effect is changed, these resources should be removed so that they will be rebuilt.

Special effects are stored in files of type 'SPFX'. They contain several resources which Premiere looks for to identify the file as an effect:

'FXvs' #1000

This resource is one word in length and contains the version of the effect. The current version is 1.

'TEXT' #1000

This is a string resource which contains the plain text description of the effect which will be displayed alongside the effect's icon in the F/X window.

'Fopt' #1000

This resource describes the features which an effect supports. The Rez template for this type is:

```
type 'Fopt' {
    byte;                                /* Valid mask
    byte;                                /* Initial values
    byte                                  No = 0, Yes = 1, Custom = 2;    /* Has Custom, In Pairs
    byte                                  /*
    byte                                  No = 0, Yes = 1;                /* Exclusive?
    byte                                  No = 0, Yes = 1;                /* Reversible?
    byte                                  No = 0, Yes = 1;                /* Has Edges?
    byte                                  No = 0, Yes = 1;                /* Has Start Point?
    byte                                  No = 0, Yes = 1;                /* Has End Point?
};
```

The first byte is a mask describing which of the edge indicator controls will be displayed for the effect. They are composed of:

```
enum {
    bitTop = 0x01,
    bitRight = 0x02,
    bitBottom = 0x04,
    bitLeft = 0x08,
    bitUpperRight = 0x10,
    bitLowerRight = 0x20,
    bitLowerLeft = 0x40,
    bitUpperLeft = 0x80
};
```

The second byte is a mask indicating the initial state of all eight of the indicators. The initial mask must be a valid combination of the bits indicated in the previous byte.

The third byte is a flag in which bit 0 is set if the indicators should operate in pairs, or cleared if they should operate one at a time. Bit 1 is set if the effect can put up a custom settings dialog. Bit 2 is set if the effect does not change over time.

The fourth byte is a flag in which bit 0 indicates if the indicators should operate exclusively. If this is set, clicking on one indicator or set of indicators will cause all others to be turned off. If this bit is clear, each indicator toggles on and off independently each time it is clicked.

The fifth byte is set to 1 if the effect should have the reverse button displayed.

The sixth byte is set to 1 if the effect has hard edges which can have borders or anti-aliasing applied to them.

The seventh byte is set to 1 if the effect has a movable open point.

The eighth byte is set to 1 if the effect has a movable close point.

'SPFX' #1000

This is the actual code module. This module is loaded in and the first byte of code is called using the following convention:

```
typedef pascal short (*EffectProcPtr)(short selector, EffectHandle theData);
```

The selector parameter can be one of the following:

```
enum {  
    esExecute = 0,  
    esSetup  
};
```

`esExecute` indicates that the code should process the input parameters and generate an output frame. The `specsHandle` parameter contains all of the relevant information about how the source frames should be processed to generate the destination frame.

`esSetup` should display any dialog box which the effect wishes to use to prompt the user for additional parameters. This call should use any data contained in the `specsHandle` parameter to set up the dialog defaults, and it should place any data pertaining to the user selected settings into `specsHandle`. `specsHandle` will be nil initially. It is up to the plug-in to create the properly sized handle and place it into the parameter record.

The parameter record for an effect is:

```
typedef struct {
    Handle specsHandle; // specification handle
    GWorldPtr source1; // source GWorld #1
    GWorldPtr source2; // source GWorld #2
    GWorldPtr destination; // Destination GWorld
    long part; // part / total = % complete
    long total;
    char previewing; // in preview mode?
    char arrowFlags; // flags for direction arrows
    char reverse; // is effect being reversed?
    char source; // are sources swapped?
    Point start; // starting point for effect
    Point end; // ending point for effect
    Point center; // the reference center point
    Handle privateData; // Editor private data handle
    FXCallbackProcPtr callBack; // callback, not valid if nil
} EffectRecord, **EffectHandle;
```

The fields are used as follows:

specsHandle: The effect defined data handle which contains all of the current settings for this effect. The effect normally creates and modifies this handle during an `esSetup` call. Premiere saves this handle in project files so that the settings will be restored when a project is re-opened.

source1: Points to source GWorld 1. This GWorld contains all of source image 1, and will always be 32 bits deep.

source2: Points to source GWorld 2. This GWorld contains all of source image 2, and will always be 32 bits deep and the same size as source2.

destination: Points to the destination GWorld. This is where the resultant frame should be stored. It will always be 32 bits deep and the same size as GWorlds 1 and 2. When processing the source data into the destination, the alpha channel in the source images may contain useful data, and should be processed just like the other 3 channels. If the alpha channel is distorted, colored edges and anti-aliasing may not work properly for the effect.

part: indicates the particular frame which is being processed.

total: indicates the total number of frames which are being processed. **part** divided by **total** gives the percentage of the effect which should be currently calculated. **part** will vary from 0 up to and including **total**.

previewing: set if the effect is being called to render a preview quality effect. If the effect can render a faster, less accurate representation of the effect, it should do so when this flag is set.

arrowFlags: The current edge indicator flags, as set by the user. The bit definitions are the same as for the 'Fopt' resource above.

reverse: Set if the effect is being run in reverse. Premiere handles swapping the order in which frames are handed to the effect automatically. This flag is for informational purposes, and the effect should not normally look at this flag.

source: Set if the sources to the effect are swapped (i.e. B --> A instead of A --> B). Premiere handles swapping the input source GWorlds automatically. This flag is for informational purposes, and the effect should not normally look at this flag.

start: If the flag in the 'Fopt' record is enabled, this will be a point defining the opening spot for the effect. It should be measured relative to the **center** parameter below.

end: If the flag in the 'Fopt' record is enabled, this will be the point defining the closing spot for the effect. It should be measured relative to the **center** parameter below.

center: The normal center point for opening and closing.

privateData: Passed to the callback below.

callback: This is a pointer to a callback routine which can be used to retrieve past or future frames from the source clips. This field is nil during the `esSetup` call; alternate frames may only be requested during an `esExecute` call. It's calling convention is:

```
typedef pascal short (*FXCallbackProcPtr)(long frame, short track, CGrafPtr
                                          thePort, Rect theBox, Handle
privateData);
```

The parameters are: `frame` is the desired frame, from 0 through `total`. `track` is 0 for A, 1 for B. `thePort` is the destination for the frame. `theBox` is the destination rectangle. `privateData` is the corresponding field from the parameter record.

Video Filters

A video filter is passed one 32-bit deep GWorld containing the source image and a second GWorld for the storage of the result. The total duration of the filter and the current frame number are used to calculate what the current frame should look like. The filter processes the source image into the destination image and returns. Premiere handles all of the overhead for retrieving the frames and storing them.

Video Filters are stored in files of type 'VFLT'. They contain several resources which Premiere looks for to identify the file as a video filter

'FXvs' #1000

This resource is one word in length and contains the version of the filter. The current version is 1.

'VFLT' #1000

This is the actual code module. This module is loaded in and the first byte of code is called using the following convention:

```
typedef pascal short (*FilterProcPtr)(short selector, FilterHandle theData);
```

The selector parameter can be one of the following:

```
enum {
    fsExecute = 0,
    fsSetup
};
```

`esExecute` indicates that the code should process the input parameters and generate an output frame. The `specsHandle` parameter contains all of the relevant information about how the source frame should be processed to generate the destination frame.

`esSetup` should display any dialog box which the filter wishes to use to prompt the user for additional parameters. This call should use any data contained in the `specsHandle` parameter to set up the dialog defaults, and it should place any data pertaining to the user selected settings into `specsHandle`. `specsHandle` will be nil initially. It is up to the plug-in to create the properly sized handle and place it into the parameter record.

The parameter record for a video filter is:

```
typedef struct {
    Handle          specsHandle;
    GWorldPtr       source;
    GWorldPtr       destination;
    long            part;
    long            total;
    char            previewing;
    Handle          privateData;
    VFilterCallBackProcPtr callBack;
} VideoRecord, **VideoHandle;
```

The fields are used as follows:

specsHandle: The filter defined data handle which contains all of the current settings for this filter. The filter normally creates and modifies this handle during an `esSetup` call. Premiere saves this handle in project files so that the settings will be restored when a project is re-opened.

source: Points to the source GWorld. This GWorld contains all of the source image and will always be 32 bits deep.

destination: Points to the destination GWorld. This is where the resultant frame should be stored. It will always be

32 bits deep and the same size as the source GWorld. When processing the source data into the destination, the alpha channel in the source images may contain useful data, and should be processed just like the other 3 channels.

part: indicates the particular frame which is being processed.

total: indicates the total number of frames which are being processed. **part** divided by **total** gives the percentage of the filter which should be currently calculated. **part** will vary from 0 up to and including **total**.

previewing: set if the filter is being called to render a preview quality filter. If the filter can render a faster, less accurate representation of the filter, it should do so when this flag is set.

privateData: Passed to the callback below.

callback: This is a pointer to a callback routine which can be used to retrieve past or future frames from the source clip. This field is nil during the `esSetup` call; alternate frames may only be requested during an `esExecute` call. It's calling convention is:

```
typedef pascal short (*VFilterCallBackProcPtr)(long frame, CGrafPtr thePort,  
                                               Rect *theBox, Handle privateData);
```

The parameters are: **frame** is the desired frame, from 0 through **total**. **thePort** is the destination for the frame. **theBox** is the destination rectangle. **privateData** is the corresponding field from the parameter record.

Audio Filters

An audio filter is passed one buffer of data containing the source audio blip and a second buffer for the storage of the result. The total duration of the filter and the current sample number are used to calculate what the current audio blip should sound like. The filter processes the source data into the destination data and returns. Premiere handles all of the overhead for retrieving the sound blips and storing them.

Audio Filters are stored in files of type 'AFLT'. They contain several resources which Premiere looks for to identify the file as an audio filter

'FXvs' #1000

This resource is one word in length and contains the version of the filter. The current version is 1.

'AFLT' #1000

This is the actual code module. This module is loaded in and the first byte of code is called using the following convention:

```
typedef pascal short (*FilterProcPtr)(short selector, FilterHandle theData);
```

The selector parameter can be one of the following:

```
enum {
    fsExecute = 0,
    fsSetup
};
```

`esExecute` indicates that the code should process the input parameters and generate an output sound buffer. The `specsHandle` parameter contains all of the relevant information about how the source samples should be processed to generate the destination samples.

`esSetup` should display any dialog box which the filter wishes to use to prompt the user for additional parameters. This call should use any data contained in the `specsHandle` parameter to set up the dialog defaults, and it should place any data pertaining to the user selected settings into `specsHandle`. `specsHandle` will be nil initially. It is up to the plug-in to create the properly sized handle and place it into the parameter record.

The parameter record for an audio filter is:

```
typedef struct {
    Handle          specsHandle;
    Ptr             source;
    Ptr             destination;
    long            sampleNum;
    long            sampleCount;
    char            previewing;
    Handle          privateData;
    AFilterCallBackProcPtr callBack;
    long            totalSamples;
    short           flags;
    long            rate;
} AudioRecord, ** AudioFilter;
```

The fields are used as follows:

specsHandle The filter defined data handle which contains all of the current settings for this filter. The filter normally creates and modifies this handle during an `esSetup` call. Premiere saves this handle in project files so that the settings will be restored when a project is re-opened.

source: Points to the source audio buffer. This buffer contains `sampleCount` bytes, starting at `sampleNum`

within the clip being processed.

destination: Points to the destination buffer. This is where the resultant audio data should be stored. It is `sampleCount` bytes long.

sampleNum: indicates the starting sample number which is being presented to the filter in the source buffer.

sampleCount: indicates the number of samples of data which are in the source and destination buffers.

previewing: set if the filter is being called to render a preview quality filter. If the filter can render a faster, less accurate representation of the filter, it should do so when this flag is set.

privateData: Passed to the callback below.

callback: This is a pointer to a callback routine which can be used to retrieve past or future audio data from the source clip. This field is nil during the `esSetup` call; alternate data may only be requested during an `esExecute` call. It's calling convention is:

```
typedef pascal short (*AFilterCallBackProcPtr)(long sample, long count,  
                                              Ptr buffer, Handle  
privateData);
```

The parameters are: `sample` is the desired starting sample number, from 0 through `totalSamples`. `count` is the number of samples to retrieve. `buffer` is the destination location for the data. `privateData` is the corresponding field from the parameter record.

totalSamples: the total number of samples which exist in the current clip. This can be used to calculate data offsets from the end of the clip to be used with the callback procedure.

flags: these flags indicate specifics about the source audio. They are:

```
enum {  
    ga5kHz      =      0x0001,  
    ga11kHz     =      0x0002,  
    ga22kHz     =      0x0004,  
    ga44kHz     =      0x0008,  
  
    gaStereo    =      0x0100,  
    ga16Bit     =      0x0200  
};
```

rate: the sample rate for the audio being processed. It can range from 5000 to 64000.