Adobe

version

Adobe Premiere™ 4.2

Software Development Kit

release 2 for Windows

Revised: 13 November 1996

Adobe Premiere 4.2 Software Development Kit, release 2
Copyright © 1992–96 Adobe Systems Incorporated. All rights reserved.

The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in this document. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

Adobe, Adobe Premiere, Adobe Photoshop, Adobe Illustrator, Adobe Type Manager, ATM and PostScript are trademarks of Adobe Systems Incorporated that may be registered in certain jurisdictions. Macintosh and Apple are registered trademarks, and Mac OS is a trademark of Apple Computer, Inc. Microsoft, Windows are registered trademarks of Microsoft Corporation. All other products or name brands are trademarks of their respective holders.

Most of the material for this document was derived from work by Bryan K. "Beaker" Ressler, Randy Ubillos, Dave Wise, Nick Schlott, and Matt Foster. It was then compiled, edited, and reformatted into its current form by Brian Andrews.

Version History		
13 February 1995	Matt Foster, Nick Schlott	Version 4.0 - The first Windows version.
9 February 1996	Brian Andrews	Version 4.2 - Reformatted and updated for Premiere 4.2.
13 November 1996	Brian Andrews	Version 4.2r2 - Incremental updates and bug fixes, expanded Photoshop section.

1 Introduction

Welcome to the Adobe Premiere™ 4.2 Software Developers Toolkit (SDK) for Windows!

With this toolkit you can create software, known as plug–in modules, that expand the capabilities of Adobe Premiere. The Adobe Premiere Plug-In Toolkit is for developers who wish to write plug-in modules for use with Adobe Premiere. Premiere plug-ins are called by Premiere to perform specific functions, such as filtering a frame of video or controlling a tape deck.

This guide assumes that you are proficient in C language programming and tools. The source code files in this toolkit are written for the Microsoft Visual C++ 2.2 and 4.2 software development environments.

You should have a working knowledge of Adobe Premiere, and understand how plug–in modules work from a user's viewpoint. This guide assumes you understand Premiere and basic video editing terminology. For more information, consult the *Adobe Premiere Users Guide* and/or the *Adobe Premiere Classroom in a Book.*

Windows vs. Macintosh Plug-Ins

This document describes only the Windows version of the Premiere SDK, there is another version of the entire SDK (including this documentation) available for Macintosh developers. Adobe Premiere 4.2 is available as a Macintosh and a Windows application. All the basic plug-in module types for Premiere are available on both platforms with the exception of Zoom modules, which are only available on Macintosh. The mechanism by which plug-ins operate is quite similar. Adobe encourages developers of Premiere plug-ins to create them for both platforms.

A key difference is the Macintosh version of Premiere offers a large function library that can be used for doing interface work, such as controls, that are not available to Premiere for Windows developers. The reason for the difference is for compatibility of your plug-ins with later versions of Premiere. The 4.0 version of Premiere for Windows was written for the Win16 API, the 4.2 version is written for the Win32 API.

How to Use This Guide

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

Chapter 2 describes The Utility Library. This provides Premiere specific calls.

Chapter 3 describes the Bottlenecks, which are a set of procedures and structures to perform common operations.

Chapter 4 on Transitions, is the first chapter on specific plug-in types. Transitions take two PWorlds and processes them into a single destination PWorld, usually applying some special transition effect.

Chapter 5 describes Video Filters, which take a single PWorld and processes into a destination PWorld, usually applying a visual effect.

Chapter 6 describes Audio Filters, which take a single source buffer of audio and processes it into a destination buffer, usually applying an audio effect.

Chapter 7 describes Data Export Modules. These appear in Premiere's Export submenu and export a given clip to some other format.

Chapter 8 describes EDL Export Modules. These also appear in Premiere's Export submenu and are used to export the current project into a text edit decision list.

Chapter 9 describes Device Control Modules. These allow Premiere to control hardware devices such as tape decks or laser disc players.

Finally, Chapter 10 mentions a few other plug-In types, such as Adobe Photoshop filters, which are largely beyond the scope of this document.

Perhaps the best way to use this toolkit documentation is to read this Introduction chapter, then read the chapter specific to the type of plug-in you're writing. You should then study and understand the sample plug-ins of the type you're writing. While studying the samples, you'll find function calls to routines provided in the Adobe Premiere library, a stub library of many useful routines. When you need documentation on specific library routines, look in chapter 2, The Utility Library.

About This Guide

This programmer's guide is designed for readability on screen as well as in printed form. The page dimensions were chosen with this in mind. The Frutiger font family is used throughout the manual with Courier used for code examples.

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

What's New

This version of the Adobe Premiere Software Developers Toolkit contains the following new features:

- Supports the newest release of Adobe Premiere for Windows, version 4.2. The libraries and headers files have been updated since the last release with most 4.2 changes commented in the header files.
- Supports development using Microsoft Visual C++ version 2.2 or 4.2 on Windows 95 and Windows/NT.
- Improved documentation. We hope you'll find this new document format more readable for both on screen and printed viewing.

- An instance handle has been added to the Video and Audio records which allows Premiere to retain and return state information for a plug-in. See Video Filters in chapter 5, Audio Filters in chapter 6, and the header files for information on this.

- In general, PWorld ID's have been hidden. You'll only need to deal with PixMaps from now on. In particular, the following structures have been modified to hide their PWorld ID's. See the EffectRecord structure in chapter 4 and the VideoRecord structure in chapter 5. You'll also want to take note of the change in the PPix structure in chapter 2.

- There has been an addition to Device Control which adds the command cmdJog, which can be used in lieu of cmdJogTo.

- Premiere can load and apply Adobe Photoshop filters to video clips. Chapter 10 provides an expand discussion of this capability and the limitations of Premiere's support of the Photoshop plug-in API.

Plug-In Overview

Adobe Premiere plug-ins are separate files that are placed in Premiere's plug-Ins folder. Plug-in files contain a single-entry-point DLL of a type specific to the purpose of the plug-in. Each plug-in can have private resources in its plug-in file.

Table 1–1: Basic Plug-In Types

Type	File Prefix	Name	Description
'SPFX'	FX-	Video transition	Create a C video frame from A and B frames.
'VFlt'	FL-	Video filter	Modify ("filter") one frame of video.
'AFlt'	FL-	Audio filter	Modify ("filter") one audio "blip".
'ExpD'	EX-	Data export module	Export video or audio from a clip.
'ExpM'	EX-	EDL export module	Export construction window information.
'DevC'	X-	Device control module	Control a hardware device like a tape deck.
'8BFM'		Photoshop filter	Apply a Photoshop filter to one frame of video.

The file prefixes list are not mandatory, however Adobe recommends you use them to avoid conflicts with other system libraries. The recommended prefixes reduce the chance for this happening with non-Premiere libraries.

Adobe Premiere 4.2 for Windows requires Windows 95 or Windows/NT running on an Intel 486 or greater and Microsoft Video for Windows or Quicktime for Windows. Your plug-ins can assume the presence of these software components.

Premiere Terminology

In the descriptions of the various types of Premiere plug-in modules, there are several terms that you'll see repeatedly. Refer to the Adobe Premiere User Guide for more information.

Clip

A **Clip** in Premiere are the pieces of media (movies, graphics, sounds, etc.) that become a part of an Adobe Premiere project. From a programming standpoint, Premiere identifies clips by their **clip ID**. Premiere keeps track of a variety of information about each clip, such as its type, markers, and in- and out-points.

File

Each clip in Premiere has an associated **file**, from which the original data is drawn. With the exception of Titles, Premiere does not modify the underlying file associated with a clip. Within Premiere, files are identified by a **file ID**.

Marker

Premiere allows movies and animations to have **markers** associated with different frames in the clip. There are 10 numbered markers and up to 1000 unnumbered markers. Internally, the in-point and out-point of a clip are just special markers.

Timecode

Several Premiere structures and callbacks include a **timecode** field. For the most part, the meaning of timecode should be apparent from the context in which it is used. In general, timecode is always a long and simply refers to the frame count.

Things to Remember

When programming plug-in modules for Adobe Premiere there are a few handy points of information that will ease your development.

- Strings in the Windows version of Premiere are always C strings (that is, several byes of text followed by a null character). Cross platform developers should be aware that the Macintosh API uses a mixture of Pascal and C strings.

- Premiere plug-ins are usually loaded and called only when needed. Typically after five seconds of non-use, a plug-in's code is unloaded and the plug-in file is closed. This means that for some plug-ins (like transitions (SPFX) or video (VFlt) and audio (AFlt) filters), you can leave Premiere running, switch back to your development environment, make a change, recompile and re-link the plug-in, switch back into Premiere, and try out your changes! This is a great time-saver when developing filters and transitions.

Macintosh Conventions

The Adobe Premiere for Windows API makes use of several functions and data types that are derived from the Macintosh API. Premiere was originally developed on the Macintosh and in porting the code to the Windows platform, it was convenient to borrow these definitions from the original platform. For Macintosh programmers, this should make learning or porting code to the Premiere for Windows API a bit easier. Windows programmers will have a few structures and conventions with which to familiarize themselves. Not all of them are discussed in detail in this document, but you will find reference to them in the header and source code examples.

Some functions are declared of type Pascal. On the Macintosh these routines are called with variables on the stack according to Pascal calling conventions. The type Pascal is for source code compatibility only; on the Windows platform it is empty definition.

Building Premiere for Windows Plug-ins

Premiere for Windows plug-ins are dynamic link libraries (DLLs), so if you are using a project oriented development environment be sure to specify this option. See the sample .def files to see examples of the filter functions that need to be exported. The plug-ins may also rely on a number of functions that Premiere exports which are accessible through the PREMINFO.LIB. Make certain that you link to this library; it is found in the LIB directory. The final target file should have the name extension .prm. Finally, you must include the required resources discussed next.

Plug-in Resources

Plug-in modules reside in their own file located in Premiere’s Plug-Ins folder. When a plug-in is called to perform its function, the current file is set to the plug-in’s file, and it is free to load and use any of the resources in the plug-in file. In most cases, plug-ins are provided with a facility by which to store and retrieve whatever parameters might be associated with their function. For instance, filters can fill out a “settings data record” that gets saved by Premiere in a Premiere project file, then later that data is handed back to the filter when it is called upon to perform its function. If a plug-in has extra state information or defaults, the plug-in can store it in a preferences file.

The resources used by Premiere for Windows are descended from Macintosh plug-in resources. Because of their heritage, all resources specified by Premiere for Windows have two zero bytes before any data. Resources not required by Premiere for Windows, such as dialog records, do not have this prefix. The required resources for all Premiere plug-ins are listed below with a more complete description following.

Table 1–2: Required Resources for Premiere Plug-Ins

Type & ID	Description
TYPE 1000	The TYPE resource identifies the plug-in type to Premiere during the loading process.
TEXT 1000	The name of the resource, to be displayed in the by Premiere in a window or menu.
??vs 1000	A two-byte version number stored as a short integer. The current version is 2.

TYPE 1000

The TYPE 1000 resource is checked by Premiere at startup and identifies the plug-in type. In the earlier Plug-In Overview section, there is a TYPE identifier next to each kind of plug-in. A typical TYPE 1000 resource would look like:

```
#define AFlttype 0x41466c74L // This indicates an audio filter
#define VFlttype 0x56466c74L // This indicates an video filter
#define DevCtype 0x44657643L // This is a device control module
#define ExpMtype 0x4578704dL // This is an EDL export module
#define ExpDtype 0x45787044L // This is a data export module
#define SPFXtype 0x53504658L // This indicates a transition

// The TYPE 1000 resource identifies the plug-in type to Premiere.
1000 TYPE DISCARDABLE
BEGIN
   0x0000,
   AFlttype, // one of the above #defines
END
```

Premiere for Macintosh developers will recognize this as the plug-in file type moved to a resource.

TEXT 1000

The TEXT 1000 resource contains the name of the plug-in. This name will be displayed in a window, a dialog, or a menu, depending on the plug-in type. It should be kept short but descriptive. Look at other plug-ins for example plug-in names. An example of a TEXT 1000 resource would be:

```
// The TEXT 1000 resource plug-in. It will appear in a window or menu.
1000 TEXT DISCARDABLE
BEGIN
   0x0000,
   "Fill Left\0"
END
```

??vs 1000

The version resource is used to identify the version of the Premiere for Windows API to which the plug-in conforms. The current version for all types of plug-ins is 2. Each of the general kinds of plug-ins has a slightly different resource identifier, as shown below:

- Filter modules use the resource type FLvs
- Transition modules use the resource type FXvs
- Export modules use the resource type EXvs
- Device control modules use the resource type DVvs

An example of a version resource for an FLvs:

```
// Plug-ins must have a version resource to identify the API expected.
1000 FLvs DISCARDABLE
BEGIN
   0x0000,
   0x0002
END
```

If a plug-in does not have a version resource or if the version number is for a later version of the API than the running Premiere supports, it will not load.

Using the FPU from Within a Plug-in

An important consideration for anyone writing drivers or DLL's that are called from any Windows application is working with floating point code. If your DLL uses any floating point code, including floating point code contained in run-time libraries, your plug-in or driver must be very careful to leave the FPU in a known state on exit.

The best way to do this is if your code is not math-intensive is to use the alternate math libraries via the /FPa compiler switch and link to the LLibCaw.lib library. If you must use the FPU for performance reasons call _FPInit() and _FPTerm at every DLL entry and exit point.

For more information, please refer to the article titled "Floating-Point in Microsoft Windows", 8/10/92 by David Long, available on the Microsoft Developer Network CD. In particular, read the section titled "DLL Considerations."

2

The Utility Library

Many Adobe Premiere plug-ins perform similar functions. To reduce code size and leverage existing code, Premiere has an extensive utility library that provides both general Macintosh utility routines as well as Premiere-specific calls. To use this library, you link your plug-in with a stub library, PREMINFO.LIB. The actual code for the utility routines is dynamically linked at run-time by Premiere when you call the library routines.

The documentation for these routines is divided into two major categories: general Macintosh related routines and Premiere-specific routines. Within those categories, the routines are divided into functional categories.

Macintosh Memory Management

The Macintosh uses a memory management scheme different from Windows. The original scheme used the concept of a Handle, a relocatable block of memory.[1] A master list of Handles was kept by the operating system. A program kept a pointer to a pointer in the master list. Thus Handles are referenced by double indirection (e.g. **myHandle). A block of memory allocated as a Handle may move to keep available heap space contiguous. There are a variety of instances when this will occur, but for the Premiere for Windows implementation it will occur only when additional memory is allocated.

Because Handles move, they can be marked as locked to keep them from moving. To avoid memory fragmentation, a locked Handle is unlocked as soon as possible. Some Handles, such as those containing user interface elements, are automatically removed from memory when they are unused and their memory is need for something else. To insure that a Handle is not removed from memory. It can be marked as unpurgeable. When the memory is no longer required, the block is marked as purgeable. For more information about these calls and the Macintosh memory manager, you should refer to the Inside Macintosh Series published in several forms by Addison-Wesley.

To aid in porting the original code base, parts of the original Macintosh memory manager were duplicated in Premiere for Windows. These routines are exported and in addition to being necessary in some cases, they should be of use in making Premiere for Windows and Macintosh plug-in code more compatible across platforms. Your plug-in is not required to use them for its own memory management. It must use them when passing a block of memory to Premiere, such as when using a specsHandle to store parameter information. It should be noted that for this revision of the Premiere API, the purgeable state of a Handle are largely ignored.

[1] Don’t confuse the term handle. A Macintosh memory Handle is not the same as a Windows HANDLE. This SDK will use the case sensitive spellings to distinguish between the two.

The memory management functions use the following type declarations:

```
typedef char *Ptr;
typedef Ptr *Handle;  /* pointer to a master pointer */
typedef long Size;    /* size of a block in bytes */
```

MemError:

Returns the error code caused by the last memory manager function used.

```
pascal OSErr MemError (void);
```

This routine is used after a memory manager function is called. These routines often return a pointer or Handle directly, so to find the whether and error occurred and the error code, you would use this call. The routine returns an variable of type OSErr, which is a long.

```
HPurge( myHandle );
if ( MemError() ) {
   // process the error
}
```

The error codes that can be returned are:

```
#define noErr 0
#define memFullErr -108 // if not enough room for
                           allocation
#define memWZErr -111   // if a nil handle is passed
                        // where an existing handle
                           is expected
```

BlockMove:

BlockMove copies memory from one place to another.

```
pascal void BlockMove (
    const void *srcPtr,
    void *destPtr,
    Size byteCount );
```

This routine moves a block of memory. The srcPtr points to the start of the memory to be moved; the destPtr points to where the memory should be moved. The bytecount is a long indicating the number of bytes that should be moved. The instance where the source and destination overlap is handled correctly. This call is provided for code compatibility with the Macintosh. You may also use any of the memory moving routines provided by the Windows API.

NewPtr:

Allocated a block of memory for direct reference.

```
pascal Ptr NewPtr (Size byteCount);
```

This routine allocates a block of memory of bytecount bytes at the lowest possible point on the application heap. It is analogous to the malloc() standard C library call. The pointer returned by the call points directly to the allocated memory. If the call fails, nil will be returned. See also DisposPtr(), NewPtrClear(), and SetPtrSize().

An example of the use of this and related calls follows:

```
Ptr myPtr;

myPtr = NewPtr(1024);
.
.
.
SetPtrSize(myPtr, 2048);
if (MemError()) {
   // whoops! better do something...
}
.
.
.
DisposePtr(myPtr);
```

DisposPtr: Free a block of memory.

```
pascal void DisposPtr (Ptr p);
```

This routine disposes memory allocated by the NewPtr() routine and reference by the pointer p. Do not call the routine with multiple copies of the same pointer. This routine may also be referred to as DisposePtr().

NewPtrClear: Allocate a cleared block of memory for direct reference.

```
pascal Ptr NewPtrClear (Size byteCount);
```

This routine is the same as NewPtr() except that it clears the block of memory to 0 before returning it.

SetPtrSize: Shrink or expand a memory block reference by a pointer.

```
pascal void SetPtrSize (
    Ptr FAR *p,
    Size newSize );
```

This routine is used to allocate additional memory to or free existing memory from a block created by NewPtr() and reference by p. The new size of the block is newSize. Reducing the size of a block of memory will always succeed. Increasing the size of a block of memory may fail if there is insufficient contiguous free memory. Using this call will cause the value of the pointer to change.

GetPtrSize: Obtain the size of a block of memory in bytes.

```
pascal Size GetPtrSize (Ptr p);
```

This routine returns the size of a block of memory referenced by the pointer p.

NewHandle: Allocate a block of memory from the application heap.

```
pascal Handle NewHandle (Size byteCount);
```

NewHandle() is used to create a block of indirectly referenced memory of size byteCount. It returns a pointer to a pointer to the allocated block of memory. If the call fails, nil will be returned.

See also DisposHandle(), NewHandleClear(), and SetHandleSize().

An example of the use of this and related calls follows:

```
Handle myHandle;

myHandle = NewHandle(1024);
.
.
.
SetHandleSize(myHandle, 2048);
if (MemError()) {
   // whoops! better do something...
}
.
.
.
DisposeHandle(myHandle);
```

DisposHandle:

Free a block of memory allocated using NewHandle().

```
pascal void DisposHandle (Handle h);
```

This routine is used to free a block of memory referred to by the Handle h. It may also be called as DisposeHandle(). After the call, the Handle variable h still refers to the master pointer list, though the pointer to which it refers is invalid and must not be used.

NewHandleClear:

Allocate and clear a block of memory from the application heap.

```
pascal Handle NewHandleClear (Size byteCount);
```

This routine is the same as NewHandle() except that it clears the allocated block of memory before it is returned.

This routine is used to allocate additional memory to or free existing memory from a block created by NewPtr() and reference by p. The new size of the block is newSize. Reducing the size of a block of memory will always succeed. Increasing the size of a block of memory may fail if there is insufficient contiguous free memory. Using this call will cause the value of the pointer to change.

GetHandleSize:

Obtain the size of an allocated block of memory.

```
pascal Size GetHandleSize (Handle h);
```

This routine returns the size of a block of memory referenced by the Handle h.

SetHandleSize:

Given a ID of a PWorld, return the associated data.

```
pascal void SetHandleSize (
    Handle h,
    Size newSize );
```

This routine is used to allocate additional to or free existing memory from a block created by NewHandle() and reference by h. The new size of the block is newSize. Reducing the size of a block of memory will always succeed. Increasing the size of a block of memory may fail if there is insufficient free memory.

HLock:

Set the locked state of a Handle to true.

```
pascal void HLock (Handle h);
```

When a Handle is locked on the Macintosh , it will not be moved in the application heap and the data can be reference directly. To lock memory referenced by a Handle h, use the call HLock(h). This call is included for code compatibility with the Macintosh.

HUnlock:

Set the locked state of a Handle to false.

```
pascal void HUnlock (Handle h);
```

To avoid heap fragmentation on the Macintosh, the locked Handle h should be unlocked as soon as possible. This call is included for code compatibility with the Macintosh.

HNoPurge:

Mark a block of memory reference by a Handle as unpurgeable.

```
pascal void HNoPurge (Handle h);
```

When a Handle is marked as unpurgeable on the Macintosh, it will not be removed from the application heap to free up memory for some other use. To make sure that the memory of a Handle h is not freed for some other use, call HNoPurge(h) This call is included for code compatibility with the Macintosh, since the current memory manager ignores the purgeable setting.

HPurge:

Mark a block of memory reference by a Handle as purgeable.

```
pascal void HPurge (Handle h);
```

On the Macintosh, after the data in a Handle is no longer required, an unpurgeable Handle h should be marked purgeable as soon as possible. This call is included for code compatibility with the Macintosh, since the current memory manager ignores the purgeable setting.

HGetState:

Get the locked and purgeable state of memory reference by a Handle.

```
pascal char HGetState (Handle h);
```

This routine is included for compatibility purposes. It returns the byte storing the locked and purgeable state of the Handle h. This should be used for information only. Rather than directly manipulating this byte, you should use the calls HPurge(), HNoPurge(), HUnlock(), and HLock().

HSetState:

Set the locked and purgeable state of memory reference by a Handle.

```
pascal void HSetState (
    Handle h,
    char flags );
```

This routine is included for compatibility purposes. It directly sets the char flags by storing the locked and purgeable state of the Handle h. Rather than using this call, you should use the calls HPurge(), HNoPurge(), HUnlock(), and HLock().

MoveHHi:

Given a ID of a PWorld, return the associated data.

```
pascal void MoveHHi (Handle h);
```

Before locking down a Handle on the Macintosh, the memory should be moved to the top of the application heap to avoid heap fragmentation. Use MoveHHi() on the Handle h you want to lock before calling HLock(). This routine is included for compatibility reasons. It doesn’t do anything in the current version of the API.

HandToHand:

Create a Handle and copy an existing Handle to it.

```
pascal OSErr HandToHand (Handle *theHndl);
```

HandToHand() creates a new Handle the same size as the one passed to it. The routine then copies the data of the original Handle theHndl to the new one. The new Handle is returned in place of the original. Because of this, you should make a copy of the original Handle before calling this function. This call returns the same error code as a call to MemError().

The HandToHand() routine could be used as follows:

```
Handle srcHandle, destHandle;

// assume srcHandle is initialized with some data
destHandle = srcHandle;
if ( HandToHand(&destHandle) ) {
   // Something went wrong...
}
```

PtrToHand:

Copy the data from a pointer in to a Handle.

```
pascal OSErr PtrToHand (
    const void *srcPtr,
    Handle *dstHndl,
    long size );
```

PtrToHand() creates a Handle dstHndl of size bytes and copies the memory pointed to by srcPtr to the new block. The destination Handle should not be created before passing it to this function. This call will fail if their is insufficient memory to create the new Handle and this call returns the same error code as a call to MemError().

HandAndHand:

Append the data referred to by one Handle to another.

```
pascal OSErr HandAndHand (
    Handle hand1,
    Handle hand2 );
```

Use HandAndHand() to append the memory referenced by hand1 to the Handle hand2. The destination Handle will be automatically sized to hold the new data. This call will fail if there is insufficient memory to increase the memory block reference by the Handle and this call returns the same error code as a call to MemError().

PtrAndHand:

Append a block of memory referred to by a pointer to a Handle.

```
pascal OSErr PtrAndHand (
    const void *ptr1,
    Handle hand2,
    long size );
```

Use PtrAndHand() to append size bytes of data from the memory pointed to by ptr1 to the Handle hand2. The Handle will be automatically sized to hold the new data. This call will fail if there is insufficient memory to increase the memory block reference by the Handle and this call returns the same error code as a call to MemError().

Graphics

About PWorlds

The Macintosh API provides a convenient graphics structure for working with off screen graphics called a GWorld. Associated with a GWorld is a PixMap, which contains information about the graphics world like bounds and depth. These structures and routines to use them have been partially recreated for the Premiere for Windows API; they are call PWorlds. All PWorlds have an ID associated with them.

Premiere 4.2 now hides all the PWorlds. The structures which used to contain PWorlds (EffectRecord and VideoRecord) now return a PPixHand, alleviating the need to call GetPWorldBits(), still documented below.

A PixMap is defined as follows. Note that this structure has changed in Premiere 4.2:

```
// video frame record, 32 bits/pixel
// All internal buffers are in this format
// Pixel buffer pointed to by *pix is the same as a DIB buffer:
// BGRA, origin at lower left.

typedef struct // note changes to Premiere 4.2
{
   RECT bounds;     // bounds of pixmap
   int rowbytes;    // rowbytes of data
   int bitsperpixel // bits per pixel - in 4.2 or earlier, always 32
   int pixelformat  // storage format for future expansion - in 4.2 or
                    // earlier, always 0 (RGB)
   char *pix        // pointer to pixel buffer
   long reserved[4]
} PPix, *PPixPtr, **PPixHand;

typedef short PWorldID;
```

Note that all pixels are 32 bits deep, with one byte each for blue, green, red, and the alpha channel in that order. This is the same format as the windows screen map 32-bit DIB as documented in the Video for Windows SDK extensions to the DIB format.

The rowbytes variable contains the number of bytes in each row, and the number of pixels in a row would be rowbytes/4. The maximum number of pixels in a row is 2000.

GetPWorldBits:

Given a ID of a PWorld, return the associated data.

```
PPixHand GetPWorldBits (PWorldID pw);
```

This routine is used to obtain the pixel data associated with a PWorldID pw. When the transition and video effect plug-ins, for instance, are called, the associated data structure has PWorldIDs for the input and output buffers. One of the first things a plug-in would do would be call GetPWorldBits() for any buffer. For instance, the following code is from the video noise filter:

```
case fsExecute:
// Establish pointers to the two pixel buffers. Our
// noise routine will copy from srcPix to destPix while
// affecting the pixels.
srcPix = *GetPWorldBits((*theData)->source);
destPix = *GetPWorldBits((*theData)->destination);
.
.
.
```

PPixToScreen:

Draw a PixMap to the screen.

```
void PPixToScreen (
    PPixHand pix,
    HDC dc,
    LPRECT drawrect,
    LPRECT theRect );
```

Use PPixToScreen() to move all or part of a PixMap to a window, perhaps to preview an effect. The PPixHand pix is the source, containing the pixels you want to move to the screen. You are responsible for creating the device control reference, dc. The drawrect is the rectangle defining the part of the PWorld from which you wish to draw. The other rectangle, theRect, is the screen destination relative to the dc. You can also use the DrawDIBDraw() function documented in the Video for Windows Toolkit to move the pixels to the screen, though using the PPixToScreen() call will avoid the ugly effects of color palette switching.

NewPWorld:

Create a new PWorld.

```
char NewPWorld (
    PWorldID *pwID,
    LPRECT bounds );
```

NewPWorld() creates a PWorld with the dimensions you define in the rectangle bounds. Enough memory is allocated for all the pixels plus 16 bytes for the PPix information. If there is insufficient memory to complete the call, and a non zero error code is return and the returned PWorldID will be nil. Note that the routine returns a PWorldID, pwID, so you would need to call GetPWorldBits() to get a handle to the pixels.

```
err = NewPWorld(&thePort, &box);
if(!err)
{
   pix = GetPWorldBits(thePort);
   ...
   if (thePort)
      DisposePWorld(thePort);

   result = 0;
}
```

DisposePWorld:

Free the memory allocated for a PWorld.

```
void DisposePWorld (PWorldID pw);
```

This routine is the complement of NewPWorld(). After you are finished using the PWorld pw you created, you should call DisposePWorld() with the PWorldID pw. See the example in NewPWorld().

Data Export Module Utilities

The routine in this section is primarly for use by data export module (ExpD, Ex-) plug-ins.

GetExportFilePath:

Return the file path of a clip.

```
void GetExportFilePath (
    DataExportHandle datahand,
    LPSTR path);
```

Given a DataExportHandle (which is returned as thedata in a data export module call back), return the full file path of the clip. This is useful if you wish to play the clip using your own player.

EDL Export Module Utilities

The routines in this category are meant for use by EDL export (‘ExpM’, ‘EX-’) plug-ins. The block routines pertain to the project data block list that is provided to EDL export modules. For more information on the format of this data, see the chapter EDL Export Modules.

CountTypeBlocks:

Return the number of blocks of a given type.

```
long CountTypeBlocks (
    long type,
    BlockRec *srcBlock );
```

This routine returns the number of blocks of type type starting from srcBlock. If type is -1, CountTypeBlocks() returns the total number of blocks after srcBlock.

FindBlock:

Find a particular block.

```
BlockRec *FindBlock (
    long type,
    long theID,
    long index,
    BlockRec *srcBlock );
```

This routine can be used to locate a specific block by index, type, or ID. The routine starts searching from srcBlock. The type parameter specifies the block type to look for, where -1 means "any type." The theID parameter allows you to specify a specific block ID of the given type. The index parameter allows you to specify a block index. FindBlock() returns a BlockRec pointer. If the specified block could not be found, FindBlock()returns nil.

Table 2–1: FindBlock Parameter Combinations

Type	theID	Index	Action
-1	-1	valid	Find the block index blocks from srcBlock.
valid	-1	valid	Find the index-th block of type type from srcBlock.
valid	valid	-1	Find the block of type type and ID theID.

GetBlock:

Return a Handle to a copy of a particular block's data.

```
BlockRec **GetBlock (
    long type,
    long theID,
    long index,
    BlockRec **srcBlock );
```

This routine calls FindBlock using the type, theID, and index parameters starting from *srcBlock. See FindBlock(), above, for details about how those parameters are used. If the specified block is found, GetBlock() calls PtrToHand() to copy the block's data into a Handle and returns the Handle. If the block is not found, GetBlock() returns nil. Note that *srcBlock is never changed (it is essentially a const parameter).

ExtractBlockData:

Copy a block's data to a destination buffer.

```
void ExtractBlockData (
    BlockRec *srcBlock,
    void *destination,
    long *maxlen );
```

This routine copies the data for srcBlock (not including the BlockRec) to the buffer destination up to a maximum of *maxlen bytes. The actual number of bytes copied is returned via the reference parameter maxlen.

Miscellaneous Routines

GetMainWnd:

Get a HANDLE to Premiere's main window.

`HWND GetMainWnd (void);`

This routine returns a HANDLE to Premiere's main window. Your plug-in will need this when displaying a parameters dialog to set the proper parent window.

Use GetLastActivePopup(GetMainWnd()) for any plug-in dialog parents.

```
case esSetup:
   myData = NULL;
   result = DialogBoxParam (
               resInst,
               MAKEINTRESOURCE(1000),
               GetLastActivePopup(GetMainWnd()),
               (DLGPROC)AskDlogProc,
               (LPARAM)theData );
   break;
```

3 Bottlenecks

Adobe Premiere provides a set of bottleneck procedures to its plug-in modules to perform common operations. This chapter describes the BottleRec structure that contains the bottleneck function pointers and describes each bottleneck.

The BottleRec Structure

Bottlenecks are passed to plug-ins through a structure called a BottleRec.

```
typedef struct {
   short                    count;           // Number of routines
   short                    reserved[14];

   StretchBitsPtr           StretchBits;
   DistortPolygonPtr        DistortPolygon;
   PolyToPolyPtr            MapPolygon;
   AudStretchPtr            AudioStretch;
   AudMixPtr                AudioMix;
   AudSumPtr                AudioSum;
   AudLimitPtr              AudioLimit;
   DistortFixedPolygonPtr   DistortFixed;
   FixedToFixedPtr          FixedToFixed;
   ImageKeyPtr              ImageKey;
   long                     unused[3];
} BottleRec;
```

The count field specifies how many bottleneck routines follow the reserved field. As of Adobe Premiere 4.2, count is 10. The reserved and unused fields are reserved for future use by Adobe Systems and are currently 0.

Most Premiere plug-ins have a standard record associated with the plug-in type. The transition, video filter, and audio filter records provide a pointer to the bottleneck record in their records. With that pointer your plug-in can call the bottleneck routines, as follows:

```
((*theData)->bottleNecks->StretchBits)(*srcpix, *dstpix, &srcbox, &srcbox,
                                       0, NULL);
```

The Bottleneck Routines

StretchBits: The StretchBits() routine is based on a Macintosh API routine call CopyBits(). It can be used to copy an image in its original aspect ratio, scaled horizontally or vertically, or clipped by a region. It properly processes the alpha channel during the copy. When the destination is larger than the source, it can perform bilinear interpolation to generate the destination. This feature provides smoothed enlargement of source material.

```
void StretchBits (
   PPixPtr srcPix,
   PPixPtr dstPix,
   LPRECT srcRect,
   LPRECT dstRect,
   short mode,
   HANDLE rgn );
```

StretchBits() only works on 32-bit deep PixMaps, the srcPix and dstPix, and not on any other bitmaps structures. The srcRect defines the area of the source PixMap from which to copy. The dstRect not only defines the area where the pixels will be copied, but is used to indicate the scale factor of the copy.

The mode argument determines the manner in which the source is copied to the destination. The valid modes are:

```
// modes for StretchBits bottleneck
#define cbBlend   0x8000
#define cbInterp  0x4000
#define cbMaskHdl 0x2000
```

cbBlend defines a blend between the source and destination bitmaps. When this mode is used, the low byte of the mode defines the amount of blend between the source and destination in a range of 0-255, e.g. to blend 30% of the source pixmap with the destination pixmap, you would use cbBlend | (30*255/100).

cbInterp mode does bilinear interpolation when resizing a source pixmap to a larger destination, resulting in a much smoother image. The trade-off is that this mode is much slower than the normal mode, while the normal copy mode does pixel replication when resizing upwards, resulting in a blockier image.

cbMaskHdl tells StretchBits that the HRGN passed in is actually a GlobalMemory handle pointing to a 1-bit deep buffer the size of the source and destination pixmaps that defines the mask to use when copying. StretchBits currently limits masking operations to working with a mask and source and destination pixmaps that are all the same size; masks cannot be used when copying pixels from a source to a destination rectangle or pixmap that are different sizes.

Use the Windows API to define the region rgn you want to use for clipping. If no clipping is desired, pass nil for the HANDLE rgn. Programmers familiar with the Premiere for Macintosh API should note that while that platform’s StretchBits() call passed off region masking to the CopyBits() call, the Premiere for Windows StretchBits() routine handles the region masking itself. This means that while the alpha channel is lost using the Macintosh StretchBits() function, it is preserved using the Windows StretchBits() call.

DistortPolygon:

The DistortPolygon() routine takes a rectangle from a source PixMap and maps the enclosed image to a four-point polygon in a destination PixMap.

```
void DistortPolygon (
    PPixHand src,
    PPixHand dest,
    RECT *srcbox,
    Point *dstpts );
```

The srcBox parameter specifies a rectangular area within the src PixMap. The dstpts parameter should be set to point to an array of four Points which describe a four-point polygon in the destination PixMap. A Point is a Macintosh API type:

```
struct Point {
   int x;
   int y;
};
```

DistortPolygon() will distort the pixels within srcbox into the specified polygon in dest. When scaling up, DistortPolygon() uses bilinear interpolation. When scaling down, it uses pixel averaging. All 32-bits of the source (that is, RGB plus the alpha channel) are transferred to the destination.

MapPolygon:

The MapPolygon() routine takes a four-point polygon in a source PixMap src and maps it into a four-point polygon (dstpts) in a destination PixMap dest. Here's the prototype for MapPolygon():

```
void MapPolygon (
    PPixHand src,
    PPixHand dest,
    Point *srcpts,
    Point *dstpts );
```

MapPolygon() is just like DistortPolygon() except that its source is specified as a four-point polygon (srcpts) in src instead of a rectangle. It also performs pixel averaging and bilinear interpolation as appropriate, and moves all 32 bits of the source to the destination.

AudioStretch:

The AudioStretch() routine performs sample rate, format (8- or 16- bit), and mono/stereo conversions between two buffers of audio. The prototype for AudioStretch() is as follows:

```
void AudioStretch (
    Ptr src,
    long srclen,
    Ptr dest,
    long destlen,
    UINT flags );
```

The src parameter is a pointer to a buffer full of audio samples, and srclen is its length in bytes. The dest parameter is a pointer to a buffer for the resampled audio, and destlen is its length. The flags parameter provides information about both buffers of audio. The high byte contains the flags for the source, the low byte contains the flags for the destination. These bits are of interest:

```
#define gaStereo 0x01
#define ga16Bit 0x02
```

For example, to go from an 8-bit stereo source to 16-bit stereo destination, set flags to (gaStereo << 8) + (ga16Bit + gaStereo).

Eight-bit audio should be in offset format, and 16-bit audio should be in signed short format. AudioStretch() stretches (or squashes) the audio in src to fit in dest, performing any format conversions according to the flags.

AudioMix: The AudioMix routine is a vestige of Premiere and is no longer supported. Use the more powerful AudioSum routine described next.

AudioSum: The AudioSum routine sums a buffer of audio into a longword accumulation buffer, providing a mix level.

```
pascal void AudioSum (
    Ptr src,
    Ptr dest,
    long width,
    long scale,
    UINT flags,
    long part,
    long total );
```

The src parameter is the source buffer that is being summed, which is regular 8- or 16-bit audio, either mono or stereo. The size in samples of src is given in width. The dest parameter points to the accumulation buffer. It is an array of longs, and must be four times the size of src. The scale parameter takes a value in 16.16 fixed-point format with a maximum value of 0x00020000, or 2.0. The audio flags are the same as for AudioMix: use the values gaStereo and ga16Bit to describe the audio in the src buffer. Part is the buffer number you're mixing, which varies from 0 to total - 1. Total is the total number of buffers you're mixing into the accumulation buffer. Note that since AudioSum adds src to dest, dest must be set to all zeros before the first call to AudioSum.

A little information on fixed-point math. Fixed-point math represents a floating point number in 4 bytes. The high byte is the integer portion (it is signed), and the low byte is the fractional portion. Fixed-point numbers can be added and subtract with normal operators, but multiplication and division will not work without special routines. The numbers defined below can be added together to get ratios to use in the fixed-point scale parameter. For instance, to use a scale factor of 0.6, pass kFixedHalf+kFixedTenth for the argument.

```
#define  FixedOne           x00010000L
#define  FixedZero          x00000000L
#define  FixedHundredth     x0000028FL
#define  FixedSixteenth     x00001000L
#define  FixedTenth         x00001999L
#define  FixedEighth        x00002000L
#define  FixedQuarter       x00004000L
#define  FixedHalf          x00008000L
#define  FixedSevenEighths  x0000E000L
#define  FixedOne1          x0000FFFFL
#define  FixedOneThird      x00005555L
#define  FixedFourThirds    x00015555L
#define  FixedThreeHalves   x00018000L
```

AudioLimit:

The AudioLimit routine clips the source audio buffer while copying to a destination buffer. The prototype for AudioLimit is as follows:

```
void AudioLimit (
    Ptr src,
    Ptr dest,
    long width,
    UINT flags,
    long total );
```

The src parameter is a longword accumulation buffer (usually one you've been accumulating into with AudioSum). The width parameter gives the size of the output buffer in samples. The dest parameter is the output buffer, which will contain regular 8- or 16-bit audio. The flags parameter describes the format of the audio that should be placed in dest. The total parameter is the total number of buffers that were mixed (with AudioSum) to get src.

DistortFixed:

The DistortFixed routine is analogous to DistortPolygon but maps the given rectangular area to a four-point polygon specified in fixed-point coordinates.

```
void DistortFixed (
    PPixHand src,
    PPixHand dest,
    Rect *srcbox,
    LongPoint *dstpts );
```

DistortFixed is just like DistortPolygon except that the destination polygon is specified with LongPoints, which have their h and v coordinates as 16.16 fixed-point values.

FixedToFixed:

The FixedToFixed routine is analogous to MapPolygon but maps a four-point polygon specified in fixed-point coordinates to another fixed-point polygon.

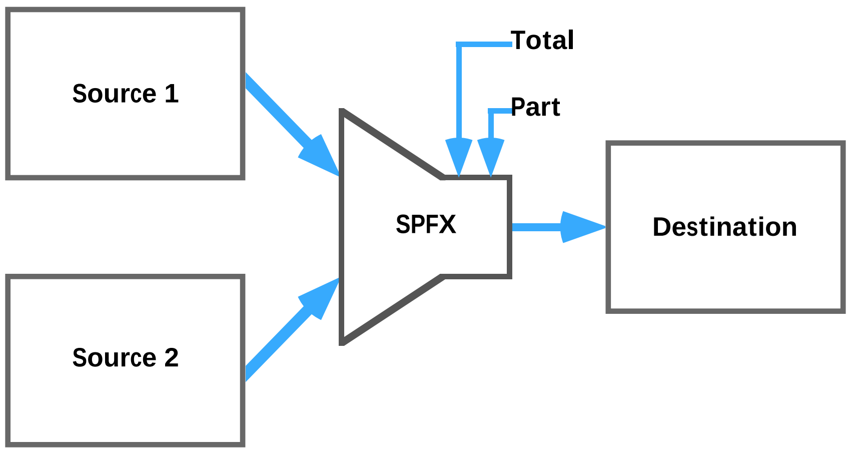
```
void FixedToFixed (
    PPixHand src,
    PPixHand dest,
    LongPoint *srcpts,
    LongPoint *dstpts );
```

FixedToFixed is just like MapPolygon except that both the source and destination polygons are specified in with LongPoints, which have their h and v coordinates as 16.16 fixed-point values.

ImageKey:	ImageKey is a private bottleneck and should not be called by plug-in modules.

4 Transitions

A transition in Adobe Premiere takes two source PixMaps and processes them into a single destination PixMap.



All three PixMaps are always 32-bits deep. Transitions are almost always time-variant, and the transition is given the total duration in frames of the transition and the current frame within that transition. Premiere handles the overhead involved with retrieving and storing the frames.

The Transitions window in Premiere shows a 9-frame animated preview of each transition. Premiere automatically generates these preview frames by calling your SPFX module and then storing the frames in the Premiere preferences file. If your plug-in changes, Premiere will regenerate and store the new transition frames.

The file name has an prefix of 'FX-' and an extension '.prm'. A transition file will contain several resources, which are listed in the table below. Following the table is a detailed description of each resource.

Table 4–1: Transition Resources

Type & ID	Description
FXvs 1000	A two-byte version number stored as a short integer. The current version is 2.
TYPE 1000	The TYPE resource identifies the plug-in type to Premiere during the loading process.
TEXT 1000	The name of the resource, to be displayed in the Transitions window.
TEXT 1001	A textual description of what the transition does.
Fopt 1000	A resource that describes what options the transition supports.
FXDF -1	A resource used for mapping the transition to one of the standard SMPTE wipes.

FXvs 1000

This resource is two bytes in length and gives the version of the module interface. The current version is $0002. See the Plug-In Resources section of the Introduction for an example.

TYPE 1000

The TYPE 1000 resource is checked by Premiere at startup and identifies the plug-in type. Transition modules have a TYPE of ‘SPFX’. Again, refer to the Plug-In Resources section of the Introduction.

TEXT 1000

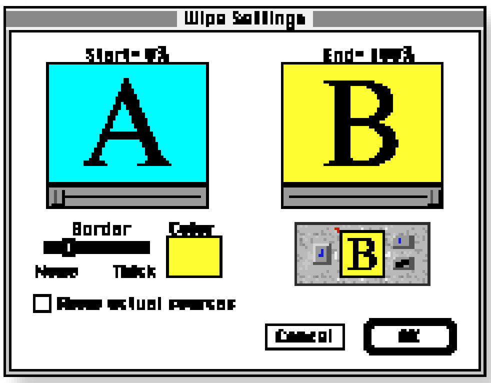
This resource contains the title of the transition resource. It appears in the Transitions window to identify a transition by name. Look at Premiere’s standard transitions for examples of how this text should be worded.

TEXT 1001

This resource contains a plain text description of the transition and is displayed beneath the title of the transition in the Transitions window. Look at Premiere’s standard transitions for examples of how this text should be worded.

Fopt 1000

This resource tells Premiere what options your transition supports and provides a set of initial values for your options. The user can bring up the standard Premiere transition options dialog for all transitions, including yours.



The structure of an Fopt resource is shown below, along with some bit values you will use with the definition:

```
'Fopt'
{
   0x0000,                          // two zero bytes at the start
   byte;                            // Byte 3, valid corners mask
   byte;                            // Byte 4, initial corners
   byte;                            // Byte 5, has custom, in pairs,
                                       and time invariant
   byte    // No = 0, Yes = 1;      // Byte 6, exclusive?
   byte    // No = 0, Yes = 1;      // Byte 7, reversible?
   byte    // No = 0, Yes = 1;      // Byte 8, has edges?
   byte    // No = 0, Yes = 1;      // Byte 9, has start point?
   byte    // No = 0, Yes = 1;      // Byte 10, has end point?
};

// Effect Corner Bits for byte 4
enum {
   bitTop         = 0x01,
   bitRight       = 0x02,
   bitBottom      = 0x04,
   bitLeft        = 0x08,
   bitUpperRight  = 0x10,
   bitLowerRight  = 0x20,
   bitLowerLeft   = 0x40,
   bitUpperLeft   = 0x80
};

// Values for byte five
#define bitPairs      0x01          // Opposite corners turn on together
#define bitCustom     0x02          // This SPFX has a custom settings dialog
#define bitInvariant  0x04          // This SPFX does not vary over time
#define bitNo1stCall  0x08          // Don't do an initial esSetup call
#define bitUsesSource 0x20          // esSetup uses callback
```

Fopt—third byte: Valid corners

The first byte of the Fopt resource uses the corner bits listed above. It has a bit set for each valid corner. For instance, if your transition can only operate top-to-bottom or bottom-to-top, you'd set the first byte of this resource to bitTop+bitBottom.

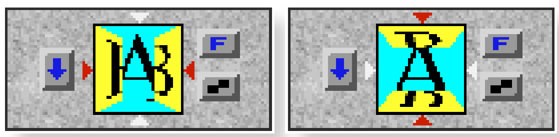
Fopt—fourth byte: Initial corners

The second byte of the Fopt resource is the initial corner settings for your transition. Choose an appropriate default. Be sure to only specify corners that are allowed according to the first byte of the Fopt.

Fopt—fifth byte: Bit flags

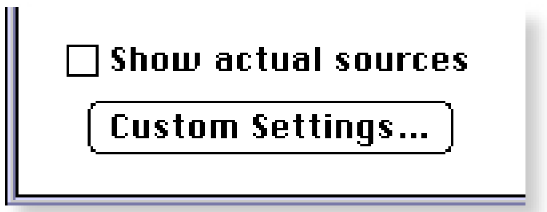
The third byte of the Fopt has four bit flags.

Bit 0: Bit zero should be set (using the bitPairs constant) if opposite corners are always to be highlighted simultaneously. The figure below shows the Doors transition, which uses this flag. Clicking the top arrow automatically selects both the top and bottom arrows.



Bit 1: Bit one should be set (using the bitCustom constant) if the transition has a custom parameters dialog—that is, it has more parameters than the standard set given in the Premiere transition options dialog. Setting this bit has two effects. First, Premiere knows your transition has custom parameters, so when the user drags your transition into the Construction window, Premiere automatically calls your transition with an esSetup message at that time. See the description of bit four if you wish to default

your custom parameters instead of getting the initial esSetup call. The second effect is that an extra button will appear on the bottom of the transition options dialog:



When the user clicks this button, your SPFX will be called with the esSetup selector. See the section below on the SPFX resource for details.

Bit 2: Bit two should be set (using the bitInvariant constant) if your transition is time-invariant—that is, it is really a two-input filter, rather than a transition. An example would be the Displace transition in Premiere, which displaces pixels in source two based on the channel values in source one. For normal transitions, this bit is set to zero.

Bit 3: Bit three should be set (using the bitNo1stCall constant) only if you use bitCustom and don't want the initial esSetup call when your transition is dragged into the Construction window. If there's a reasonable set of default values you can use for your custom parameters, set this bit along with bitCustom so the user is not interrupted with your custom settings dialog. Premiere's Pinwheel transition is a good example of a transition that uses this flag. It has a custom parameter for the number of wedges in the pinwheel. However, it defaults this number to 8, and doesn't bother the user with a custom parameters dialog every time they drag the transition into the Construction window.

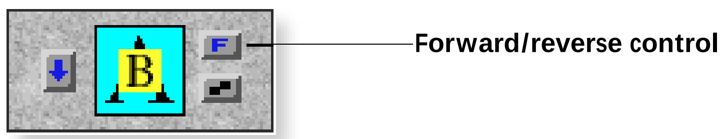
Bit 5: Bit five should be set (using the bitUsesSource constant) if your transition needs the callback function to work at setup time. For instance, a "video echo" transition that uses past video frames would get those frames by calling the callback function, and therefore should set this bit to one. The callback procedure pointer is invalid at esSetup time if this bit is not set.

Fopt—sixth byte: Exclusive flag

The fourth byte of the Fopt resource is a boolean flag that tells Premiere whether the corner arrows are to be exclusive. If this flag is set, the arrows will act like radio buttons. If this flag is clear, the arrows will act like checkboxes.

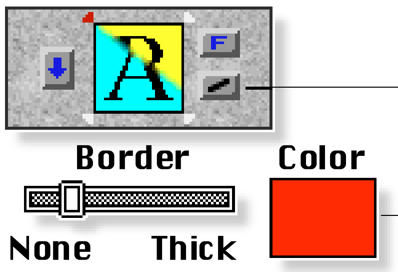
Fopt—seventh byte: Reversible

The fifth byte of the Fopt resource is a boolean flag that tells Premiere whether the transition is reversible, that is the transition can proceed either from source 1 to source 2, or vice versa. If this flag is set, the transition direction control will be shown.



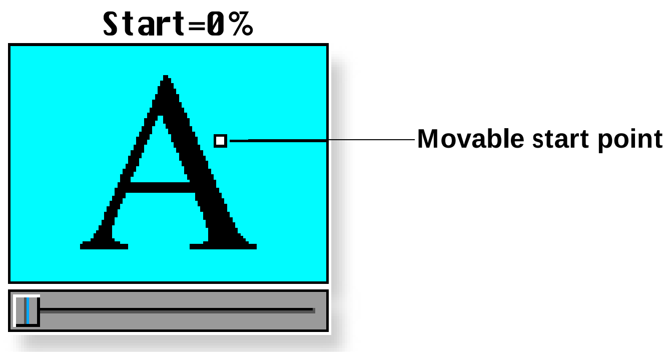
Fopt—eighth byte: Has edges flag

The sixth byte of the Fopt resource is a boolean flag that tells Premiere whether the transition has well-defined edges that can have borders or anti-aliasing applied to them. Setting this flag will cause the anti-aliasing level control, the border thickness slider, and the border color controls to be shown.


Border
Color
None
Thick
Anti-aliasing level control
Border color and thickness control

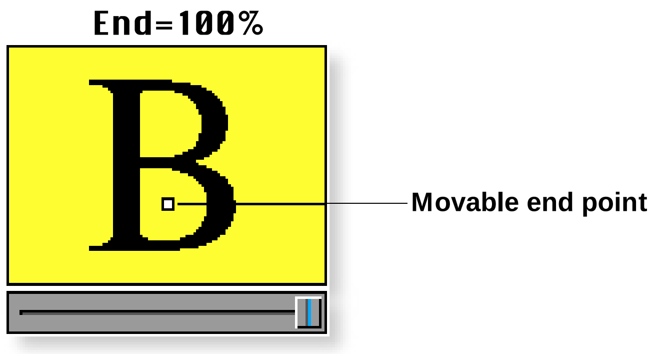
Fopt—ninth byte: Movable start point flag

The seventh byte of the Fopt resource is a boolean flag that tells Premiere whether the transition supports a movable start point. Setting this flag will cause the start point to show up in the start portion of the options dialog.



Fopt—tenth byte: Movable end point flag

The eighth byte of the Fopt resource is a boolean flag that tells Premiere whether the transition supports a movable end point. Setting this flag will cause the end point to show up in the end portion of the options dialog.



FXDF -1

This resource provides a four-byte tag that tells Premiere how to map this transition to one of the standard SMPTE wipes. Below is a table of valid values for this resource. Premiere uses this information when assigning wipe codes during the generation of an edit decision list.

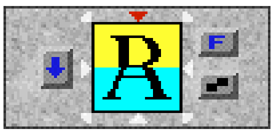
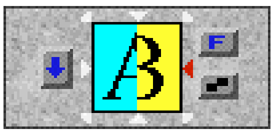
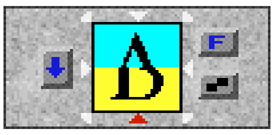
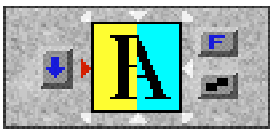
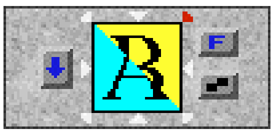
Table 4–2: SMPTE Wipe Values

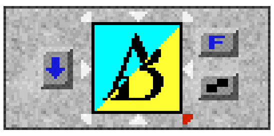
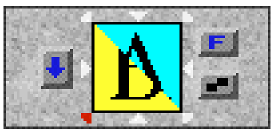
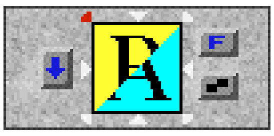
Tag	SMPTE wipe description
DISS	Cross dissolve
TAKE	“Take” or cut
WI00	Vertical wipe from the left edge

Table 4–2: SMPTE Wipe Values

Tag	SMPTE wipe description
WI01	Horizontal wipe from the top edge
WI02	Vertical wipe from the right edge
WI03	Horizontal wipe from the bottom edge
WI04	Diagonal wipe from upper left corner
WI05	Diagonal wipe from upper right corner
WI06	Diagonal wipe from lower right corner
WI07	Diagonal wipe from lower left corner
WI08	Vertical split wipe
WI09	Horizontal split wipe
WI10	Horizontal/vertical split wipe
WI11	Box wipe out from the center
WI12	Circular wipe from the center
WI13	Inset wipe from upper left
WI14	Inset wipe from upper right
WI15	Inset wipe from lower right
WI16	Inset wipe from lower left

There may be one or more FXDF resources for a transition. If there is only one FXDF, its resource ID must be -1. If the transition maps to different SMPTE wipe code based on the direction arrows, there may be an FXDF resource for each of the arrow settings, where the byte value of the arrow flags is the ID of the associated FXDF resource (up to a theoretical maximum of 256 FXDF resources). The following figure describes how the example Wipe SPFX module uses multiple FXDF resources:

Arrows	Bits	Value	FXDF
	bitTop	1	ID 1,"WI01"
	bitRight	2	ID 2,"WI02"
	bitBottom	4	ID 4,"WI03"
	bitLeft	8	ID 8,"WI00"
	bitUpperRight	16	ID 16,"WI05"

Arrows	Bits	Value	FXDF
	bitLowerRight	32	ID 32,"WI06"
	bitLowerLeft	64	ID 64,"WI07"
	bitUpperLeft	128	ID 128,"WI04"

The Transition Code

The entry point of the code should be declared like this:

```
int PRMEXPORT xEffect (short selector, EffectHandle theData) {...
```

The return value should be noErr (0) if the transition completed without error. Return any non-zero value to indicate an error. In such case, Premiere will fill in the destination frame with black.

The selector can take the following values:

Table 4–3: Transition Selector Values

Selector name	Value	Description
esExecute	0	Execute your transition.
esSetup	1	Execute your custom parameter dialog.

esExecute

The esExecute selector indicates that you should process the source frames and generate a destination frame. The specsHandle (described below) will contain your custom parameters, if any.

esSetup

The esSetup selector indicates that you should display your custom settings dialog. This call should use the specsHandle to fill the dialog with initial values, and should place the new values back into specsHandle. If no esSetup call has ever been made (or stored in a project), specsHandle will be nil. In such case you should provide reasonable default values, create a properly-sized Handle with the provided NewHandle() call, and place the handle into (*theData)->specsHandle, then show your dialog. Note that you will only get this message if you set bitCustom in your Fopt resource.

The EffectRecord Structure

Your transition is passed a handle to an EffectRecord through parameter theData. Here’s the structure of an EffectRecord:

```
typedef struct {
   Handle specsHandle;          // The specification handle
   PPixHand source1;            // Source pixels #1 (note change to 4.2)
   PPixHand source2;            // Source pixels #2 (note change to 4.2)
   PPixHand destination;        // Destination pixels (note change to 4.2)
   long part;                   // part / total = % complete
   long total;
   char previewing;             // In preview mode?
   unsigned 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
   BottleRec *bottleNecks;      // Bottleneck callback routines
   short version;               // The version of this record
   short sizeFlags;             // Frame processing flags
   long flags;                  // Audio flags
   short fps;                   // Frame rate in frames per second
} EffectRecord, **EffectHandle;
```

The fields are used as follows:

specsHandle
The specsHandle field holds transition-defined data which contains all the current settings for this transition. The transition normally creates this Handle when it gets a esSetup call. Premiere saves this Handle in the project file so that settings are restored when a project is reopened. This field is only used by transitions that have custom parameters. It must be created using NewHandle().

source1
The source1 field is the PixMap pointer for source image 1 (normally corresponding to the A video track). It will always be 32 bits deep.

source2
The source2 field is the PixMap pointer for source image 2 (normally corresponding to the B video track). It too will always be 32 bits deep, and the same size as source1.

destination
The destination field is the PixMap pointer for the destination image. This is where you store the calculated frame on an esExecute call. It will always be 32 bits deep and the same size as source1 and source2. When processing the two sources into the destination, the alpha channels of the sources may contain useful data, and should be processed just like the red, green, and blue channels. If the destination alpha channel is distorted or destroyed, automatic anti-aliasing and colored bordering may malfunction for your transition.

part
The part field tells you how far into the transition you are in frames. Part varies from 0 to total (described next), inclusive.

total
The total field tells you how many frames the transition covers in total. By dividing part by total, you can calculate the percentage of the transition that you should perform for a given esExecute call.

previewing
The previewing field is a flag that is no longer supported. You may ignore its value.

arrowFlags
The arrowFlags field gives you the corner flags (using the same bit definitions described above in the Fopt resource section) as set by the user.

reverse
The reverse field is a flag telling you that Premiere is performing the transition in reverse. Premiere automatically calls your transition with the frames in the reverse order. The flag is provided for informational purposes, normally you don't need to do anything differently.

source
The source field is a flag telling you the Premiere has swapped the source PixMaps (that is, you're doing the transition from video track B-to-A instead of A-to-B). The flag is provided for informational purposes, normally you don't need to do anything differently.

start
The start field is the start point of the transition as specified by the user. You only look at this field if the "movable start point" flag is turned on in the Fopt resource for your transition. This point is relative to the center point described below.

end
The end field is the end point of the transition as specified by the user. You only look at this field if the "movable end point" flag is turned on in the Fopt resource for your transition. This point is relative to the center point described below.

center
The center field is the normal center point for transitions that open and close. The start and end fields described above are measured relative to center.

privateData
The privateData field is a Handle of data that is private to Premiere. It is passed to the frame-retrieval callback (described below) when you need to get a frame from some other point in time. It should be created with the NewHandle call.

callBack
The callBack field contains a pointer to a routine you can use to get past or future frames from the source clips. This field is always available during the esExecute call but is only valid during the esSetup call when the bitUsesSource bit is set in the flags byte of the Fopt resource. FXCallBackProcPtr is defined as follows:

```
typedef pascal short (*FXCallBackProcPtr)(
    long frame,
    short track,
    PPixHand thePort,
    RECT *theBox,
    Handle privateData);
```

The frame parameter is the desired frame, from 0 to total inclusive. The track parameter is 0 for the A video track, 1 for the B video track. The thePort parameter is the destination for the frame, normally a locally allocated PWorld. The theBox parameter is the destination rectangle in thePort. Finally, the privateData parameter is (*theData)->privateData.

bottleNecks

The bottleNecks field is a pointer to a standard Premiere bottleneck record, as described in the Bottlenecks chapter of this documentation. You may use these routines to help you perform your transition.

version

The version field tells the version of this EffectRecord. Currently this field is set to zero.

sizeFlags

The sizeFlags field gives you some information about the preview or output options that are in effect. The following bit flags are of interest:

Table 4–4: EffectRecord Size Flags

Flag	Description
gvHalfV	User has specified half-vertical processing
gvHalfH	User has specified half-horizontal processing
gvFieldsEven	User has specified field-based processing, even fields first
gvFieldsOdd	User has specified field-based processing, odd fields first

To perform field processing, Premiere splits each frame into even and odd fields (each image being half-height) and calls your transition once for each field, then reassembles the two fields into a single frame. This allows transitions such as wipes to have field-based, 60-position-per-second motion. Beware that when field processing is turned on, (*theData)->total will be twice as big, and each frame will be half-height. Many transitions must special-case this situation in order to have the proper appearance.

flags

The flags field contains audio flags. These are not used by transitions.

fps

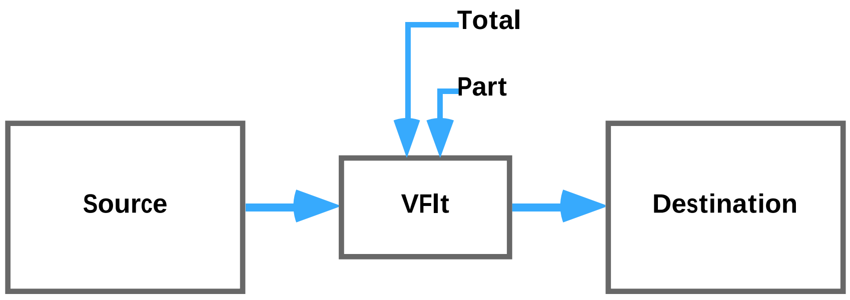
The fps field gives the frame rate in frames-per-second, in case your transition needs to know this information.

Examples

The Adobe Premiere Plug-In SDK comes with source code for a transition module.

5 Video Filters

A video filter in Adobe Premiere takes a single source PixMap and processes it into a destination PixMap.



Both PixMaps are always 32-bits deep. Video filters may be time-variant, and the filter is given the total duration in frames of the filter and the current frame number. Video filters may present a user interface and store parameters that specify how they process frames. Premiere handles the overhead of retrieving and storing frames.

Video filter modules have a file prefix of 'FL-' and an extension of .prm. Video filter files contain several kinds of resources which are listed below. Following the table is a detailed description of each resource.

Table 5–5: Video Filter Resources

Type & ID	Description
FLvs 1000	A two-byte version number stored as a short integer. The current version is 2.
TYPE 1000	The TYPE resource identifies the plug-in type to Premiere during the loading process.
TEXT 1000	The name of the resource, to be displayed in the Transitions window.
FltD 1	An optional time-animation resource describing the format of your settings data record.

FLvs 1000

This resource is two bytes in length and gives the version of the module interface. The current version is $0002. See the Plug-In Resources section in the Introduction for an example.

TYPE 1000

The TYPE 1000 resource is checked by Premiere at startup and identifies the plug-in type. Video filter modules have a TYPE of 'VFlt'. See the Plug-in Resources section in the Introduction.

TEXT 1000

This resource contains the title of the transition resource. It appears in the Filter dialog to identify a video filter by name. Look at Premiere’s filter names for examples of how this text should be worded.

FLTD 1

This optional resource is used to cause Premiere to automatically add time based effects to a filter. You would use it if your filter code does not explicitly handle time based effects, such as converted Photoshop or other imaging code. It contains a variable length description of your filter’s parameters (that is, the format of the data you choose to store in the specsHandle field of the VideoRecord structure described below).

If your filter does not directly support variability over time but such functionality is meaningful, you can include a FltD resource in your filter’s resource file. The presence of this type of resource makes Premiere enable the Filter dialog box “Start” and “End” settings buttons and save a copy of the specsHandle for each endpoint. When the execute message is sent to your filter, Premiere uses the information in your FLTD to interpolate between the values in the two specsHandles over time and pass the plug-in a single specsHandle with those calculated values.

Let’s look at an example specsHandle data structure and the associated FLTD resource:

```
typedef struct
{
   long randomSeed;  // Random number seed for my filter
   short horiz;      // The horizontal offset (user setting)
   short vert;       // The vertical offset (user setting)
   float scale;      // The scale factor (user setting)
   long magicNumber; // The magic number that my filter uses
} MyFilterSpecs;
```

The randomSeed value is calculated once and stored, and we don’t want Premiere to change it. The horiz, vert and scale fields are settings values that the user sets with the filter’s settings dialog. We want Premiere to interpolate those. The magicNumber field contains a magic number we use in our filter calculation, which we don’t want Premiere to interpolate. So here’s what the FLTD resource would look like:

```
1 FLTD DISCARDABLE
BEGIN
   0x0000,
   pdOpaque, 4,      // Don’t interpolate the random number seed
   pdShort, 0,       // Interpolate the short horiz value
   pdShort, 0,       // Interpolate the short vert value
   pdFloat, 0,       // Interpolate the float scale value
   pdOpaque, 4,      // Don’t interpolate the magic number
END
```

Each FLTD element is a type followed by a repeat count. The repeat count is only valid for the pdOpaque type and should be zero for all other types. To keep Premiere from interpolating the variable randomSeed, we specified that the first four bytes of our data structure are opaque by specifying “pdOpaque, 4,”. For the other fields we’ve simply informed Premiere of the data types. Note that the FLTD must describe each parameter in the filter data structure.

The valid type identifiers for the resource are:

```
//-----------------------------------------------------------------------
// Descriptor for allowing filters to animate over time. A structure of
// this type can be added to a 'VFlt', an 'AFlt', or a PhotoShop filter to
// describe the data structure of its parameters. Specify pdOpaque for any
// non-scalar data in the record, or data that you don't want Premiere to
// interpolate for you. Make the FLTD describe all the bytes of the
// parameter block. Use ID 1.
//-----------------------------------------------------------------------
// Specifies the type of the data
#define pdOpaque 0x0000        // Opaque - don't interpolate this
                               // Followed by count of bytes to skip
                               // with pdOpaque, eg, pdOpaque, 4
#define pdChar 0x0001          // Interpolate as signed byte
#define pdShort 0x0002         // Interpolate as signed short
#define pdLong 0x0003          // Interpolate as signed long
#define pdUnsignedChar 0x0004  // Interpolate as unsigned byte
#define pdUnsignedShort 0x0005 // Interpolate as unsigned short
#define pdUnsignedLong 0x0006  // Interpolate as unsigned long
#define pdExtended 0x0007      // Interpolate as a double
                               // Note-This is different from the Mac
#define pdDouble 0x0008        // Interpolate as a double
#define pdFloat 0x0009         // Interpolate as a float
```

The Filter Code

The entry point of the code should be declared like this:

```
int PRMEXPORT xFilter (short selector, VideoHandle theData) {...
```

The return value should be noErr (0) if the filter completed without error. Return any non-zero value to indicate an error. In such case Premiere will fill the destination frame with black.

The selector can take the following values:

Table 5–6: Video Filter Selector Values

Selector name	Value	Description
fsExecute	0	Execute your video filter.
fsSetup	1	Execute your settings dialog, if any.
fsDisposeData	2	Dispose of any instance data you may have created.

fsExecute
The fsExecute selector indicates that you should process the source frame and generate a destination frame. The specsHandle (described below) will contain all of your filter settings so you know how the source frame should be processed to generate the destination frame.

fsSetup
The fsSetup selector indicates that you should display your filter settings dialog box. You should use the information in the specsHandle to fill the dialog with initial values, and should place the new values back into specsHandle. If no fsSetup call has ever been made (or stored in a project), specsHandle will be nil. In such case you should provide reasonable default values, create a properly-sized handle with the NewHandle() call, place that handle into specsHandle, then show your dialog.

fsDisposeData

The fsDisposeData selector was added in Premiere 4.2. Premiere will send this selector when it is time to dispose of any instance data you have have created. See the new InstanceData member of the VideoRecord structure for further information.

The VideoRecord Structure

Your video filter is passed a handle to a VideoRecord through the parameter theData. Here's the structure of a VideoRecord:

```
typedef struct
{
   Handle specsHandle;               // The specification handle
   PPixHand source;                  // The source PixMap
   PPixHand destination;             // The destination PixMap
   long part;                        // part/total = %complete
   long total;
   char previewing;                  // In preview mode?
   Handle privateData;               // Private data handle
   VFilterCallBackProcPtr callBack;  // Callback, invalid if nil
   BottleRec *bottleNecks;           // Bottleneck callbacks
   short version;                    // Version of this record
   short sizeFlags;                  // Frame processing flags
   long flags;                       // Audio flags
   short fps;                        // Frame rate in frames/sec
   Handle InstanceData               // New in 4.2 - Private data for filter
} VideoRecord, **VideoHandle;
```

The fields are used as follows:

specsHandle

The specsHandle field holds filter-defined data which contain all the current settings for this filter. The filter normally creates this handle when it gets an fsSetup call. Premiere saves this handle in the project file so that settings are restored when a project is reopened. It must be created using NewHandle().

source

The source field is the PixMap pointer for the source image. It will always be 32 bits deep.

destination

The destination field is the PixMap pointer for the destination image. This is where you store the calculated frame on an fsExecute call. It will always be 32 bits deep and the same size as source. When processing the source into the destination, the alpha channel of the source may contain useful data, and should be processed just like the red, green, and blue channels.

part

The part field tells you how far into the filter you are in frames. Part varies from 0 to total (described next) inclusive.

total

The total field tells you how many frames the filter covers in total. By dividing part by total, you can calculate the percentage of a time-variant filter that you should perform for a given fsExecute call. Time-invariant filters can ignore part and total.

previewing

The previewing field is a flag that is no longer supported. You may ignore its value.

privateData

The privateData field is a handle of data that is private to Premiere. It is passed to the frame-retrieval callback (described below) when you need to get a frame from some other point in time.

callback

The callBack field contains a pointer to a routine you can use to get past or future frames from the source clip. The VFilterCallBackProcPtr is defined as follows:

```
typedef short (CALLBACK *VFilterCallBackProcPtr) (
   long frame,
   PPixHand thePort,
   RECT *theBox,
   Handle privateData );
```

The frame parameter is the desired frame, from 0 to total inclusive. The thePort parameter is the PixMap pointer to the destination for the frame. Use the NewPWorld() call to create this if necessary. The theBox parameter is the destination rectangle in thePort. Finally, the privateData parameter is the parameter described above. Use (*theData)->privateData to pass it.

bottleNecks

The bottleNecks field is a pointer to a standard Premiere bottleneck record, as described in the Bottlenecks chapter of this documentation. You may use these routines to help you perform your video filter.

version

The version field tells the version of this VideoRecord. In previous versions of Premiere this should have been set to 0. For Premiere 4.2 this should be set to 2.

sizeFlags

The sizeFlags field gives you some information about the preview or output options that are in effect. The following bit flags are of interest:

Table 5–1: VideoRecord Size Flags

Flag	Description
gvHalfV	User has specified half-vertical processing
gvHalfH	User has specified half-horizontal processing
gvFieldsEven	User has specified field-based processing, even fields first
gvFieldsOdd	User has specified field-based processing, odd fields first

Beware that when field processing is turned on, (*theData)->total will be twice as big, and each frame will be half-height. Some video filters must special-case this situation in order to have the proper appearance. See the description of this field in the Transitions chapter for more information.

flags

The flags field contains audio flags. These are not used by video filters.

fps
The fps field gives the frame rate in frames-per-second, in case your filter needs to know this information.

InstanceData
New in Premiere 4.2, this field allows a plug-in to have Premiere save and return some private data between invocations. You are responsible for allocating and freeing any memory used with this field. You would probably allocate memory for this field when getting an fsSetup selector, but you must deallocated it when getting an fsDisposeData. To utilize this new field, the version field above must be set to 2.

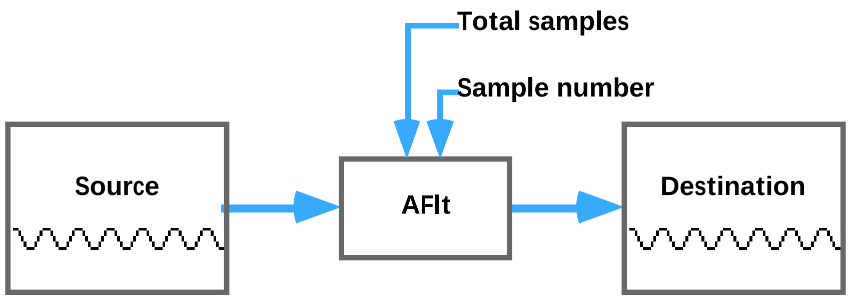
Examples

The Adobe Premiere Plug-In SDK comes with source code for a video filter module that you can use as an example of how to write your own.

Video Noise
This is the source code for the Video Noise filter that ships with Adobe Premiere. It is a good example of a basic video filter with no settings dialog.

6 Audio Filters

An audio filter in Adobe Premiere takes a source buffer of audio and processes it into a destination buffer.



Both buffers will be the same size. Audio filters may be time-variant, and the filter is given the total duration in samples of the filter and the sample number of the first sample in the source buffer. Audio filters may present a user interface and store parameters that specify how they process the audio. Premiere handles the overhead of retrieving and storing sound data.

Audio filter modules have a file prefix of 'FL-' and an extension of .prm. Audio filter files contain several kinds of resources which are listed below. Following the table is a detailed description of each resource.

Table 6–1: Audio Filter Resources

Type & ID	Description
FLvs 1000	A two-byte version number stored as a short integer. The current version is 2.
TYPE 1000	The TYPE resource identifies the plug-in type to Premiere during the loading process.
TEXT 1000	The name of the resource, to be displayed in the Transitions window.
FltD 1	An optional time-animation resource describing the format of your settings data record.

FLvs 1000

This resource is two bytes in length and gives the version of the module interface. The current version is $0002. See the Plug-In Resources section in the Introduction for an example.

TYPE 1000

The TYPE 1000 resource is checked by Premiere at startup and identifies the plug-in type. Audio filter modules have a TYPE of 'AFlt'. See the Plug-in Resources section of the Introduction.

TEXT 1000

This resource contains the title of the audio filter resource. It appears in the Filter dialog to identify a transition by name. Look at Premiere's filter names for examples of how this text should be worded.

FltD 1

This optional resource is used to cause Premiere to automatically add time based effects to a filter. You would use it if your audio filter code does not explicitly handle time based effects. It contains a variable length description of your filter's parameters (that is, the format of the data you choose to store in the specsHandle field of the AudioRecord structure described below).

If your filter does not directly support variability over time but such functionality is meaningful, you can include a FltD resource in your filter's resource file. The presence of this type of resource makes Premiere enable the Filter dialog box "Start" and "End" settings buttons and save a copy of the specsHandle for each endpoint. When the execute message is sent to your filter, Premiere uses the information in your FltD to interpolate between the values in the two specsHandles over time and pass the plug-in a single specsHandle with those calculated values.

For more information about FltD resources, see the description in the chapter Video Filters.

The Filter Code

The entry point of the code should be declared like this:

```
int PRMEXPORT xFilter(short selector, AudioFilter theData) {...
```

The return value should be noErr (0) if the filter completed without error. Return any non-zero value to indicate an error. In such case Premiere will fill the destination frame with the original data.

The selector can take the following values:

Table 6–2: Audio Filter Selector Values

Selector name	Value	Description
fsExecute	0	Execute your audio filter.
fsSetup	1	Execute your settings dialog, if any.
fsDisposeData	2	Dispose of any instance data you may have created.

fsExecute

The fsExecute selector indicates that you should process the source buffer and generate a destination buffer. The specsHandle (described below) will contain all of your filter settings so you know how the audio data should be processed to generate the destination data.

fsSetup

The fsSetup selector indicates that you should display your filter settings dialog box. You should use the information in the specsHandle to fill the dialog with initial values, and should place the new values back into specsHandle. If no fsSetup call has ever been made (or stored in a project), specsHandle will be nil. In such case you should provide reasonable default values, create a properly-sized handle with the NewHandle() call, place that handle into specsHandle, then show your dialog.

fsDisposeData

The fsDisposeData selector was added in Premiere 4.2. Premiere will send this selector when it is time to dispose of any instance data you have have created. See the new InstanceData member of the AudioRecord structure for further information.

The AudioRecord Structure

Your audio filter is passed a handle to an AudioRecord through the parameter theData. Here’s the structure of an AudioRecord:

```
typedef struct
{
   Handle specsHandle;              // The specification handle
   Ptr source;                      // The source buffer
   Ptr destination;                 // The destination buffer
   long samplenum;                  // First sample number
   long samplecount;                // Num of samples in source
   char previewing;                 // In preview mode?
   Handle privateData;              // Private data handle
   AFilterCallBackProcPtr callBack; // Callback, invalid if nil
   long totalsamples;               // Total samples in clip
   short flags;                     // Audio flags
   long rate;                       // Sample rate, 16.16 Fixed
   BottleRec *bottleNecks;          // Bottleneck callbacks
   short version;                   // Version of this record
   long extraFlags;                 // Other flags
   short fps;                       // Frame rate in frames/sec
   Handle InstanceData              // New in 4.2 - private data for filter
} AudioRecord, **AudioFilter;
```

The fields are used as follows:

specsHandle

The specsHandle field holds filter-defined data which contain all the current settings for this filter. The filter normally creates this handle when it gets an fsSetup call. Premiere saves this handle in the project file so that settings are restored when a project is reopened. This must be allocated with NewHandle().

source

The source field points to the source audio buffer. This buffer contains sampleCount samples, starting at sampleNum in the clip being processed. During an fsSetup call, this will be nil. At esExecute time, this buffer contains the data you are to filter.

destination

The destination field points to the destination audio buffer. This is where you store the calculated audio data on an fsExecute call. It will always be the same size as source. At esSetup time this buffer is nil.

sampleNum

The sampleNum field tells you the sample number of the first sample in the source buffer.

IMPORTANT! *This value is in bytes, so you need to divide by the "bytes-per-sample" value to determine an actual sample number.*

The table below shows the possible number of bytes-per-sample based on the value of two bits in the flags field (described below):

Table 6–3: Audio Sampling Rate

Flags bits	Bytes per sample
0	1 (8-bit mono)
gaStereo	2 (8-bit stereo)
ga16Bit	2 (16-bit mono)
ga16Bit | gaStereo	4 (16-bit stereo)

sampleCount

The sampleCount field tells you how many bytes are in source and destination. Note that, like sampleNum, you need to divide by bytes-per-sample to determine the actual sample count in samples.

previewing

The previewing field is a flag that is no longer supported. You may ignore its value.

privateData

The privateData field is a handle of data that is private to Premiere. It is passed to the audio-retrieval callback (described below) when you need to get audio from some other point in time.

callback

The callBack field contains a pointer to a routine you can use to get past or future audio data from the source clip. The AFilterCallBackProcPtr is defined as follows:

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

The sample parameter is the desired starting sample number, from 0 to totalsamples - 1 inclusive, in bytes. The count parameter specifies the number of bytes you wish to retrieve. The buffer parameter is the destination buffer for the audio data, which is usually a locally allocated. Finally, the privateData argument is the field described above, passed as (*theData)->privateData.

totalsamples
The totalsamples field tells you the total number of bytes in the filtered clip. Divide by bytes-per-sample to determine the total number of samples.

flags
The flags field describes the audio data in the source buffer. It may have either of the following two flags set:

```
  #define gaStereo 0x0100
  #define ga16Bit  0x0200
```

Using these flags you can tell the number of bytes in the buffer. Your output should be in the same format.

rate
The rate field provides the sample rate as a integer value in samples per second. For instance, if a clip contained sound data at the standard sample rate 22 kHz, rate would contain 22050. This is for informational purposes in case your filter needs it. Note that it is different from the Premiere for Macintosh API.

bottleNecks
The bottleNecks field is a pointer to a standard Premiere bottleneck record, as described in the Bottlenecks chapter of this documentation. You may use these routines to help you perform your audio filter.

version
The version field tells the version of this VideoRecord. In previous versions of Premiere this should have been set to 0. For Premiere 4.2 this should be set to 2.

extraFlags
The extraFlags field is a flags field for future use by Adobe. Currently it is set to zero.

fps
The fps field gives the frame rate in frames-per-second, in case your filter needs to know this information.

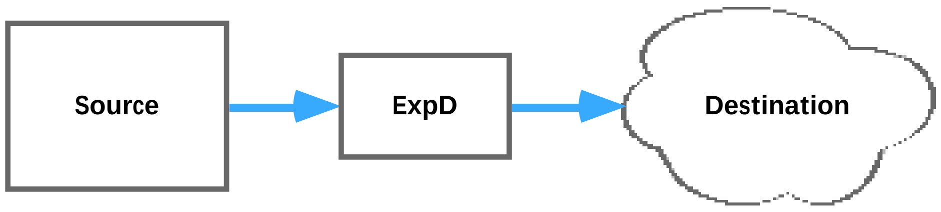
InstanceData
New in Premiere 4.2, this field allows a plug-in to have Premiere save and return some private data between invocations. You are responsible for allocating and freeing any memory used with this field. You would probably allocate memory for this field when getting an fsSetup selector, but you must deallocated it when getting an fsDisposeData. To utilize this new field, the version field above must be set to 2.

Examples

The Adobe Premiere Plug-In SDK comes with source code for an audio filter module that you can use as an example of how to write your own.

7 Data Export Modules

A data export module in Adobe Premiere is called when the user opens some kind of clip window and chooses an item from the Export submenu of the File menu. The export module's job is to export the given clip to some other format.



The export module tells Premiere whether it can export audio, video, or both. It is provided with information about the source clip and two callback routines to allow it to retrieve audio and video from the clip. Export modules normally put up a modal dialog asking the user for appropriate export parameters, then put up a file dialog to request a destination file. They then export the source clip into another file format.

Adobe Premiere export modules are not limited to file-format-conversion type export operations. Premiere's "Print To Video" export module is a good example of a different kind of module, where the output is to the screen rather than to a file.

Data export modules have a file prefix of 'EX-' and an extension of .prm. Data export files contain several kinds of resources which are listed below. Following the table is a detailed description of each resource.

Table 7–1: Data Export Resources

Type & ID	Description
EXvs 1000	A two-byte version number stored as a short integer. The current version is 2.
TYPE 1000	The TYPE resource identifies the plug-in type to Premiere during the loading process.
TEXT 1000	The name of the resource, to be displayed in the Export menu.
FLAG 1000	A two-byte flags word that tells the capabilities of the data export module.

EXvs 1000

This resource is two bytes in length and gives the version of the module interface. The current version is $0002. See the Plug-In Resources section in the Introduction for an example.

TYPE 1000

The TYPE 1000 resource is checked by Premiere at startup and identifies the plug-in type. Data export modules have a TYPE of ‘ExpD. See the Plug-In Resources section in the Introduction.

TEXT 1000

This resource contains the title of the data export module. It appears in the Export menu to identify a export method by name. Look at Premiere’s export menu commands for examples of how this text should be worded.

FLAG 1000

This resource tells Premiere whether the data export module can export video, audio, or both. Premiere uses this information to dim or undim the export module’s menu item in the Export submenu based on the type of clip that is in the front window. The structure of a FLAG resource is shown below, along with some bit values you’ll use with the definition:

```
// This identifies the type of data the exporter can handle
// Valid values are:
#define mExpVid 0x8000 // Can export video/still image data
#define mExpAud 0x4000 // Can export audio data

1000 FLAG DISCARDABLE
BEGIN
   0x0000,
   0x??00 // the export capabilities go here
END
```

For example, if your export module can export both audio and video, you’d make a FLAG resource containing mExpVid + mExpAud.

The Export Code

The entry point of your code should be declared like this:

```
int PRMEXPORT xExport (short selector, DataExportHandle theData) {...
```

The return value is currently ignored, but you should return noErr (0) for future compatibility.

The selector can take the following values:

Table 7–2: Data Export Selector Values

Selector name	Value	Description
edExecute	0	Execute your data export process.

edExecute

The edExecute selector indicates that you should perform your data export function. You may use the information in the DataExportRec (described below) to help you.

The DataExportRec Structure

Your data export module is passed a handle to a DataExportRec through the parameter theData. Here's the structure of a DataExportRec:

```
typedef struct {
   long markers[12];          // Clip markers (0 = in, 1 = out)
   long numframes;            // Number of frames in the clip
   short framerate;           // Frames/second of source material
   RECT bounds;               // Video bounds box, empty if no video
   short audflags;            // Audio flags, zero if no audio
   long audrate;              // The audio rate in Hz
   GetVidCallBack getVideo;   // Video reader callback
   GetAudCallBack getAudio;   // Audio reader callback
   Handle privateData;        // Private data for above routines
   long specialRate;          // Special rate
} DataExportRec, **DataExportHandle;
```

The fields are used as follows:

markers

The markers field is an array of twelve clip markers. The value of markers[0] is the clip's in-point, markers[1] its out-point. Entries 2-11 are the numbered markers 0-9. The marker values are in the time units specified by the framerate field (described below).

Note: On the Macintosh platform, the markers array is vestigial since its API supports access to unnumbered markers. Developers writing a cross-platform export plug-in should be aware of this.

numframes

The numframes field specifies the duration of the source material in the units specified by the framerate field.

framerate

The framerate field gives the frame rate in frames per second at which this export operation is being performed. It corresponds to the Time Base preference setting for clip windows.

bounds

The bounds field specifies bounds for the video portion of the source clip's data. If it is an empty rectangle, there source clip contains no video.

audflags

The audflags field describes the audio data in the source clip. It may have either of the following two flags set:

```
   #define gaStereo 0x0100
   #define ga16Bit  0x0200
```

These flags are the same as are used in audio filters and with the audio bottleneck routines.

audrate
The rate field provides the sample rate as a integer value in samples per second. For instance, if a clip contained sound data at the standard sample rate 22 kHz, audrate would contain 22050. If audrate is 0, the source clip contains no audio.

getVideo
The getVideo field contains a pointer to a routine you can use to get video data from the source clip. GetVidCallBack is defined as follows:

```
// Callback to get one frame of video
typedef short (CALLBACK *GetVidCallBack) (
   long frame,
   PWorldID thePort,
   LPRECT *theBox,
   LPVOID privateData );
```

The frame parameter is the desired frame in the units defined by the framerate field documented above. For example, you would use (*theData)->markers[0] to retrieve the frame at the in-point. The thePort parameter is the PWorldID of the destination buffer for the frame. You can use NewHandle() to allocate a local PWorld. The theBox parameter is the destination rectangle in thePort. Finally, the privateData parameter is the record field discussed below and accessed as (*theData)->privateData.

getAudio
The getAudio field contains a pointer to a routine you can use to get audio data from the source clip. The GetAudCallBack is defined as follows:

```
// Callback to get one second of audio
typedef short (CALLBACK *GetAudCallBack) (
   long second,
   short formatFlags,
   LPSTR  buffer,
   LPVOID  privateData );
```

The second parameter is the desired second (such as second #0, second #1, etc.). The formatFlags parameter is ignored by the Windows version of Premiere. The buffer parameter is the destination buffer for the audio data. Following is a table showing you how big your buffer should be based on the flag values:

Table 7–3: Audio Buffer Sizes

Sample rate flags	8-bit mono	8-bit stereo	16-bit mono	16-bit stereo
aflag11KHz	11025	22050	22050	44100
aflag11KHz + aflagDropFrame	11014	22028	22028	44056
aflag22KHz	22050	44100	44100	88200
aflag22KHz + aflagDropFrame	22028	44056	44056	88112
aflag44KHz	44100	88200	88200	176400
aflag44KHz + aflagDropFrame	44056	88112	88112	176224

Note: The values in this table are different than those for the Premiere for Macintosh API.

Finally, the privateData parameter is a field of the DataExportRec accessed as (*theData)->privateData.

privateData
The privateData field is a handle of data that is private to Premiere. It is passed to the getVideo and getAudio callbacks.

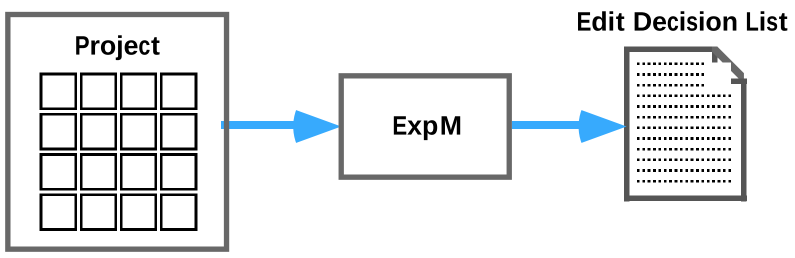
specialRate
This field is ignored by the Windows version of Premiere.

Examples

The Adobe Premiere Plug-In SDK comes with source code for a data export module that you can use as an example of how to write your own.

8 EDL Export Modules

An EDL export module in Adobe Premiere is called when the user chooses an EDL export item from the Export submenu of the File menu. The export module's job is to export the current project into a text edit decision list (EDL) format. Usually these EDL text files are used to drive a hardware device, such as a video switcher like the CMX 3600.



The EDL export module is provided with a nested, block-formatted data structure that describes the project that the user has assembled in Premiere's Construction window. Export modules normally unroll this data structure and generate a text file. It is also possible for an EDL module to directly control a hardware device to perform autoassembly of the project from source tapes.

EDL export modules have a file prefix of 'EX-' and an extension of .prm. EDL export module files contain several kinds of resources which are listed below. Following the table is a detailed description of each resource:

Table 8–4: EDL Export Module Resources

Type & ID	Description
EXvs 1000	A two-byte version number stored as a short integer. The current version is 2.
TYPE 1000	The TYPE resource identifies the plug-in type to Premiere during the loading process.
TEXT 1000	The name of the resource, to be displayed in the Export menu.

EXvs 1000

This resource is two bytes in length and gives the version of the module interface. The current version is $0002. See the Plug-in Resources section of the Introduction for an example.

TYPE 1000

The TYPE 1000 resource is checked by Premiere at startup and identifies the plug-in type. Data export modules have a TYPE of 'ExpM'. See the Plug-in Resources section of the Introduction.

TEXT 1000

This resource contains the title of the EDL export module. It appears in the Export menu to identify a export method by name. Look at Premiere’s export menu commands for examples of how this text should be worded.

The Export Code

The entry point of the code should be declared like this:

```
int PRMEXPORT xExport (short selector, ExportHandle theData) {...
```

The return value is currently ignored for the exExecute message but is used with the exTrue30fps message. In order to ensure future compatibility, you should return noErr (0) from your exExecute message.

The selector can take the following values:

Table 8–5: EDL Selector Values

Selector name	Value	Description
exExecute	0	Execute your EDL export process.
exTrue30fps	1	Tell Premiere whether you want edits in 29.97 or 30 fps.

exExecute

The exExecute selector indicates that you should perform your EDL export function. You will use the information in the ExportRecord (described below) to generate your list.

exTrue30fps

The exTrue30fps selector is made just before the exExecute call. If you return a result code of 1, Premiere passes all of the data in the project at 30 frames per second. If you return 0, Premiere converts all of the times to 29.97 frames per second, which is the frame rate at which color video actually runs. Normally an EDL export module should defer such time conversion to Premiere rather than attempting it within the module.

The ExportRecord Structure

Your EDL export module is passed a handle to a ExportRecord through the parameter theData. Here’s the structure of a ExportRecord:

```
typedef struct
{
   Handle dataHandle;  // The project data handle
   short timeBase;     // The current default timebase
   Ptr projectName;    // A pointer to current project name
} ExportRecord, **ExportHandle;
```

The fields are used as follows:

dataHandle

The dataHandle field contains a hierarchical block of data which describes everything about an Adobe Premiere project. This format is described in detail below in the section entitled The EDL Project Data Format.

timeBase

The timeBase field tells your EDL export module the basic frame rate. It will be 24, 25, or 30. When timebase is 30, the actual time base depends on your response to the exTrue30fps message. If you returned 0 in response to the exTrue30fps message, the actual rate is 29.97; if you returned 1, the rate is 30.00.

projectName

The projectName field gives you with the name of the project. This is usually used as the basis of the default name of an output EDL text file.

The EDL Project Data Format

When your EDL export module gets an exExecute message, the entire current Premiere project will be handed to your plug-in via the dataHandle field of the ExportRecord. The data is in a hierarchical, block-structured format. Each block has the following structure:

```
typedef struct
{
   long size;      // Total block size, w/static data & sub-blocks
   long dataSize;  // The static data size for this block
   long type;      // The block type (basically an OSType)
   long theID;     // Block ID or 0 for blocks that don’t need an ID
} BlockRec;
```

Following the header is a block of local static data owned by this block of the size given in the dataSize field of the BlockRec. Following the local static data is a series of zero or more sub-blocks, each with their own block headers (and potentially their own data chunks and sub-blocks). The types and IDs for the currently defined blocks are listed in the following table. The types are already #defined in the Premiere header file by using the format bTYPE, where TYPE is the four character Type in the table, e.g. use bTRKB for the block type ‘TRKB’, use bRATE for the block type ‘RATE’.

Table 8–6: EDL Types and IDs

Type	ID	Parent	Data	Description
'BLOK'	0	none	L-wrk strt, L-wrk end, sub-blks	Container for everything
'TRKB'	0	BLOK	track blocks	Container for all of the tracks
'TRAK'	ID	BLOK	S-flags, TREC blocks	Contains all of the blocks for an entire track
'FVID'	0	TRAK	none	Flag: track contains video records
'FSUP'	0	TRAK	none	Flag: track contains superimpose records
'FAUD'	0	TRAK	none	Flag: track contains audio records
'AMAP'	0	FAUD	S-audio mapping bits	Bits indicate target audio tracks
'FF_X'	0	TRAK	none	Flag: track contains F/X records

Table 8–6: EDL Types and IDs

Type	ID	Parent	Data	Description
'TREC'	n	TRAK	S-clipID, L-strt, L-end, sub-blks	Contains the blocks for a single track item
'RBND'	0	TREC	S-max, RPNT blocks	[The rubber band info for a track item]
'RPNT'	0-n	RBND	L-h, S-v	Rubber band point
'FXOP'	0	TREC	C-crnr, C-dir, S-strt, S-end, blks	[The options controlling F/X options]
'FXDF'	0	FXOP	OSType	The base type of the effect
'EDGE'	0	FXOP	S-thickness, COLR block	[Describes edge thickness]
'MPNT'	0	FXOP	Point	[Reference point for next two types]
'SPNT'	0	FXOP	Point	[User specified open point]
'EPNT'	0	FXOP	Point	[User specified close point]
'OVER'	0	TREC	S-type, info blocks	[The parameters for an overlay item]
'COLR'	0	OVER, FILE	RGBColor	[Key or fill color]
'SIMI'	0	OVER	S-similarity	[Similarity value]
'BLND'	0	OVER	S-blend	[Blend value]
'THRS'	0	OVER	S-threshold	[Threshold value]
'CUTO'	0	OVER	S-cutoff	[Cutoff value]
'ALIA'	0	OVER	S-level	[Anti-aliasing level]
'SHAD'	0	OVER	none	[Flag: shadowing is on]
'RVRS'	0	OVER	none	[Flag: key is reversed]
'GARB'	0	OVER	R-ref rect, point blocks	Garbage matte points
'PONT'	0-n	GARB, RBND	Point	
'MATI''	0	OVER	S-clipID	[The ID of the clip describing an overlay Matte]
'VFLT'	0	TREC	sub-blocks	[Followed by individual filter blocks]
'AFLT'	0	TREC	sub-blocks	[Followed by individual filter blocks]
'FILT'	0-n	VFLT, AFLT	S-fileID, data block	File ID followed by an opaque data block
'MOTN'	0	TREC	R-ref rect, sub blocks	[Record giving motion path for a track item]
'SMTH'	0	MOTN	none	Flag: motion path is smoothed
'MREC'	0-n	MOTN	S-zoom, P-spot, P-dest[4]	Describes each motion point
'DATA'	0	any	data block	[Generic block for storing parm handles]
'CLPB'	0	BLOK	clip blocks	Contains all of the clip blocks
'CLIP'	ID	CLPB	S-fileID, L-in, L-out	The descriptive info for a clip
'MARK'	0-9	CLIP	L-location	[For set markers, defines the markers]
'LOCK'	0	CLIP	none	[Flag: clip has locked aspect]
'RATE'	0	CLIP	S-rate * 100	[Defines a rate other than 1.00]

Table 8–6: EDL Types and IDs

Type	ID	Parent	Data	Description
'FILB'	0	BLOK	file blocks	Contains all of the file blocks
'FILE'	ID	FILB	info blocks	The descriptive blocks for a file
'MACS'	0	FILE	FSSpec	The Mac file spec
'MACP'	0	FILE	string	The full Mac pathname
'FRMS'	0	FILE	L-#frames	[Number of frames for a file w/content]
'VIDI'	0	FILE	L-video frame, S-depth	[Describes the video portion of the file]
'AUDI'	0	FILE	S-aud flags, L-aud rate	[Describes the audio portion of the file]
'TIMC'	0	FILE	timecode	[Gives the timecode for the first file frame]
'TIMB'	0	FILE	L-frame, C-dropframe, C-fmt	[Specifies the binary timecode, as above]
'REEL'	0	FILE	Str-reel name	[String giving the source reel for the file]

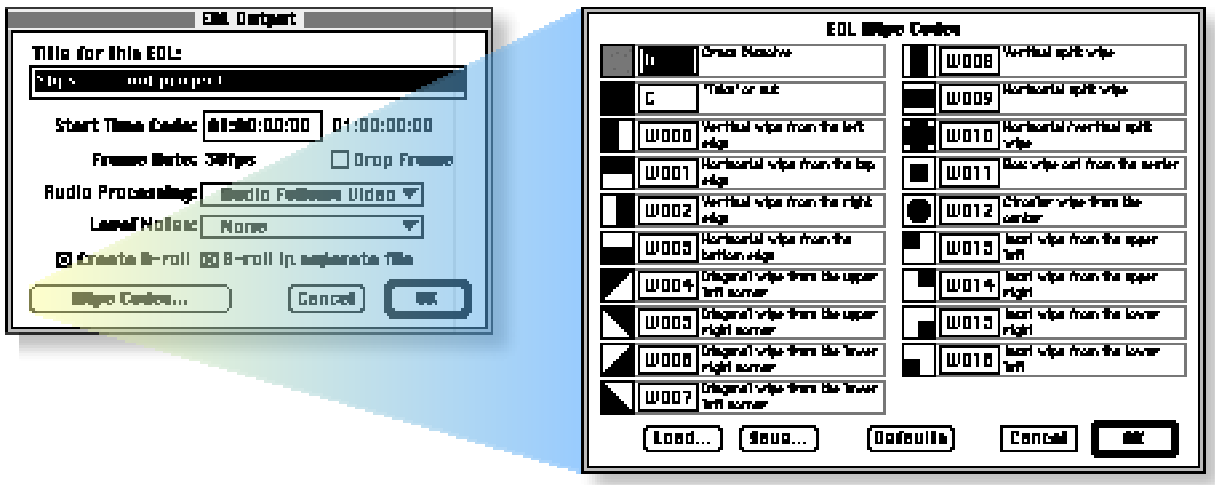
Abbreviation	Description
C-	char
S-	short
L-	long
P-	Point
R-	Rect
Flag:	If block is present, condition is true
[...]	Optional block

Many of the blocks have an associated structure that describes their contents. Those are listed below:

```
typedef struct
{
   long start;            // Starting position for the work area
   long end;              // Ending position for the work area
} Rec_BLOK;

typedef struct
{
   short fileID;          // The dependent file ID
   long in;               // The IN point within the source material
   long out;              // The OUT point within the source material minus 1
} Rec_CLIP;

typedef struct
{
   short clipID;          // The dependent clip ID
   long start;            // The clip starting position
   long end;              // The clip ending position
} Rec_TREC;

typedef struct
{
   short zoom;            // Zoom factor 1 to 400, 100 is normal
   short time;            // Time location 1 to 1000
   short rotation;        // Rotation factor -360 to 360, 0 is normal
   short delay;           // Delay factor 0 to 100, 0 is normal
   Point spot;            // The center point for the image at this point
} Rec_MREC;

typedef struct
{
   unsigned char corners;// User direction flags, one bit each
   char direction;        // Direction flag, 0 = A->B, 1 = B->A
   short startPercent;    // Starting percentage times 100
   short endPercent;      // Ending percentage times 100
} Rec_FXOP;

typedef struct
{
   long h;                // Horiz (time) loc of band point
   short v;               // Vertical (amplitude/level) loc of band point
} Rec_RPNT;

typedef struct
{
   Rect frame;            // Bounding frame for video data
   short depth;           // Bit depth for video data
} Rec_VIDI;

typedef struct
{
   long frames;           // Binary frame count
   char dropframe;        // true = DF, false = NDF
   char format;           // true=NTSC(30), false=PAL(25), 2=Film(24)
} Rec_TIMB;
```

Wipe Code Details

The EDL export modules that ship with Adobe Premiere allow the user to set up wipe codes for use in their EDL text files. The user can edit these wipe codes using the Wipe Code editor dialog, which is accessible from the dialog used to get the file name and other information.



EDL Output

Title for this EDL:
[illegible]

Start Time Code: 01:00:00:00 01:00:00:00

Frame Rate: 30fps ☐ Drop Frame

Audio Processing: Audio Follows Video

Level Notes: None

☒ Create B-roll ☒ B-roll in separate file

EDL Wipe Codes

D	Cross Dissolve	W008	Vertical split wipe
C	"Take" or cut	W009	Horizontal split wipe
W000	Vertical wipe from the left edge	W010	Horizontal / vertical split wipe
W001	Horizontal wipe from the top edge	W011	Box wipe out from the center
W002	Vertical wipe from the right edge	W012	Circular wipe from the center
W003	Horizontal wipe from the bottom edge	W013	Inset wipe from the upper left
W004	Diagonal wipe from the upper left corner	W014	Inset wipe from the upper right
W005	Diagonal wipe from the upper right corner	W015	Inset wipe from the lower right
W006	Diagonal wipe from the lower right corner	W016	Inset wipe from the lower left
W007	Diagonal wipe from the lower left corner		

You may want to provide a similar mechanism for obtaining a list of wipe codes to which you can map Premiere transition information. The Premiere for Macintosh API provides routines to facilitate collecting this information, but these are not provided by the Premiere for Windows API.

Relevant Routines in the Utility Library

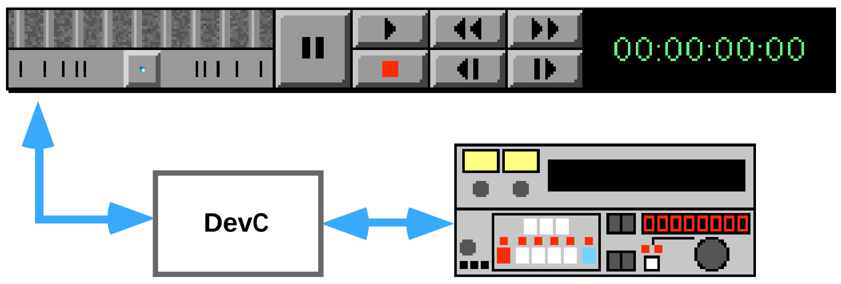
There are a few routines exported by Premiere that will make writing an EDL export module easier. See the definitions of the block routines in the Premiere Specific Routines section of the Utility Library chapter for details.

Examples

The Adobe Premiere Plug-In SDK comes with source code for an EDL export module that you can use as an example of how to write your own. It contains code to recursively unroll the project data handle and parse the blocks.

9 Device Control Modules

A device control module allows Adobe Premiere to control hardware devices such as tape decks or laser disc players.



Device control modules are called by parts of Premiere that take video input, like the Movie Capture window and the Waveform Monitor. A device control module's most important functions are to set hardware operating modes, tell Premiere what mode the hardware is in, and provide Premiere with timecode from the hardware.

Device control modules have a file prefix of 'X-' and an extension of .prm. Device control files contain several kinds of resources which are listed below. Following the table is a detailed description of each resource:

Table 9–7: Device Control Module Resources

Type & ID	Description
DVvs 1000	A two-byte version number stored as a short integer. The current version is 2.
TYPE 1000	The TYPE resource identifies the plug-in type to Premiere during the loading process.
TEXT 1000	The name of the resource, to be displayed in the Export menu.

Note: Any other resources contained in an device control module's resource file should have IDs in the range 600 to 999.

DVvs 1000

This resource is two bytes in length and gives the version of the module interface. The current version is $0002. See the Plug-In Resources section in the Introduction for an example.

TYPE 1000

The TYPE 1000 resource is checked by Premiere at startup and identifies the plug-in type. Data export modules have a TYPE of 'DevC'. See the Plug-In Resources section in the Introduction for an example.

TEXT 1000

This resource contains the name of the device controller. The name is what appears in the popup menu in the Device Control dialog box.

The Device Control Code

The entry point of the code should be declared like this:

```
int PRMEXPORT xDevice (short selector, DeviceHand theData) {...
```

The return value should be noErr (0) if no error occurs during the execution of your device control module, or any non-zero value if an error occurs. The selector can have the following values:

Table 9–8: Device Control Selector Values

Selector name	Value	Description
dsInit	0	Create data structures, choose an operating mode.
dsSetup	1	Put user settings dialog, if any.
dsExecute	2	Perform a specified device control command.
dsCleanup	3	Dispose data structures.
dsRestart	4	Restart module—used at startup to reconnect to a device.

dsInit

The dsInit selector tells your device control module to create its local data structure and store its handle in the deviceData field provided in the DeviceRec structure passed to you in the call. You should choose a default operating mode if more than one are available. If necessary, a dialog can be presented during this call to prompt the user for settings. If you need to open drivers or make serial connections to your hardware, you also do this here. See the Implementation Tips section below for more information about the dsInit selector.

dsSetup

The dsSetup selector tells your device control module to put up a custom settings dialog box, if any. This might include choosing between several device control methods or selecting a serial port. If your device control module doesn't require any additional parameters from the user, calls with the dsSetup selector can be safely ignored (but should return noErr).

dsExecute

The dsExecute selector tells your device control module to perform a device control operation based on (*theData)->command. See the Commands section below for a detailed description of the different commands and the actions you should take.

dsCleanup

The dsCleanup selector tells your device control module to disconnect from any hardware and dispose of its local data handle (that is, the data you may have stored in (*theData)->deviceData).

dsRestart

The dsRestart selector is just like dsInit, except that (*theData)->deviceData handle has already been set up. Premiere stores this information in the preferences file so that when Premiere is started up, connections to hardware devices can be reestablished. See the Implementation Tips section below for more information about the dsInit and dsRestart selectors.

The DeviceRec Structure

Your device control module is passed a handle to a DeviceRec through the parameter theData. Here's the structure of a DeviceRec:

```
typedef struct
{
   Handle deviceData;    // Local data which plug-in creates
   short command;        // The command to perform
   short mode;           // New mode (in) or current mode (out)
   long timecode;        // New timecode (in) or current (out)
   short timeformat;     // Format: 0=non-drop, 1=drop-frame
   short timerate;       // Frames/second for timecode above
   long features;        // Features (out) for cmdGetFeatures
   short error;          // Error code (out) from any routine
   short preroll;        // Pre-roll time (secs) for cmdLocate
   CallBackPtr callback; // Abort-check proc for cmdLocate
   ProcPtr PauseProc;    // Pause-current-operations proc
   ProcPtr ResumeProc;   // Resume-current-operations proc
   char reserved[256];   // Reserved;
} DeviceRec, **DeviceHand;
```

The fields are as follows:

deviceData

The deviceData field is where you should store a handle to your local data at dsInit time. Premiere stores this data in the Premiere preferences file for later retrieval (after which it is passed to a dsRestart handler). The value of this field is retained across calls.

command

The command field tells you what command is being executed when you get a call with the dsExecute selector. See the Commands section below for detailed information about this field's possible values.

mode

The mode field is used three ways. For dsExecute/cmdNewMode calls, mode contains the new mode into which Premiere is instructing you to put a device. For dsExecute/cmdStatus calls, mode is where you store the current mode of the device. The last mode you reported will still be there. For dsExecute/cmdShuttle calls, mode contains the shuttle rate, which may have the value -100 to 100. Negative values indicate you should shuttle backwards, positive values indicate that you should shuttle forward.

timecode

The timecode field is used three ways. For dsExecute/cmdGoto and dsExecute/cmdLocate commands, the timecode field tells you the timecode to which Premiere wants you to move the deck. For dsExecute/cmdStatus calls, you return the deck's current timecode via the timecode field, where -1 will display "N/A" (not available), -2 will blank the timecode display, and -3 will display “Searching…”. For dsExecute/cmdJogTo calls, timecode specifies the location to which you should jog the deck.

timeformat

The timeformat field is used to report the format of timecode for a dsExecute/cmdStatus call. The field should be set to 0 for non-drop frame, or 1 for drop-frame.

timerate

The timerate field is used to report the frames-per-second rate of timecode for a dsExecute/cmdStatus call. The field should be set to 24, 25, or 30.

features

The features field is used to report the features that a device and/or device control module is capable of in response to a dsExecute/cmdGetFeatures call. See cmdGetFeatures in the Commands section below for more details.

error

The error field is used to report errors that occur within your device control module. Whenever an error occurs, set (*theData)->error to the appropriate error code and return a non-zero value from your device control module.

preroll

The preroll field is used when you get a dsExecute/cmdLocate call. The preroll amount is how far before (smaller timecode) the time specified in timecode you should seek the deck. The preroll value is the product of a calibration sequence the user can perform. See cmdLocate in the Commands section below for more details on how to use the preroll value.

callback

The callback field contains a pointer to a routine that you can call during dsExecute/cmdLocate calls to determine if the user is attempting to abort the locate operation (by hitting ESC for instance). The prototype for the abort callback routine is:

```
typedef long (*CallBackPtr)();
```

A non-zero result indicates that the user has attempted to terminate the locate operation.

PauseProc

The PauseProc field contains a pointer to a routine that you can call to temporarily pause any Video for Windows sequence grabber operations in a device-controlled window. Normally you would call this routine before putting up an error alert, for instance:

```
(*(*theData)->PauseProc)();
myPoseAlert(kErrors, kMemFailure, 0, 0); // your error handler
(*(*theData)->ResumeProc)();
```

Important! *If you don’t call the PauseProc before putting up an error alert (or any other kind of window), video may be played through over your window. That is the purpose of the PauseProc.*

ResumeProc

The ResumeProc field contains a pointer to a routine that you should call to resume sequence grabbing after calling the PauseProc. It is important that every call to PauseProc be matched by a call to ResumeProc.

Commands

When you receive a call with the dsExecute selector, the command field of the DeviceHand tells you what device control command to execute. Here's a list of the commands and their basic functions. A detailed description of each command follows the list:

Table 9–9: Device Control Commands

Command name	Value	Function
cmdGetFeatures	0	Fill in the features field with the device's features.
cmdStatus	1	Return the deck mode and current timecode.
cmdNewMode	2	Change the deck's mode to a new mode.
cmdGoto	3	Go to a particular time code.
cmdLocate	4	Go to a particular time code and return when you're there.
cmdShuttle	5	Shuttle the deck at a specified rate.
cmdJogTo	6	Position the deck quickly to the location in timecode.
cmdJog	7	New in 4.2 – Jog at rate specified in 'mode', from -15 to +25.

cmdGetFeatures

The cmdGetFeatures command tells you to fill out (*theData)->features with the features of your deck (or of your device control module, if the module can only control a subset of the deck's capabilities). The value you set should be made up of the following bit flags:

```
enum {
   fHasJogMode              // New in 4.2 - device has jog capabilities
   fStepFwd = 0x8000,       // Can step forward one frame
   fStepBack = 0x4000,      // Can step back one frame
   fRecord = 0x2000,        // Can record
   fPositionInfo = 0x1000,  // Returns position (timecode) info
   fGoto = 0x0800,          // Can seek to a specific frame
   f1_5 = 0x0400,           // Can play at 1/5 speed
   f1_10 = 0x0200,          // Can play at 1/10 speed
   fBasic = 0x0100,         // Supports Stop, Play, Pause, FF, Rew
   fHasOptions = 0x0080,    // Plug-in puts up an options dialog
   fReversePlay = 0x0040,   // Supports reverse play
   fCanLocate = 0x0020,     // Can locate a specific timecode
   fStillFrame = 0x0010,    // Frame addr-able device like laser disc
   fCanShuttle = 0x0008,    // Supports the Shuttle command
   fCanJog = 0x0004,        // Supports the JogTo command
};
```

New in Premiere 4.2 is the fHasJogMode bit. When set, Premiere will use cmdJog with a rate modifier rather than sending a new timecode to cmdJogTo each time.

The fStepFwd bit indicates that you can step your deck forward one frame. If you set this bit, Premiere will make available a step-forward button, and you may get called to change your mode to modeStepFwd.

The fStepBack bit indicates that you can step your deck backward one frame. If you set this bit, Premiere will make available a step-backward button, and you may get called to change your mode to modeStepBack.

The fRecord bit indicates that your deck can record. If you set this bit, you may get called to change your mode to modeRecord.

The fPositionInfo bit indicates that your deck and device control module can retrieve position information and pass it back to Premiere.

The fGoto bit indicates that your device can seek to a particular frame. If you set this bit, you must also set fPositionInfo, and you must be prepared to get cmdGoto calls.

The f1_5 bit indicates that your device can play at one-fifth speed. If you set this bit Premiere makes available the 1/5 speed playback option and you may get called to change your mode to modePlay1_5.

The f1_10 bit indicates that your device can play at one-tenth speed. If you set this bit Premiere makes available the 1/10 speed playback option and you may get called to change your mode to modePlay1_10.

The fBasic bit indicates your deck and perform the basic five deck control operations: stop, play, pause, fast-forward, and rewind. If you set this bit, Premiere makes available controls for these functions and you must be prepared to get called to change your mode to modeStop, modePlay, modePause, modeFastFwd, and modeRewind, respectively.

The fHasOptions bit indicates that your device control module has an options dialog, and that you support the dsSetup message. If you set this bit, Premiere makes available the “Options…” button in the Device Control Preferences dialog box. If the user clicks this button, your device control module will get a dsSetup call.

The fReversePlay bit indicates that your deck can play in reverse. If you set this bit, you may bet calls to change your mode to modePlayRev, (and also modePlayRev1_5 and modePlayRev1_10 if you set the f1_5 or f1_10 bits).

The fCanLocate bit indicates that your deck can accurately locate a particular timecode and supports the cmdLocate command. Adobe encourages developers of device control modules to support cmdLocate, which is typically more accurate than cmdGoto.

The fStillFrame bit indicates that your device is frame-addressable, like a laser disk player, and that it is capable of very clean still-frame output. This bit is currently not used by the Movie Capture, Step Capture, or Waveform Monitor windows.

The fCanShuttle bit indicates that your device is capable of variable-speed shuttle operations, both forward and backwards. If you set this bit, Premiere may make cmdShuttle calls to your device control module.

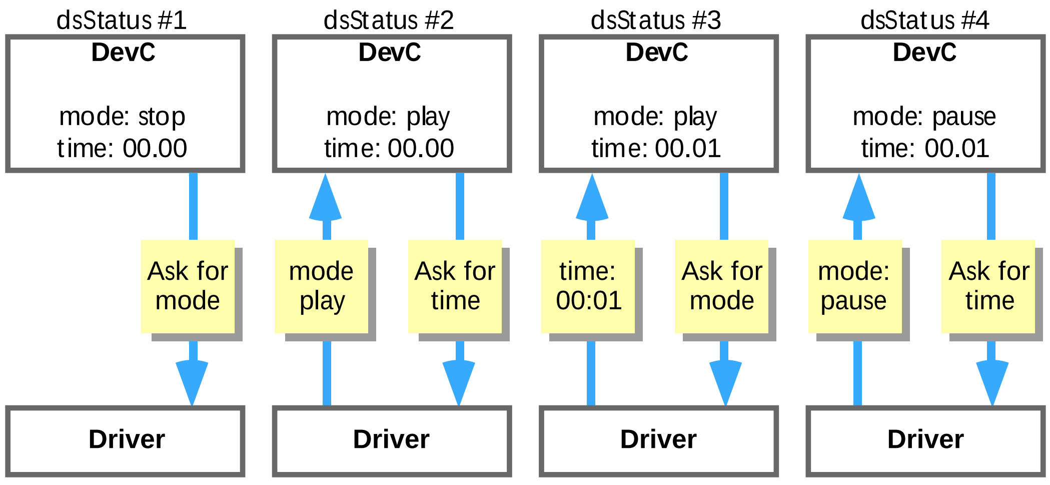
The fCanJog bit indicates that your device can quickly move to a nearby timecode location. If you set this bit, Premiere may make cmdJogTo calls to your module.

cmdStatus

The cmdStatus command is Premiere’s way of finding out what’s going on with your device. It wants two pieces of information: the deck’s current mode (like play, pause, etc.) and the timecode currently rolling under the deck’s head.

You should store the device's current mode into (*theData)->mode, and store the current timecode value into (*theData)->timecode. Be sure to set (*theData)->timerate and (*theData)->timeformat as described in The DeviceRec Structure above.

The values of mode and timecode persist from one cmdStatus call to the next. So, if you only know one of the two pieces of information, store the one you know, and leave the other alone. For instance, it may be that your device control module has to make two separate driver calls to determine these two pieces of information. In that case, you should alternately request one and return the other, as shown in the figure below:



cmdNewMode

The cmdNewMode command tells you to put your device into a new operating mode, as specified in (*theData)->mode. The modes you may be asked to go into (depending upon your features) follow. They have corresponding meanings to the features described above.

```
enum
{
   modeStop = 0,
   modePlay,
   modePlay1_5,
   modePlay1_10,
   modePause,
   modeFastFwd,
   modeRewind,
   modeRecord,
   modeGoto,
   modeStepFwd,
   modeStepBack,
   modePlayRev,
   modePlayRev1_5,
   modePlayRev1_10
};
```

cmdGoto

The cmdGoto command tells you to seek your device to the timecode specified by (*theData)->timecode. Subsequently you should place the device in pause mode (if you were able to complete the seek) or stop mode (if there was an error). Typically you will set up some kind of asynchronous seek and return immediately.

Premiere will continue sending cmdStatus requests until the mode changes to cmdPause or cmdStop. While you are seeking you should place the value modeGoto in (*theData)->mode. This will cause Premiere to put "Searching..." in the time code display of the supervising window. Once you've completed the seek, store the new mode (modePause or modeStop) in (*theData)->mode. Note that Premiere prefers cmdLocate (described below) to cmdGoto, and often cmdLocate is easier to implement anyway (because it is synchronous).

cmdLocate

The cmdLocate command tells you to seek you device to an exact frame location and return immediately with the device in modePlay. This is to be done as a synchronous operation (your device control module should not return until the operation is complete or an error occurs).

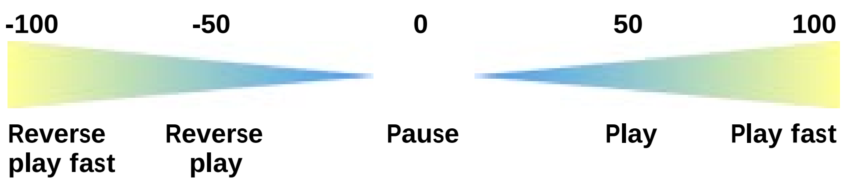
The value in (*theData)->preroll tells you how far before the time specified in (*theData)->timecode you should actually seek to. In other words, subtract (*theData)->preroll from (*theData)->timecode and seek there. The preroll value can be set by the user and is generally determined through a calibration process that takes into account the various latencies of the computer, deck, and device control I/O channel.

During the execution of this command, you can use the abort-check routine (*theData)->callback to determine if the user has attempted to abort the operation (with ESC, for instance).

cmdShuttle

The cmdShuttle command is sent when the user grabs the shuttle control on the screen. The farther the user drags the control from the center, the higher speed he wants the deck to shuttle.

When you receive a cmdShuttle command, (*theData)->mode is the shuttle speed:



If the deck can handle intermediate speeds, you should use them. The idea is to simulate a shuttle control on the front panel of a deck. You may need to map speed values to speeds differently than shown above to get the right feel. If your deck doesn't support continuously variable speeds (which many don't), then quantize the speed. For example, here's how a device control for a Visca device control module might quantize the speed value into the set of available deck play speeds:

```
if (speed <= -90) pb->csParam[4] = kRevScan;
else if (speed <= -70) pb->csParam[4] = kRevFast2;
else if (speed <= -50) pb->csParam[4] = kRevFast1;
else if (speed <= -20) pb->csParam[4] = kRevPlay;
else if (speed <= -12) pb->csParam[4] = kRevSlow1;
else if (speed <= -5) pb->csParam[4] = kRevSlow2;
else if (speed < 5) pb->csParam[4] = kPause;
else if (speed < 12) pb->csParam[4] = kSlow2;
else if (speed < 20) pb->csParam[4] = kSlow1;
else if (speed < 50) pb->csParam[4] = kPlay;
else if (speed < 70) pb->csParam[4] = kFast1;
else if (speed < 90) pb->csParam[4] = kFast2;
else pb->csParam[4] = kScan;
```

To get the right feel, the Visca module places kRevPlay at -20 rather than -50.

If a device control module does not implement shuttling but supports multiple play speeds, Premiere will simulate shuttling by telling the module to play at different rates depending upon the shuttle control position. Of course, better results can be obtained by directly supporting shuttling with the cmdShuttle command.

cmdJogTo

The cmdJogTo command is sent when the user grabs the tractor tread control on the screen. Premiere calculates a new target timecode based on the current time and the distance the user has dragged the tread.

When you receive a cmdJogTo command (*theData)->timecode is the target time code. You should attempt to jog the deck to this location as quickly as possible. (*theData)->timecode will never be far from the current time. How exactly you choose to get the deck to the desired time is up to you—you may choose to step the deck, shuttle, seek, or whatever.

If a device control module does not implement jogging but supports stepping, Premiere will simulate jogging by stepping forward or backward. This does not take into account the distance the user dragged the tractor tread—only the direction. Therefore, better results can be achieved by implementing the cmdJogTo command.

Implementation Tips

Handling dsInit and dsRestart

The dsInit and dsRestart selectors are nearly the same, except that dsInit needs to allocate a new (*theData)->deviceData handle and dsRestart uses one that is provided. Because of this, a handy way of handling these selectors is to let the dsInit selector fall into the dsRestart case, as the code extract from Premiere's Visca module below shows:

```
switch (selector)
{
   case dsInit: // INIT
     if (!((*theData)->deviceData = NewHandleClear(sizeof(LocalRec))))
     {
         // Allocation failed
         result = kMemFailure;
         (*(**theData).PauseProc)();
         myPoseAlert(kErrors, kMemFailure, 0, 0); // your error handler
         (*(**theData).ResumeProc)();
         break;
      }
         // Allocation succeeded so fall through...

   case dsRestart: // RESTART
      // Same as dsInit, except the local data handle has already been
      // allocated and filled in with its contents from the last time.
      // For development purposes, we do a SetHandleSize, in case the
      // definition of the local data has changed.
      if ((*theData)->deviceData &&
         GetHandleSize((Handle)(*theData)->deviceData) != sizeof(LocalRec))
      {
         SafeSetHandleSize((Handle)(*theData)->deviceData, sizeof(LocalRec));
         FillMem(*(*theData)->deviceData, sizeof(LocalRec), 0x00);
      }
      if ((*theData)->deviceData == nil || MemError())
      {
         result = kMemFailure;
         (*(**theData).PauseProc)();
         myPoseAlert(kErrors, kMemFailure, 0, 0); // your error handler
         (*(**theData).ResumeProc)();
         break;
      }
      // Open the driver
      if (result = OpenDriver((StringPtr)"\p.ViSCA",
          &(*(LocalRec **)(*theData)->deviceData)->dRefNum))
      {
```

Notice thatthe code checks whether the (*theData)->deviceData is the same size as LocalRec, the device control module's parameter record. The device control data record is stored by Premiere in the preferences file. If you change the size or layout of your parameter record during development and re-run Premiere, Premiere will kindly pass you a now-invalid deviceData record. That's why the check is there—if the size isn't right, it just reallocates it and zeros the handle.

Putting up error alerts

Remember to frame any error alert routine calls with calls to PauseProc and ResumeProc so that Premiere can suspend any video that might be playing through the supervising window, as shown below:

```
(*(**theData).PauseProc)();
myPoseAlert(kErrors, kMemFailure, 0, 0); // your error handler
(*(**theData).ResumeProc)();
```

Examples

The Adobe Premiere Plug-In SDK comes with source code for a skeleton device control module that you can use as the basis for your own.

10 Other Plug-In Types

Premiere supports several more plug-in types whose descriptions are beyond the scope of this document, they are briefly listed below.

Photoshop Filters

Premiere can load and apply Adobe Photoshop filters to video clips, but there are a few limitations to this. Unlike the Mac version of Premiere 4.2 (which only supports the Photoshop 2.5 API), the Windows version supports the Photoshop 3.0 API. However, there are 2 major, known bugs in Premiere's Photoshop interface.

The first problem is when queried, using AdvanceStateAvailable(), Premiere reports that it supports AdvancedState. However, upon receiving the Start selector, any use of AdvancedState can cause Premiere to crash. A work around is to query the host program, using IsWindows(HostIsPremiere()), and avoid the use of AdvancedState.

The second problem is Premiere reverses the blue and red planes. Again, the work around is to query the host program and if it's Premiere on Windows, reverse the order of the color planes.

It should also be noted that because Photoshop can work with images exceeding the capacity of memory, image data is parceled out to Photoshop filters in a less efficient manner than native Premiere video filters ('VFlt' modules). If you are writing a video-specific filter you will find it is generally easier to write a VFlt than to write a Photoshop filter.

If you do choose to ship Premiere-compatible Photoshop filters (which we certainly encourage), Premiere supports the inclusion of the FltD resource in the Photoshop filter's resource fork. This optional resource describes the filter's data structure such that Premiere can interpolate the filter's settings over time. For more information about the FltD resource, see the FltD section of the Video Filters chapter.

For information on writing Photoshop filters, please refer to the *Adobe Photoshop Plug-Ins Software Development Kit*. It is available from the Adobe Developers Association and is included on the *Adobe Graphics and Publishing SDK* CD-ROM and is also available on the Adobe Web site (www.adobe.com).

The DissolveSans filter included in the Photoshop 4.0 SDK is a good example of a truly cross-application Photoshop filter. It is fully functional in current and older versions of Photoshop and Premiere 4.2, and it includes an FltD resource. It also demonstrates the work arounds to the problems noted above.

Window Handler Modules (‘HDLR’)

Window handler modules are how all the windows in Adobe Premiere are implemented. Most of the windows you see are serviced by handlers in the Adobe Premiere resource file. Others, like Movie Capture and Title are stored in plug-ins. Handlers are first-class entities in Premiere, receiving events and Premiere’s drag-and-drop functionality.

Audio/Video Import Modules (‘Draw’)

Audio/video import modules handle file-format conversion for Adobe Premiere. Draw modules make all types of video and audio media look the same to Premiere internally. New file formats can be supported through the implementation of Draw modules.

Bottleneck Modules (‘Botl’)

Bottleneck modules are like ‘INIT’s for Adobe Premiere. They are loaded and run once at application startup. The main use for a Botl module is the replacement of one of Premiere’s basic bottleneck routines. It is possible to provide hardware acceleration of Premiere through Botl modules.

How to Get More Information

These plug-in types (HDLR, Draw, Botl) are more complex and development requires the disclosure of Adobe proprietary information to the developer. The Adobe Premiere Advanced Plug-In Supplement is available from Adobe only under non-disclosure and by special arrangement. Contact Adobe Developer Relations for more information.