



Adobe Premiere™ version 4.2



Software Development Kit release 3 for Macintosh

Adobe Premiere 4.2 Software Development Kit, release 3

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		
14 September 1994	Bryan K. "Beaker" Ressler	Version 4.0
14 December 1995	Brian Andrews	Version 4.2 - Reformatted and updated for Premiere 4.2
9 February 1996	Brian Andrews	Version 4.2r2 - Minor editorial updates.
13 November 1996	Brian Andrews	Version 4.2r3 - Incremental updates and bug fixes, expanded Photoshop section.

1	Introduction	9
	Windows vs. Macintosh Plug-Ins	9
	How to Use This Guide	9
	About This Guide	10
	What's New	10
	Plug-In Overview	11
	Resources	12
	Native Power Macintosh Plug-Ins	12
	Building Premiere Plug-Ins	13
	Premiere Terminology	14
	Timecode	14
	Things to Remember	15
2	The Utility Library	16
	General Macintosh Routines	16
	Memory Routines	16
	Dialog Routines	17
	Window Routines	26
	String Routines	27
	Graphics Routines	28
	List Manager Lists Routines	33
	Data Lists Routines	34
	File Routines	35
	Menu Routines	36
	Math Routines	37
	Premiere-Specific Routines	38
	Interface Compatibility Routines	38
	Global Accessor Routines	41
	Text, Strings, and Memory Routines	41
	Clip Routines	44
	File Routines	47
	Debugging Routines	50
	Cursor Control Routines	51
	Event Routines	52
	Standard File Routines	53
	Dialog Routines	54
	Window Routines	56
	Graphics Routines	57
	Time Code Routines	59
	Data Export Module Utilities	62
	EDL Export Module Utilities	63

3	Bottlenecks	66
	The BottleRec Structure	66
	The Bottleneck Routines	67
4	Globals	71
	Look But Don't Touch!	71
	Read/Write Globals	71
	Read Only Globals	72
5	CDEFs	78
	Control Hits	78
	Control Types	79
	Horizontal Sliders	79
	Vertical Sliders	80
	Popup Menus	80
6	Transitions	81
	FXvs 1000	82
	TEXT 1000	82
	Fopt 1000	82
	Fopt—first byte: Valid corners	83
	Fopt—second byte: Initial corners	83
	Fopt—third byte: Bit flags	83
	Fopt—fourth byte: Exclusive flag	84
	Fopt—fifth byte: Reversible	84
	Fopt—sixth byte: Has edges flag	84
	Fopt—seventh byte: Movable start point flag	85
	Fopt—eighth byte: Movable end point flag	85
	FXDF -1	85
	SPFX/SPFx 1000	87
	esExecute	87
	esSetup	88
	The EffectRecord Structure	88
	specsHandle	88
	source1	88
	source2	88
	destination	88
	part	89
	total	89
	previewing	89
	arrowFlags	89

- reverse 89
- source 89
- start 89
- end 89
- center 89
- privateData 90
- callback 90
- bottleNecks 90
- version 90
- sizeFlags 90
- flags 91
- fps 91
- Examples..... 91**
- Additive Dissolve 91
- Cross Zoom 91
- Wipe 91

7 Video Filters 92

- FXvs 1000 92
- FltD 1000..... 92
- VFit /VFIT 1000 94
- fsExecute 94
- fsSetup 94
- fsDisposeData 95
- The VideoRecord Structure 95**
- specsHandle 95
- source 95
- destination 95
- part 95
- total 95
- previewing 96
- privateData 96
- callback 96
- bottleNecks 96
- version 96
- sizeFlags 96
- flags 96
- fps 97
- InstanceData 97
- Examples..... 97**
- Video Noise 97
- Burn Time Code 97

8	Audio Filters	98
	FXvs 1000	98
	FltD 1000	98
	AFit/AFIT 1000	99
	fsExecute	99
	fsSetup	99
	fsDisposeData	99
	The AudioRecord Structure	100
	specsHandle	100
	source	100
	destination	100
	sampleNum	100
	sampleCount	101
	previewing	101
	privateData	101
	callback	101
	totalsamples	101
	flags	101
	rate	102
	bottleNecks	102
	version	102
	extraFlags	102
	fps	102
	InstanceData	102
	Examples	102
	Backwards [Audio]	102
	Pan	102
9	Data Export Modules	103
	FXvs 1000	103
	FLAG 1000	104
	ExpD/Expd 1000	104
	edExecute	104
	The DataExportRec Structure	104
	markers	105
	numframes	105
	framerate	105
	bounds	105
	audflags	105
	audrate	105
	getVideo	105
	getAudio	106
	privateData	107

specialRate	107
Relevant Routines in the Utility Library	107
Examples	107
Flattened Movie	107
Storyboard Image	107
10 EDL Export Modules	109
FXvs 1000	109
ExpM/ExpM 1000	110
exExecute	110
exTrue30fps	110
The ExportRecord Structure	110
dataHandle	110
timeBase	110
projectName	111
The EDL Project Data Format	111
Wipe Code Details	114
Relevant Routines in the Utility Library	115
Examples	116
Generic EDL	116
11 Zoom Modules	117
Zoom/Zoom 1000	117
cmdCanZoom	118
cmdZoomIn	118
cmdZoomOut	118
cmdCanDo	118
cmdGetSupportedModes	118
cmdGetMode	119
cmdSetMode	119
The ZoomRec Structure	119
theDevice	119
boardID	119
zoomData	119
mode	119
Other Details	120
Examples	120
Video - SuperMac	120
12 Device Control Modules	121
DevC/DevC 1000	121
dslnit	122

dsSetup	122
dsExecute	122
dsCleanup	122
dsRestart	122
dsQuiet	122
The DeviceRec Structure	123
deviceData	123
command	123
mode	123
timecode	123
timeformat	123
timerate	124
features	124
error	124
preroll	124
callback	124
PauseProc	124
ResumeProc	124
Commands	125
cmdGetFeatures	125
cmdStatus	127
cmdNewMode	127
cmdGoto	128
cmdLocate	128
cmdShuttle	128
cmdJogTo	129
Implementation Tips	129
Handling dslnit and dsRestart	129
Putting up error alerts	130
Examples.	130
Device	131
13 Other Plug-In Types	132
Photoshop Filters	132
Window Handler Modules ('HDLR')	132
Audio/Video Import Modules ('Draw').	133
Bottleneck Modules ('Botl').	133
How to Get More Information	133
Index	134

1 Introduction

Welcome to the Adobe Premiere™ 4.2 Software Developers Toolkit for Macintosh!

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 Metrowerks CodeWarrior 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 Macintosh version of the Premiere SDK, there is another version of the entire SDK (including this documentation) available for Windows 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.

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 both general Macintosh and Premiere specific calls.

Chapter 3 describes the [Bottlenecks](#), which are a set of procedures and structures to perform common operations.

Chapter 4 describes Premiere's [Globals](#) which can be examined by a plug-in.

Chapter 5 describes a set of [CDEFs](#), or control definition procedures, that a plug-in can use to help achieve a consistent look and feel with the rest of Premiere.

Chapter 6 on [Transitions](#), is the first chapter on specific plug-in types. Transitions take two GWorlds and processes them into a single destination GWorld, usually applying some special transition effect.

Chapter 7 describes [Video Filters](#), which take a single GWorld and processes it into a destination GWorld, usually applying a visual effect.

Chapter 8 describes [Audio Filters](#), which take a single source buffer of audio and processes it into a destination buffer, usually applying an audio effect.

Chapter 9 describes [Data Export Modules](#). These appear in Premiere's Export submenu and export a given clip to some other format.

Chapter 10 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 11 describes [Zoom Modules](#). Zoom modules handle hardware-specific details of zooming and video card mode-switching.

Chapter 12 describes [Device Control Modules](#). These allow Premiere to control hardware devices such as tape decks or laser disc players.

Finally, Chapter 13 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](#).

If you're programming transitions or filters, you may need sliders or other special controls. In that case, look at the [CDEFs](#) chapter for important information about proclIDs and calling conventions.

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 Macintosh, version 4.2. The libraries and headers files have been updated since the last release with the 4.2 changes commented in the header files.
- Supports development using Metrowerks CodeWarrior 10. While the example plug-ins should still build using MPW, the emphasis has been on converting the previous examples to the new CodeWarrior environment and making use of some CodeWarrior specific features (such as precompiled headers) to speed development.
- Improved documentation. We hope you'll find this new document format more readable for both on screen and printed viewing.
- There have been some improvements to device control which you'll find described in chapter 12, [Device Control Modules](#), and also in the header files. Several new feature bits, codes, and commands have been added.
- Several new globals are available for inspection, including one which aids in error detection. You find these described in [Globals](#), chapter 4, and in the header files.
- 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](#), chapter 7, [Audio Filters](#), chapter 8 and the header files for information on this.
- Premiere can load and apply Adobe Photoshop filters to video clips. Chapter 13 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 code resource of a type specific to the purpose of the plug-in. Each plug-in can have private resources in its plug-in file.

Table 0-1: Overview of Adobe Premiere Plug-In Modules

Type	Name	Description
'SPFX'	Video transition	Create a C video frame from A and B frames
'VFit'	Video filter	Modify ("filter") one frame of video
'AFit'	Audio filter	Modify ("filter") one audio "blip"
'ExpD'	Data export module	Export video or audio from a clip
'ExpM'	EDL export module	Export construction window information
'ZooM'	Zooming module	Perform hardware zooming on a video card
'DevC'	Device control module	Control a hardware device like a tape deck
'Draw'	A/V import module	Import audio or video from a file
'HDLR'	Window module	A complete functioning window, like Title
'BotI'	Bottleneck module	Accelerate Premiere by overriding bottlenecks
'8BFM'	Photoshop filter	Apply a Photoshop filter to one frame of video

The last three types of plug-ins, 'Draw', 'HDLR', and 'Botl', are documented separately in the Adobe Premiere Plug-In Toolkit Supplement, also known as the Advanced SDK, which is available only under non-disclosure and by special request from Adobe. See the [Other Plug-Ins](#) chapter at the end of this document for more information.

Adobe Premiere 4.2 requires a Macintosh with a 68020 or later processor or a Power Macintosh. Premiere runs under System 7.5 or later, and requires QuickTime 2.1 and 32-bit QuickDraw. Your plug-ins can assume the presence of these software components.

Resources

Plug-in modules reside in their own resource file located in Premiere's Plug-Ins folder. When a plug-in is called to perform its function, the current resource file is set to the plug-in's resource 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 blob" that gets saved by Premiere in a Premiere project file, then later that blob 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, they can be stored in the plug-in's resource fork.

Native Power Macintosh Plug-Ins

Adobe encourages you to create "fat" (68K + PPC) Premiere plug-ins as Adobe Premiere 4.2 is a native Power Macintosh application. All the plug-in modules for Premiere are also PPC native, but Premiere can still run 68K plug-ins on a Power Mac.

When Premiere is running on a Power Mac and it calls a plug-in, it looks first for a Power Mac resource (Table 1-2, below). If no PPC code resource is present, it loads the 68K code resource and calls it through the 68K emulator.

Table 0-2: Alternate Resource Types for Power Mac

Power Mac	68K Mac
'SPFx'	'SPFX'
'VFIT'	'VFit'
'AFIT'	'AFit'
'Expd'	'ExpD'
'ExpM'	'ExpM'
'Zoom'	'ZooM'
'DevC'	'DevC'
'DraW'	'Draw'
'HDLr'	'HDLR'
'BotL'	'Botl'
'8BFm'	'8BFM'

Note that the 68K and PPC resource types are the same except for the capitalization of the final letter.

The PPC code resource for a plug-in must contain a straight resource PEF with no routine descriptor or glue code at the beginning. That is, if you derez the PPC resource, you should see something like this:

```
data 'VF1T' (1000) {
$"4A6F 7921 7065 6666 7077 7063 0000 0001" /* Joy!peffpwpc.... */
$"AA40 C123 0000 0000 0000 0000 0000 0000" /* "i#..... */
$"0003 0002 0000 0000 FFFF FFFF 0000 0000" /* .....^... */
.
.
.
};
```

Note that some development environments might place some standard defProc-type 68K glue code at the beginning of PPC code you place in a resource. The code is roughly equivalent to the code in MPW's MixedMode.r file for the description of the 'sdes' "safe fat resource." ***This won't work!*** You'll need to strip or prevent this glue code if you intend to use these environments for Premiere plug-in development.

Building Premiere Plug-Ins

This section contains information on building Premiere plug-ins with MPW or Metrowerks CodeWarrior. All examples in this SDK are built using one or both of these development environments.

MPW

Here are the basic MPW commands you'd use to build a "fat" Adobe Premiere video filter:

```
# Build the 68K resource
c -r -b2 -mc68020 -o MyFiler.c.o MyFilter.c
link -rt VF1t=1000 -m XFILTTER -t VF1t -c PrMr -o MyFilter  
MyFilter.c.o UtilLib.o

# Build the PowerPC resource
ppcc -d PrPPC -align mac68k -appleext on MyFilter.c
ppclink -mf -main xFilter -o MyFilter.xcoeff  
MyFilter.c.o  
ProcStubs.xcoeff  
{PPCLibraries}QuickTimeLib.xcoeff  
{PPCLibraries}DragLib  
{PPCLibraries}StdCRuntime.o  
{PPCLibraries}StdCLib.xcoeff  
{PPCLibraries}InterfaceLib.xcoeff  
{PPCLibraries}PPCCRuntime.o  
{PPCLibraries}MathLib.xcoeff
makepef -b -o MyFilter.pef MyFilter.xcoeff  
-l QuickTimeLib.xcoeff=QuickTimeLib~  
-l DragLib.xcoeff=DragLib~  
-l MathLib.xcoeff=MathLib  
-l StdCLib.xcoeff=StdCLib  
-l InterfaceLib.xcoeff=InterfaceLib  
-l ProcStubs.xcoeff=AdobePremiere
echo "read  VF1T ' (1000)  MyFiler.pef ";" |  
rez -a -t VF1t -c PrMr -o MyFilter

# Rez the filter's resources
rez -a -o MyFilter MyFilter.r
```

Metrowerks CodeWarrior

Starting with version 4.2 of the Adobe Premiere Software Developers Toolkit, the Metrowerks CodeWarrior environment is supported. The 12 example plug-ins supplied with this toolkit all include CodeWarrior 10 project files for building fat plug-ins. By looking at the project preferences, you should be able to see how to set up your own projects. A few things to note are:

- Since Adobe Premiere and the toolkit libraries were built using MPW, you need to use MPW C calling conventions when making the calls described in this document. The example projects all have this option set globally in the project preferences.
- Included with the examples are library files for linking the 68k and PPC plug-ins with both CodeWarrior and MPW. You'll be able to tell which to use by looking at the sample projects and *.lib files* folder.
- The *.h files* folder contains all the header files. To greatly speed compilation, the header files have been precompiled for use by CodeWarrior. The examples make use of this, so if you modify any of Premiere's header files, don't forget to precompile them. The *.pch* files contained in *.h files* will aid you in this.

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

Clips 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. You'll see some utility routines documented below that return or operate on clip IDs.

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**. Adobe provides some utility routines that deal with file IDs.

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.

- Unless stated otherwise, strings in Premiere are always Pascal strings (that is, a length byte followed by that many bytes of text).
- Use `UserItem`, `ShowModal`, and `DisposeModal` for modal dialogs (see the chapter [The Utility Library](#) for details).
- Whenever one's available, use the "Pr-" versions of a routine. For instance, use `PrModalDialog` instead of `ModalDialog`. See the category Interface Compatibility in the Premiere-Specific Routines section of the chapter [The Utility Library](#).
- Use `PrDebug` to send formatted strings to the Debug window in Premiere to help you debug your plug-ins. To bring up the Debug window, press Control-Option-0 (zero).
- 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 SPFX, VFit, and AFit), 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.

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. 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 routines and Premiere-specific routines. Within those categories, the routines are divided into functional categories.

General Macintosh Routines

Memory Routines

LockHHi: Move a handle high in the heap and lock it.

```
void LockHHi (void *h);
```

This routine moves the given handle *h* high in the heap with a call to `MoveHHi`, then locks it with `HLock`. Note that while *h* is declared as a `void *`, it must be a handle.

SafeSetHandleSize: Premiere's more effective version of `SetHandleSize`.

```
short SafeSetHandleSize (  
    void *h,  
    long size );
```

This routine attempts to set the size of handle *h* to the new byte size *size*, just like the Mac Toolbox routine `SetHandleSize`. `SafeSetHandleSize`, however, is willing to reallocate and copy the handle, which `SetHandleSize` is not. Therefore, `SafeSetHandleSize` will sometimes succeed when `SetHandleSize` fails. We recommend using `SafeSetHandleSize` instead of `SetHandleSize` within Premiere plug-ins. Note that while *h* is declared as a `void *`, it must be a handle.

FillMem: Set an area of memory to a specified byte value.

```
void FillMem (  
    void *dest,  
    long count,  
    char value );
```

This routine sets *count* bytes starting at *dest* to the byte value *value*. It is a handy way to clear a structure to all zeros or fill an area of memory.

StructsSame: Compares two structures for an exact match.

```
char StructsSame (
    void *a,
    void *b,
    long size );
```

This routine compares the structure pointed to by pointers a and b for size bytes. If they are exactly byte-for-byte the same, it returns true, otherwise it returns false. Be careful about using StructsSame to compare structures containing strings (like FSSpecs, for instance) because dead bytes at the ends of the strings will make StructsSame return false when the structures are functionally identical.

Dialog Routines

EnableDItem: Enable a dialog item.

```
void EnableDItem (
    DialogPtr theDialog,
    short item );
```

This routine enables item number item in dialog theDialog by setting the enabled bit in the item's item type.

DisableDItem: Disable a dialog item.

```
void DisableDItem (
    DialogPtr theDialog,
    short item );
```

This routine disables item number item in dialog theDialog by clearing the enabled bit in the item's item type.

HiliteDControl: Set a control in a dialog to a specific highlight value.

```
void HiliteDControl (
    DialogPtr theDialog,
    short item,
    short value );
```

This routine gets the item handle for item number item from dialog theDialog, which must be a control-type item. It then calls HiliteControl to set that control's highlighting to value.

GetCValue: Get the value of a control in a dialog.

```
short GetCValue (
    DialogPtr theDialog,
    short item );
```

This routine gets the item handle for item number item from dialog theDialog, which must be a control-type item. It then returns GetCtlValue for that control. This routine is convenient for retrieving the values of such controls as check boxes, radio buttons, and scroll bars in dialogs.

SetCValue: Set the value of a control in a dialog.

```
void SetCValue (
    DialogPtr theDialog,
    short item,
    short value );
```

This routine gets the item handle for item number `item` from dialog `theDialog`, which must be a control-type item. It then calls `SetCtlValue` to set the value of that control to `value`. This routine is convenient for setting the values of such controls as check boxes, radio buttons, and scroll bars in dialogs.

GetGroupVal: Return the number of the set control in a radio button group.

```
short GetGroupVal (
    DialogPtr theDialog,
    short first,
    short last );
```

This routine looks in dialog `theDialog` for the first “set” radio button between item number `first` and item number `last` (inclusive) and returns the relative value. So, if `first` is set, `GetGroupVal` returns 0. If `last` is set, `GetGroupVal` returns `last - first`.

DrawDItem: Force-redraw a single dialog item.

```
void DrawDItem (
    DialogPtr theDialog,
    short item );
```

This routine redraws a item number `item` in dialog `theDialog` without redrawing any other items. It saves the clipping for the dialog, sets the clipping to the single specified item, calls `UpdtDialog` to redraw the item, `ValidRgn` to validate the item, then restores the dialog’s clipping.

GetDRect: Get the rectangle of a dialog item.

```
void GetDRect (
    DialogPtr theDialog,
    short item,
    Rect *box );
```

This routine returns the rectangle of item number `item` from dialog `theDialog` via the reference parameter `box`.

SetDRect: Set the rectangle of a dialog item.

```
void SetDRect (
    WindowPtr thewindow,
    short item,
    Rect *box );
```

This routine sets the rectangle of item number `item` from dialog `theDialog` via to the rectangle pointed to by `box`.

InvalItem: Invalidate a dialog item.

```
void InvalItem (
    DialogPtr theDialog,
    short item );
```

This routine invalidates the item rectangle for item number `item` in dialog `theDialog`. It does not force-redraw the item. It will be redrawn the next time `ModalDialog` or `DialogSelect` is called.

GetDType: Returns a dialog item's type.

```
void GetDType (
    DialogPtr theDialog,
    short item,
    short *itemType );
```

This routine returns the item type of item number `item` from dialog `theDialog` via the reference parameter `itemType`. Note that the Dialog Manager stores some bit-flags in the item type, so you generally cannot do checks like `(itemType == iconItem)`. See IM-I p404.

GetDHandle: Returns a dialog item's item handle.

```
void GetDHandle (
    DialogPtr theDialog,
    short item,
    Handle *itemHand );
```

This routine returns the item handle of item number `item` from dialog `theDialog` via the reference parameter `itemHand`.

ModifyDItem: Sets the item handle and rectangle for a dialog item.

```
void ModifyDItem (
    DialogPtr theDialog,
    short item,
    Handle itemHand,
    Rect *box );
```

This routine sets the item handle and item rectangle to `itemHand` and `box`, respectively, for dialog item number `item` in dialog `theDialog`.

Important! Do not use `ModifyDItem` to install user item procedures or your code will not run correctly on the PowerPC. For that purpose use the `UserItem` function described below.

UserItem: Sets the item handle and rectangle for a dialog item.

```
void UserItem (
    DialogPtr theDialog,
    short item,
    UserItemProcPtr theProc );
```

This routine sets the item handle to `theProc` for dialog item number `item` in dialog `theDialog`. This is a convenient way to install user item procedures into dialog items.

Important! On the PowerPC this function plays a key role. When you use `UserItem` to install a user item procedure on the PowerPC, `UserItem` allocates an appropriate UPP and

passes it to the Dialog Manager. The Premiere DisposeModal function iterates through the dialog's item list and frees these UPPs before disposing the dialog. For this reason it is of utmost importance to use UserItem and no other function to install user item procedures in your dialogs. See also the functions SetCAction (below), and ShowModal and DisposeModal in the Dialog Routines category of the Premiere Specific Routines section in this chapter.

FlashControl: Flashes a dialog button item as if it were clicked.

```
void FlashControl (
    DialogPtr theDialog,
    short item );
```

This routine flips item number item from dialog theDialog (which must be a control-type item), to its "highlighted" state, holds it that way for 15 ticks (1/4 second), then flips it back. Normally this is used for providing visual feedback to the user when a keyboard-equivalent for a button has been used.

FrameGrayButton: Frame the default button in a dialog, perhaps in gray.

```
void FrameGrayButton (
    DialogPtr theDialog,
    short item,
    char gray );
```

This routine draws the three-pixel-wide bold outline around the item number item in dialog theDialog. If the parameter gray is true, the bold outline is drawn in gray. If the parameter gray is false, the outline is drawn in black. This routine is normally used to draw the bold outline around the default button in a dialog.

FrameButton: Frame the default button in a dialog with a black bold outline.

```
void FrameButton (
    DialogPtr theDialog,
    short item );
```

This routine draws the three-pixel-wide bold outline around the item number item in dialog theDialog. The outline is drawn in black.

ButtonFrame: A user item procedure to draw the default button frame around a button.

```
pascal void ButtonFrame (
    DialogPtr theDialog,
    short item );
```

This routine, when installed as a user item procedure, draws a 3-pixel-thick rounded rectangle within item's rectangle. Typically the user item is placed centered over a button and is exactly four pixels larger than the button in all directions.

PositionDialog: Position a dialog appropriately on the main screen.

```
void PositionDialog (
    short theID,
    Point *where );
```

This routine prepositions a dialog with DLOG resource ID theID appropriately on the main screen. It returns the upper-left point of the dialog in global coordinates via the reference parameter where. Call this routine before calling GetNewDialog.

Validate: Validate the numeric value of a dialog editText item.

```
char Validate (
    DialogPtr theDialog,
    short item,
    long min,
    long max );
```

This routine gets the text of editText item item from dialog theDialog and converts the string into a long. It then checks to make sure the item's value is between min and max (inclusive). If the item's value is in range, Validate returns false. If the item's value is not in range, Validate sets the item's text to either min or max (depending upon which direction the item value was out of range), select the item, does a SysBeep, and returns true. A convenient way to use Validate is to put a series of Validate calls in an if statement:

```
if (!Validate(theDialog, kLength, 0, 15) &&
    !Validate(theDialog, kWidth, -10, 10) &&
    !Validate(theDialog, kTime, 0, 100))
{
    // It's okay to dismiss the dialog
}
else
{
    // At least one value is out of range, but validate
    // has already selected it so the user knows which
    // one it is. Just fall back into our dialog loop
}
```

SetIVal: Set the value of an editText item to a numeric value.

```
void SetIVal (
    DialogPtr theDialog,
    short item,
    long val );
```

This routine converts the parameter val to a string and sets the text of item number item in dialog theDialog to that string. The conversion is done with a call to NumToString, so the range is -2,147,483,649 to 2,147,483,648.

GetIVal: Get the value of an editText item containing a numeric value.

```
long GetIVal (
    DialogPtr theDialog,
    short item );
```

This routine converts the text from editText item number item in dialog theDialog to an integer and returns that value. Conversion is done with a call to StringToNum, so the range is -2,147,483,649 to 2,147,483,648.

SetEText: Set the text of an editText item.

```
void SetEText (
    DialogPtr theDialog,
    short item,
    StringPtr str );
```

This routine sets the text of the editText item number item in dialog theDialog to the text given in parameter str. Note that str must be a Pascal string.

GetEText: Get the text of an editText item into a string.

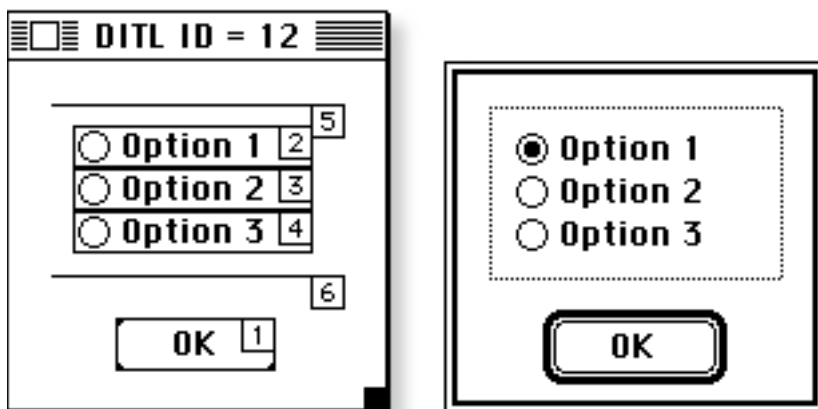
```
void GetEText (
    DialogPtr theDialog,
    short item,
    StringPtr str );
```

This routine gets the text of the editText item number item in dialog theDialog into the string pointed to by parameter str. Note that str is returned as a Pascal string.

DrawItemBox: A user item procedure to draw a rectangular frame in a dialog.

```
pascal void DrawItemBox (
    DialogPtr theDialog,
    short item );
```

This routine, when installed as a user item procedure, draws a box from the bottom of the previous item's rectangle to the bottom of item's rectangle, with the width of item's rectangle. Typically, you use two user items to specify the frame:



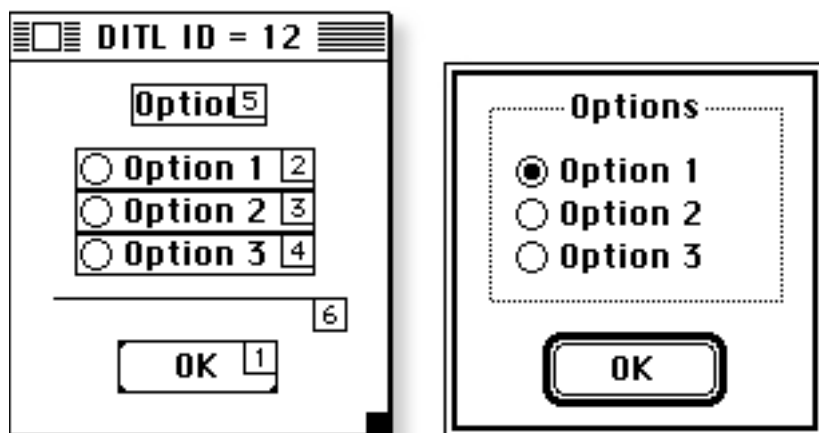
In the dialog shown above, items 5 and 6 are user items. Item 5 has no procedure (it doesn't draw at all), and item 6 has the DrawItemBox procedure attached to it. The right hand picture shows the results.

DrawItemFrame: A user item procedure to draw a group frame in a dialog.

```
pascal void DrawItemFrame (
    DialogPtr theDialog,
    short item );
```

This routine, when installed as a user item procedure, draws a box from the middle of the previous item's rectangle to the bottom of item's rectangle, with the width of item's rectangle. Typically, you use two user

items to specify the frame where item - 1 is a statText item (the group title), and item is a user item:



In the dialog shown above, item 5 is a statText item containing the group title "Options." Item 6 has the DrawItemFrame procedure attached to it. The right hand picture shows the results. As you can see, the DrawItemFrame procedure avoids drawing the frame over the group title text.

SetCMax: Set the maximum value of a control in a dialog.

```
void SetCMax (
    DialogPtr theDialog,
    short item,
    short value );
```

This routine gets the item handle for item number item from dialog theDialog, which must be a control-type item. It then calls SetCtlMax to set that control's maximum value to value.

GetCMax: Get the maximum value of a control in a dialog.

```
short GetCMax (
    DialogPtr theDialog,
    short item );
```

This routine gets the item handle for item number item from dialog theDialog, which must be a control-type item. It then returns GetCtlMax for that control.

SetCRef: Set the refCon value of a control in a dialog.

```
void SetCRef (
    DialogPtr theDialog,
    short item,
    long value );
```

This routine gets the item handle for item number item from dialog theDialog, which must be a control-type item. It then calls SetCRefCon to set that control's refCon to value.

GetCRef: Get the refCon value of a control in a dialog.

```
long GetCRef (
    DialogPtr theDialog,
    short item );
```

This routine gets the item handle for item number `item` from dialog `theDialog`, which must be a control-type item. It then returns `GetCRefCon` for that control.

SetCAction: Get the action procedure pointer of a control in a dialog.

```
void SetCAction (
    DialogPtr theDialog,
    short item,
    ProcPtr theProc );
```

This routine gets the item handle for item number `item` from dialog `theDialog`, which must be a control-type item. It then calls `SetCtlAction` to set the action procedure for that control to `theProc`.

Important! *On the PowerPC this function plays a key role. When you use `SetCAction` to install an action procedure on the PowerPC, `SetCAction` allocates an appropriate UPP and passes it to the Control Manager. The `Premiere DisposeModal` function iterates through the dialog's control list and frees these UPPs before disposing the dialog. For this reason it is of utmost importance to use `SetCAction` to install control action procedures in your dialogs. See also the functions `UserItem` (above), and `ShowModal` and `DisposeModal` in the *Dialog Routines* category of the *Premiere Specific Routines* section in this chapter.*

SetResCTitle: Get the title of a control in a dialog from a STR# resource.

```
void SetResCTitle (
    DialogPtr theDialog,
    short item,
    short resID,
    short strNum );
```

This routine gets the item handle for item number `item` from dialog `theDialog`, which must be a control-type item. It then retrieves a string number `strNum` from the STR# resource with ID `resID`. Finally, it calls `SetCTitle` to set the title for that control to the retrieved string.

OffsetCSize: Add values to the size of an item in a dialog.

```
void OffsetCSize (
    DialogPtr theDialog,
    short item,
    short h,
    short v );
```

This routine gets the item handle for item number `item` from dialog `theDialog`. It retrieves the dialog item rectangle for the item, adds `v` to its bottom and `h` to its right, then (for control-type items) calls `SizeControl` to set the size of the control. Finally, it sets the dialog item rectangle to the newly calculated rectangle. Normally this routine is called for control-type items, but it works for any dialog item.

OffsetControl: Move an item in a dialog by a relative amount.

```
void OffsetControl (
    DialogPtr theDialog,
    short item,
    short h,
    short v );
```

This routine gets the item handle for item number `item` from dialog `theDialog`. It retrieves the dialog item rectangle for the item and offsets it by `v` vertically and `h` horizontally, then (for control-type items) calls `MoveControl` to move the associated control. Finally, it sets the dialog item rectangle to the newly calculated rectangle. Normally this routine is called for control-type items, but it works for any dialog item.

DrawAControl: Draw a single control item in a dialog.

```
void DrawAControl (
    DialogPtr theDialog,
    short item );
```

This routine gets the item handle for item number `item` from dialog `theDialog`, which must be a control-type item. It then calls `Draw1Control` to redraw that control without redrawing any other items in the dialog.

SafeDrawControl: Draw part or all of a control item in a dialog safely.

```
void SafeDrawControl (
    DialogPtr theDialog,
    short item,
    short part );
```

This routine allows you to redraw one part (or an entire control) safely from within a control's action procedure. `SafeDrawControl` retrieves the item handle for item number `item` in dialog `theDialog`, which must be a control-type item. It then redraws part code `part`, which is one of the control's part codes, or 0 for the entire control. `SafeDrawControl` is normally called from `SafeSetCValue` (described below), so normally you won't need to call it yourself.

SafeSetCValue: Set the value of control item in a dialog safely.

```
void SafeSetCValue (
    DialogPtr theDialog,
    short item,
    short value );
```

This routine allows you to set the value of a control (and thereby redraw all or part of it) safely from within a control's action procedure. `SafeSetCValue` retrieves the item handle for item number `item` in dialog `theDialog`, which must be a control-type item. It then pins your new control value `value` within the minimum and maximum values stored in the control and store the new value in the control. Finally, it calls `SafeDrawControl` (described above) to redraw then "inThumb" part of the control. You would use this, for instance, in a Premiere filter dialog where there are two sliders and one slider must set the value of another slider.

Window Routines

CenterWin2Win: Center one window over another window.

```
void CenterWin2Win (
    WindowPtr moveWindow,
    WindowPtr overWindow );
```

This routine centers the window `moveWindow` over the window `overWindow`. Normally this is called with (at least) `moveWindow` hidden. `CenterWin2Win` is handy when, for instance, you want an options dialog to be visually associated with a specific window on the screen.

CenterWinOffset: Center a window within a specific global rectangle with offsets.

```
void CenterWinOffset (
    WindowPtr theWindow,
    Rect *box,
    short h,
    short v );
```

This routine centers the window `theWindow` within the global-coordinate rectangle `box`, then offsets the window's position by `h` horizontally and `v` vertically. Typically, `box` points to a copy of `GDevice's gdRect`, thereby centering a window in a given `GDevice` with offset. If you don't need the offsets, it's more convenient to use `CenterWindow` (see below). Note: `CenterWinOffset` does not make a local copy of the rectangle referred to by `box`, so don't pass the address of a `gdRect` directly inside a `GDHandle`, but instead copy it to a local `Rect` first.

CenterWindow: Center a window within a specific global rectangle.

```
void CenterWindow (
    WindowPtr theWindow,
    Rect *box );
```

This routine centers the window `theWindow` within the global-coordinate rectangle `box`. Typically, `box` points to a `GDevice's gdRect`, thereby centering a window in a given `GDevice`. `CenterWindow` immediately makes a local copy of the rectangle referred to by `box` (which allows you to pass in the address of an actual `gdRect`, as shown below). `CenterWindow` notices if `*box` is exactly equal to the main device's `gdRect`. If so, it adjusts the top by the height of the menu bar. Also, if `theWindow` refers to a dialog (that is, if `theWindow->windowKind == dialogKind`), it adds another 20 on to the top of `box`. This is because modal dialogs in Premiere are exclusively movable-modal. Here's an example of how you might use `CenterWindow`:

```

short DoMyDialog (void)
{
    DialogPtr myDialog;
    Boolean done = true;

    // Get dialog from a template that is set "not
    // visible." Call CenterWindow to center the dialog
    // on the main device.
    myDialog = GetNewDialog(kMyDLOGID, nil,
                           (WindowPtr)-1);
    CenterWindow(myDialog, &(*GetMainDevice())->gdRect);
    ShowModal(myDialog);

    // Do all the rest of my window stuff
    do {
        .
        .
        .
    } while (!done);

    DisposeModal(myDialog);
}

```

CenterWindowOnMain: Center a window on the main screen.

```
void CenterWindowOnMain (WindowPtr theWindow);
```

This routine centers the window theWindow on the main screen. This call is equivalent to:

```
CenterWindow(theWindow, &(*GetMainDevice())->gdRect);
```

String Routines

Append: Append one Pascal string onto another

```
void Append (
    StringPtr str1,
    StringPtr str2 );
```

This routine appends the Pascal string str2 onto the end of Pascal string str1.

StrCopy: Append one Pascal string onto another

```
void StrCopy (
    StringPtr src,
    StringPtr dst );
```

This routine copies the Pascal string src to the destination buffer dst.

StringsSame: Test whether two strings are byte-for-byte identical.

```
char StringsSame (
    StringPtr str1,
    StringPtr str2 );
```

This routine compares the byte values of every character of Pascal string str1 with those of Pascal string str2 including the length bytes. If any values don't match, StringsSame returns false. If they all match, StringsSame returns true. This routine is not internationally friendly—it is for testing that strings are identical.

TypeToStr: Returns a string from a longword integer constant.

```
void TypeToStr (
    long type,
    StringPtr str );
```

This routine sets the Pascal string referred to by `str1` to a string made up of the four bytes in the longword parameter `type`. For example, `TypeToStr('PrMr', str)` sets the `str` to the four-byte Pascal string "PrMr". `TypeToStr` is convenient for turning `OSTypes` and `ResTypes` into strings for display when necessary.

FindIndString: Find a given string in a STR# resource and return its index.

```
short FindIndString (
    StringPtr str,
    short theID );
```

This routine, given a pointer to a Pascal string `str` and the resource ID of a STR# resource `theID`, returns the index into the STR# string list corresponding to `str` (1 to `n`), or 0 if the string is not found. `FindIndString` uses `StringsSame` for comparison, so the STR# string must exactly match `str` to have its index returned.

Graphics Routines

FrameErase: Frame a rectangle then erase inside the frame.

```
void FrameErase (Rect *box);
```

This routine copies the rectangle referred to by `box`, calls `FrameRect` on it, insets it by one pixel all around, then calls `EraseRect` on the inset rectangle.

SetGray: Set the color and/or pattern to a given white level value.

```
void SetGray (unsigned short value);
```

This routine takes a white level value, where 0x0000 is black and 0xffff is white.

`SetGray` behaves differently depending upon the depth of the port you're drawing into (according to the function `DepthOf` described below). For ports ≥ 4 bits-per-pixel, `SetGray` does an `RGBForeColor` call with an `RGBColor` that has all three channels set to `value`. For ports < 4 bits-per-pixel, `SetGray` chooses from among the standard `QuickDraw` patterns `patBlack`, `patDkGray`, `patGray`, `patLtGray`, or if the value is close to white, then `SetGray` sets the pattern to `patBlack` and does an `RGBForeColor` with a white `RGBColor`. `SetGray` is a convenient, pseudo-port-depth-independent way of specifying a gray shade that is used extensively within `Premiere`.

SetBackGray: Set the color and/or pattern to a given white level value.

```
void SetBackGray (unsigned short value);
```

This routine makes an `RGBColor` that has all three channels set to white level value. Regardless of port depth, it then makes an `RGBBackColor` call to set the background color the specified gray level.

SetColor: Set the color and/or pattern to given RGB values.

```
void SetColor (
    unsigned short red,
    unsigned short green,
    unsigned short blue );
```

This routine takes three channel levels red, green, and blue. SetColor behaves differently depending upon the depth of the port you're drawing into (according to the function DepthOf described below). For ports ≥ 4 bits-per-pixel, SetColor does an RGBForeColor call with an RGBColor that has the channels set from the parameters. For ports < 4 bits-per-pixel, SetColor acts according to the following table:

Table 2-1: SetColor Actions

Red	Green	Blue	Action
0	0xffff	0	green, so do a SetGray(0xffff)
0xffff	0	0	red, so do a SetGray(0)
0xffff	0xffff	0	yellow, so do a SetGray(0x7fff)
0xffff	0	0xffff	magenta, so do a SetGray (0x7fff)

If the values don't match any of the table entries, then if any of the channels are $\geq 0x7fff$, it does a SetGray(0xffff), otherwise a SetGray(0). The table handles some special cases that Premiere needs to be readable on black-and-white screens. More often than not, you'll be drawing to a color screen and your RGB values will be used directly.

SetBackColor: Set the background color to given RGB values.

```
void SetBackColor (
    unsigned short red,
    unsigned short green,
    unsigned short blue );
```

This routine takes three channel levels red, green, and blue. SetBackColor behaves differently depending upon the depth of the port you're drawing into (according to the function DepthOf described below). For ports ≥ 4 bits-per-pixel, SetColor does an RGBBackColor call with an RGBColor that has the channels set from the parameters. For ports < 4 bits-per-pixel, SetBackColor looks for any channel value $\geq 0x3000$, then it sets the background color to white, otherwise black.

SetColorFace: Set the text face depending upon the depth of the port.

```
void SetColorFace (short face);
```

This routine takes a text face face. If the depth of the current port is ≥ 4 bits-per-pixel, then SetColorFace simply sets the text face to face with a TextFace call. If the port is < 4 bits-per-pixel, it does a TextFace(bold) and ignores face. You might use this routine instead of TextFace in code that might end up drawing 9-point Geneva on a 50% gray pattern. This routine will substitute bold in that case, and then you'll at least be able to read the text in black-and-white.

- EraseGrow:** Erase the grow box in a window.
- ```
void EraseGrow (WindowPtr theWindow);
```
- This routine erases the grow box in the bottom right corner of theWindow.
- DepthOf:** Makes an educated guess at the depth of a port.
- ```
short DepthOf (GrafPtr thePort);
```
- This routine first checks if thePort is a old-style GrafPort. If so, it returns 1 (meaning 1 bit-per-pixel). If thePort is a CGrafPort, DepthOf returns the depth of the first GDevice that intersects thePort.
- pt2GDevice:** Returns the GDevice on which a global point lies.
- ```
char pt2GDevice (
 Point thePoint,
 GDHandle *theDevice);
```
- This routine returns, via the reference parameter theDevice, the GDHandle of the device on which the point thePoint (in global coordinates) lies, and returns 0. If thePoint lies on no device, then \*theDevice is set to nil and pt2GDevice returns 1.
- pt2GDeviceRect:** Returns the rectangle of the GDevice on which a global point lies.
- ```
char pt2GDeviceRect (
    Point thePoint,
    Rect *theRect );
```
- This routine returns, via the reference parameter theRect, the gdRect of the GDevice on which the point thePoint (in global coordinates) lies, and returns 0. If thePoint lies on no device, then *theRect is left untouched and pt2GDeviceRect returns 1. If the device in question is the main device, the returned rectangle already has the menu bar height removed from it.
- VertCenter:** Draw string vertically, centered in a rectangle.
- ```
void VertCenter (
 char *str,
 Rect *box,
 short spacing);
```
- This routine draws the given Pascal string str vertically in the rectangle box. The spacing parameter allows the caller to compress the characters vertically (if spacing is  $\geq -10$ , VertCenter uses spacing in place of the font's descent value when calculating character height). Each character is centered, and the pen is moved vertically by the font's ascent plus the font's descent, for each character. The string is drawn using the current port's text settings. VertCenter works for both Roman and Kanji text.

**DrawSTRVert:** Draw string vertically, centered in a rectangle with options.

```
void DrawSTRVert (
 short resID,
 short strNum,
 Rect *box,
 short font,
 short size,
 short spacing);
```

This routine retrieves a Pascal string from the STR# resource with resource ID `resID`, string index `strNum`. It then sets the font and size of the current port to `font` and `size`, respectively. Finally, it draws the string vertically in the rectangle `box`, using `VertCenter` (described above). The spacing parameter is passed to `VertCenter`. `DrawSTRVert` works for both Roman and Kanji text.

**PinPt:** Pin a point within a rectangle.

```
void PinPt (
 Point *where,
 Rect *box);
```

This routine is just like the Macintosh Toolbox routine `PinRect` (see IM-I p293) except for how it returns its result. It pins the point referred to by `where` inside the rectangle `box`, and returns its value via `where`.

**DrawSIC4:** Draw a `ics4` icon with a specified destination and transfer mode.

```
void DrawSIC4 (
 short resID,
 Rect *box,
 short mode);
```

This routine loads the `ics4` resource with ID `resID` and draws it into the rectangle `box` in the current port. If the depth of the current port is  $> 4$  bits-per-pixel, then the transfer mode `mode` is used, otherwise `ditherCopy` is used instead. If `box` is less than 16-by-16 in size, then only the needed top-left part of the `ics4` is drawn (that is, this routine will not scale the `ics4` to fit `box`).

**DrawICL8:** Draw a `icl8` icon with a specified destination port and rectangle.

```
void DrawICL8 (
 short resID,
 GrafPtr thePort,
 Rect *box);
```

This routine loads the `icl8` resource with ID `resID` and draws it into the rectangle `box` in the port `thePort`. The transfer occurs in `srcCopy` for destination ports of  $< 8$  bits-per-pixel, and `ditherCopy` for others. If `box` is less than 32-by-32 in size, then only the needed top-left part of the `icl8` is drawn (that is, this routine will not scale the `icl8` to fit `box`). It does the drawing with a call to `DrawICL8Hand`, described below.

- DrawICL8Hand:** Draw a ics4 icon from a handle with a specified destination.
- ```
void DrawICL8Hand (
    Handle theicl,
    GrafPtr thePort,
    Rect *box );
```
- This routine draws the icl8 stored in Handle theicl into the rectangle box in the port thePort. The transfer occurs in srcCopy for destination ports of < 8 bits-per-pixel, and ditherCopy for others. If box is less than 32-by-32 in size, then only the needed top-left part of the icl8 is drawn (that is, this routine will not scale the icl8 to fit box).
- DrawFullICL8Hand:** Scales an ics4 icon into a specified destination.
- ```
void DrawFullICL8Hand (
 Handle theicl,
 GrafPtr thePort,
 Rect *box);
```
- This routine draws the entire 32-by-32 icl8 stored in Handle theicl into the rectangle box in the port thePort scaling if necessary. The transfer occurs in srcCopy for destination ports of < 8 bits-per-pixel, and ditherCopy for others.
- SlotToGD:** Returns the GDevice handle associated with a given slot.
- ```
GDHandle SlotToGD (short slot s);
```
- This routine returns the GDevice handle corresponding to a given slot number. If there is no corresponding GDevice, (because the slot number is out of range or there's no video card in the slot), then SlotToGD returns nil.
- SlotToRect:** Returns the gdRect of the GDevice for a given slot.
- ```
void SlotToRect (
 short slot,
 Rect *box);
```
- This routine returns the gdRect of the GDevice corresponding to a given slot number, via the reference parameter box. If there is no corresponding GDevice, (because the slot number is out of range or there's no video card in the slot), then SlotToRect returns an empty rectangle.
- PtClose:** Returns true if two points are "close" to each other.
- ```
char PtClose (
    Point pt1,
    Point pt2 );
```
- This routine returns true if pt1 is within 2 pixels of pt2 both horizontally and vertically, in either direction.
- HatchBox:** Draws a hatch pattern in a given rectangle.
- ```
void HatchBox (Rect *box);
```



This routine paints a crosshatch pattern in the given rectangle box, using whatever is the current transfer mode. Afterwards, it does a `PenNormal()` call. This routine can give many different hatching effects by using different transfer modes.

**GetArrow:** Returns a pointer to the standard QuickDraw arrow cursor.

```
CursPtr GetArrow (void);
```

This routine returns a pointer to the standard QuickDraw arrow cursor, that is, `&qd.arrow`.

**EqualColor:** Returns true if two RGB colors are identical.

```
Boolean EqualColor (
 RGBColor *color1,
 RGBColor *color2);
```

This routine returns true if the color pointed to by *color1* is exactly identical to the color pointed to by *color2*, otherwise it returns false.

**EqualColor8:** Returns true if two 8-bit RGB colors are identical.

```
Boolean EqualColor8 (
 Color8 *color1,
 Color8 *color2);
```

This routine returns true if the 8-bit color pointed to by *color1* is exactly identical to the 8-bit color pointed to by *color2*, otherwise it returns false.

## List Manager Lists Routines

**FindSelect:** Find the cell index of the first selected cell in a list.

```
short FindSelect (ListHandle theList);
```

This routine returns the index of the first selected cell in the list *theList*. If no cell is selected, `FindSelect` returns -1. `FindSelect` is designed for n-row, 1-column lists with only one selected cell at a time.

**SetSelect:** Select a given cell in a list, deselecting all others.

```
void SetSelect (
 ListHandle theList,
 short item);
```

This routine selects the cell with index *item* in the list *theList*, and deselects all other cells. If *item* is beyond the bounds of the list, no cell is selected. `SetSelect` is designed for n-row, 1-column lists with only one selected cell at a time.

**WhichCell:** Tells which cell a given point is over.

```
char WhichCell (
 Point where,
 Cell *whichcell,
 ListHandle thelist);
```

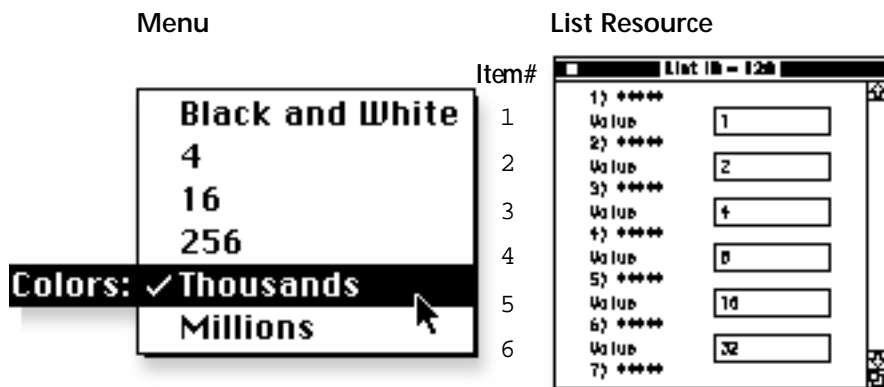
This routine determines which cell of list the list the point where is over. If where is over some cell, the cell is returned via the reference parameter whichcell and the function returns true. If where is not over any cell, the function returns false.

## Data Lists Routines

**DataLookup:** Return an indexed short from an array stored in a List resource.

```
short DataLookup (
 short resID,
 short item);
```

This routine returns the value of the short at index item in the array of shorts stored in a resource of type List with the resource ID resID. If there are n shorts in the List resource, then item's range is from 0..n-1. This routine, in conjunction with ReverseLookup (described below), is a handy way to associate parallel lists of values, as in the example below:

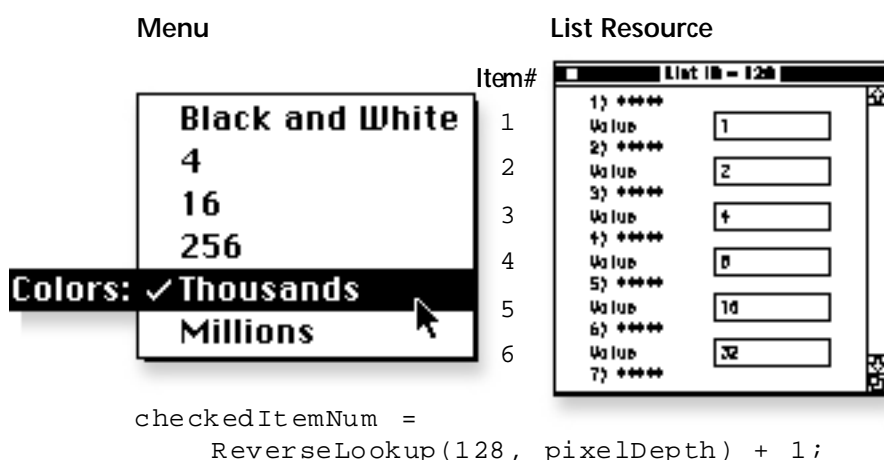


```
pixelDepth = DataLookup(128, itemNum - 1);
```

**ReverseLookup:** Given a value from a List resource, return the associated index.

```
short ReverseLookup (
 short resID,
 short value);
```

This routine returns the index of the short value value in the array of shorts stored in a resource of type List with the resource ID resID. If value is found in the specified List resource, the index (0..n-1) is returned. If value is not found, ReverseLookup returns -1. This routine, in conjunction with DataLookup (described below), is a handy way to associate parallel lists of values, as in the example below:

**DeleteHandItem:**

Delete an item from a list of fixed-sized items stored in a Handle.

```
void DeleteHandItem (
 short which,
 void *data,
 short dataSize);
```

This routine deletes a single fixed-size item from a list of items stored in a Handle. DeleteHandItem deletes item number which (in the range 0..n-1) from the list stored in handle data, given the item size in bytes dataSize. DeleteHandItem, in conjunction with InsertHandItem (described below) provide an easy way to manage lists of records stored in handles. Note that while h is declared as a void pointer it must be a handle.

**InsertHandItem:**

Insert an item into a list of fixed-sized items stored in a Handle.

```
void InsertHandItem (
 short which,
 void *data,
 short dataSize,
 void *item);
```

This routine inserts a single fixed-size item into a list of items stored in a Handle at a given position. InsertHandItem inserts the item item the list stored in handle data, such that its item number in the list is which (in the range 0..n-1). The item size in bytes is given by dataSize. If which is 0, data is inserted at the beginning of the list. If which is  $\geq n$ , data is stored at the end of the list. Using 32767 always adds to the end of the list. InsertHandItem, in conjunction with DeleteHandItem (described above) provide an easy way to manage lists of records stored in handles. Note that while h is declared as a void pointer it must be a handle.

**File Routines****DirIDFromPath:**

Translate a full path into a directory ID and a volume refNum.

```
short DirIDFromPath (
 StringPtr path,
 long *dirID,
 short *vRefNum);
```

This routine converts a full text path into a dirID and vRefNum. The full text path stored in Pascal string path is translated into a directory ID and a volume reference number via the reference parameters dirID and vRefNum, respectively. DirIDFromPath returns an OSerr.

**PathFromDirID:** Translate a directory ID and a volume refNum into a full path.

```
short PathFromDirID (
 long dirID,
 short vRefNum,
 StringPtr path);
```

This routine converts a dirID and vRefNum into a full text path. The directory ID dirID and the volume reference number vRefNum are translated into a full path which is stored as a Pascal string in the buffer pointed to by path. PathFromDirID returns an OSerr.

**CountVolumes:** Returns the number of currently mounted volumes.

```
short CountVolumes (void);
```

This routine returns the number of currently mounted volumes. Use this routine in conjunction with GetIndVolume (described below) to iterate through volumes.

**GetIndVolume:** Returns and indexed volume reference number.

```
short GetIndVolume (short which);
```

This routine returns the volume reference number of the volume index which (ranging from 1..CountVolumes()). Use this routine in conjunction with CountVolumes to iterate through the mounted volumes.

**GetVolIndex:** Returns the volume index for a given volume reference number.

```
short GetVolIndex (
 short ref,
 short count);
```

This routine returns the volume index, maximum count, associated with the volume reference number ref. Use this routine in conjunction with CountVolumes and GetIndVolume.

## Menu Routines

**WidenMenu2Box:** Widen the first item of a menu to fit a particular box.

```
void WidenMenu2Box (
 MenuHandle theMenu,
 Rect *box);
```

This routine widens the first item of the popup menu stored in theMenu such that it is as wide as the rectangle box, leaving room for various popup adornments. Premiere's popup CDEF uses this routine at CDEF init time to widen the menu to fit the control rectangle.

Normally it is called by `WidenMenu` (described below), so you probably won't need to call it yourself.

**WidenMenu:** Widen the first item of a popup menu to fit its dialog item.

```
void WidenMenu (
 DialogPtr theDialog,
 short item);
```

This routine widens the first item of menu associated with item number `item` in dialog `theDialog` (which must be a Premiere popup CDEF control item) such that the menu fully occupies the width of the dialog item. Use this routine if, in the course of conducting a dialog, you change or rebuild a popup menu.

**GetMenuWidth:** Widen the first item of a popup menu to fit its dialog item.

```
short GetMenuWidth (MenuHandle theMenu);
```

This routine returns the width of the widest menu item text in the menu `theMenu` plus 28 to compensate for the width of a popup arrow.

## Math Routines

**LimitLong:** Limit a long coordinate to -16,384 to 16,383.

```
short LimitLong (long num);
```

This routine limits `num` to within a 32,768-pixel range centered about zero. Premiere uses this to avoid trying to draw lines with lengths greater than 32,768 pixels (which, because of integer overflow, don't draw). You can use it in a similar fashion.

**QuickAbs:** Return the absolute value of a long integer.

```
long QuickAbs (long val);
```

This routine returns the absolute value of `val`.

**FixMulDiv:** Calculates  $a * b / c$  in 16.16 fixed-point with an intermediate 32.32 value.

```
Fixed FixMulDiv (
 Fixed value,
 long mul,
 long div);
```

This routine calculates `value * mul / div` in 16.16 fixed-point. During the calculation an intermediate value of 32.32 precision is kept to prevent overflow.

**FixedDiv:** Calculates a 16.16 fixed-point division quickly.

```
Fixed FixedDiv (
 Fixed value,
 Fixed div);
```

This routine returns `value / div` quickly in 16.16 fixed-point format.

- inttox80:** Converts an int to an extended80 floating point value.  
`extended80 inttox80 (long value);`  
 This routine converts the long value to and extended80 floating point type.
- longdoubletox80:** Converts a long double (aka "extended") to an extended80 floating point value.  
`extended80 longdoubletox80 (long double value);`  
 This routine converts the long double ("extended") value to and extended80 floating point type.
- x80tolongdouble:** Converts an extended80 floating point value to a long double.  
`long double x80tolongdouble (extended80 value);`  
 This routine converts the extended80 floating point value to an long double.
- x80toint:** Converts an extended80 floating point value to an int.  
`long x80toint (extended80 value);`  
 This routine converts the extended80 floating point value to an int (truncating).
- pie:** Returns  $\pi$  as a long double.  
`long double pie (void);`  
 This routine returns the constant  $\pi$  (3.1415926...) as a long double floating point value. This function is provided because the 68K headers define pi as a constant and the PowerPC headers define pi() as a function, which makes source compatibility a problem. Just use pie() instead.

## Premiere-Specific Routines

### Interface Compatibility Routines

These routines make coding plug-ins for the Premiere on the PowerPC easier by providing glue routines between your code and some Toolbox, OS, and QuickTime routines that use callbacks. For example, instead of calling ModalDialog, call PrModalDialog with the address of your filter procedure as you're used to, and the filter proc UPP is handled automatically. On the 68K these routines simply call through to their Toolbox, OS, or QuickTime counterparts.

- PrModalDialog:** Premiere glue for ModalDialog.  
`void PrModalDialog (  
     ModalFilterProcPtr filterProc,  
     short *itemHit );`  
 This routine is just like ModalDialog but on the PowerPC handles the temporary creation and disposal of an appropriate UPP for the filterProc parameter.

**PrCustomGetFile:** Premiere glue for CustomGetFile.

```
pascal void PrCustomGetFile (
 FileFilterYDProcPtr fileFilter,
 short numTypes,
 SFTypeList typeList,
 StandardFileReply *reply,
 short dlgID,
 Point where,
 DlgHookYDProcPtr dlgHook,
 ModalFilterYDProcPtr filterProc,
 short *activeList,
 ActivateYDProcPtr activateProc,
 void *yourDataPtr);
```

This routine is just like CustomGetFile but on the PowerPC handles the temporary creation and disposal of appropriate UPPs for the fileFilter, dlgHook, filterProc, and activateProc parameters.

**PrCustomPutFile:** Premiere glue for CustomPutFile.

```
pascal void PrCustomPutFile (
 ConstStr255Param prompt,
 ConstStr255Param defaultName,
 StandardFileReply *reply,
 short dlgID,
 Point where,
 DlgHookYDProcPtr dlgHook,
 ModalFilterYDProcPtr filterProc,
 short *activeList,
 ActivateYDProcPtr activateProc,
 void *yourDataPtr);
```

This routine is just like CustomPutFile but on the PowerPC handles the temporary creation and disposal of appropriate UPPs for the dlgHook, filterProc, and activateProc parameters.

**PrCustomGetFilePreview:** Premiere glue for CustomGetFilePreview.

```
pascal void PrCustomGetFilePreview (
 FileFilterYDProcPtr fileFilter,
 short numTypes,
 SFTypeList typeList,
 StandardFileReply *reply,
 short dlgID,
 Point where,
 DlgHookYDProcPtr dlgHook,
 ModalFilterYDProcPtr filterProc,
 short *activeList,
 ActivateYDProcPtr activateProc,
 void *yourDataPtr);
```

This routine is just like CustomGetFilePreview but on the PowerPC handles the temporary creation and disposal of appropriate UPPs for the fileFilter, dlgHook, filterProc, and activateProc parameters.

**PrSCSetInfo:** Premiere glue for SCSetInfo.

```
pascal ComponentResult PrSCSetInfo (
 ComponentInstance ci,
 OSType type,
 SModalFilterProcPtr scModalFilter,
 SModalHookProcPtr scModalHook,
 long refCon,
 StringPtr customName);
```

This routine is just like SCSetInfo but on the PowerPC handles the temporary creation and disposal of appropriate UPPs for the scModalFilter and scModalHook parameters.

**PrSGSettingsDialog:** Premiere glue for SGSettingsDialog.

```
pascal ComponentResult PrSGSettingsDialog (
 SeqGrabComponent s,
 SGChannel c,
 short numPanels,
 Component *panelList,
 long flags,
 SModalFilterProcPtr proc,
 long procRefNum);
```

This routine is just like SCSetInfo but on the PowerPC handles the temporary creation and disposal of an appropriate UPP for the proc parameter.

**PrTrackControl:** Premiere glue for TrackControl.

```
pascal short PrTrackControl (
 ControlHandle theControl,
 Point thePoint,
 ProcPtr actionProc);
```

This routine is just like TrackControl but on the PowerPC handles the temporary creation and disposal of an appropriate UPP for the actionProc parameter.

**PrSndNewChannel:** Premiere glue for SndNewChannel.

```
pascal OSErr PrSndNewChannel (
 SndChannelPtr *chan,
 short synth,
 long init,
 SndCallbackProcPtr userRoutine);
```

This routine is just like TrackControl but on the PowerPC handles the creation of an appropriate UPP for the userRoutine parameter.

***Important!*** If you use the routine, be sure to dispose the channel with PrSndDisposeChannel, otherwise the UPP will not be deallocated.

**PrSndDisposeChannel:** Premiere glue for SndDisposeChannel.

```
pascal OSErr PrSndDisposeChannel (
 SndChannelPtr chan,
 Boolean quietNow);
```



This routine is just like `SndDisposeChannel` but on the PowerPC handles the disposal of the UPP for the callback procedure.

**Important!** Only use this routine to dispose a sound channel you created with `PrSndNewChannel`.

## Global Accessor Routines

**GetAGlobal:** Get the value of a Premiere global variable.

```
pascal long GetAGlobal (long identifier);
```

This routine returns the value of the global variable specified symbolically by identifier. The global identifiers are provided in the Premiere headers. Specific useful globals are documented in detail in the [Globals](#) chapter.

**Important!** Do not attempt to use the values of global variables that are not specifically documented in the [Globals](#) chapter.

**SetAGlobal:** Set the value of a Premiere global variable.

```
pascal void SetAGlobal (
 long identifier,
 long value);
```

This routine sets the value of the global variable specified symbolically by identifier to the value value. The global identifiers are provided in the Premiere headers. Specific useful globals are documented in detail in the [Globals](#) chapter.

**Important!** Do not change the values of Premiere globals that are not specifically documented in the [Globals](#) chapter—stability will be compromised.

## Text, Strings, and Memory Routines

**BuildString:** Expand text from a 'TEXT' resource with parameter replacement.

```
Handle BuildString (
 short which,
 short listID,
 long value1,
 ...);
```

This routine returns a handle containing the text from the 'TEXT' resource with ID which using expansion values value1, etc. BuildString works somewhat like the way ParamText strings are substituted in a dialog or alert. The parameter listID gives the resource ID of the string list from which BuildString will retrieve strings for '&n' expansion identifiers, or, in the case of 'n' and '%n' identifiers, the time code format (see the list of formats below). The n in the identifiers determines which function parameter should be used to substitute for the identifier. They are specified by number so that the parameters may be reordered for international localization without changing the code. Your TEXT resource may contain the following expansion

identifiers:

Table 0-2: BuildString Identifiers

| Identifier | Parameter | Description                                                      |
|------------|-----------|------------------------------------------------------------------|
| ^n         | long      | Replace with ASCII decimal value of parameter n.                 |
| ^ ^        | none      | Replace with single " ^ " character.                             |
| %n         | long      | Replace with ASCII timecode value of parameter n, showing hours. |
| %%         | none      | Replace with single "% " character.                              |
| }n         | long      | Replace with ASCII timecode value of parameter n.                |
| }}         | none      | Replace with single " } " character.                             |
| /n         | char *    | Replace with Pascal string pointed to by parameter n.            |
| //         | none      | Replace with single " / " character.                             |
| &n         | long      | Replace with string number given by parameter n from STR#.       |
| &&         | none      | Replace with single " & " character.                             |
| {n         | long      | Replace with ASCII byte quantity of parameter n (use K, M).      |
| {{         | none      | Replace with single " { " character.                             |

When using a '%n' or '}n' identifier, the listID is interpreted as having one of the values from the first part of the list, with some of the values from the second part of the list possibly ORed over it to specify the time code format. The constants shown below are not included in the Premiere header files that come with the basic Plug-in Developer's kit, but you can type them in and use them.

```
enum {
 ffFrameCount = 1, // count in frames FFFFFF
 ffNonDrop24 = 24, // 24fps HH:MM:SS:FF
 ffNonDrop25 = 25, // 25fps HH:MM:SS:FF
 ffNonDrop30 = 30, // 30fps, non-drop
 // HH:MM:SS:FF
 ffNonDrop100 = 100, // 100fps HH:MM:SS:FFF
 ffNonDrop600 = 600, // 600fps HH:MM:SS:FFF
 ffDropFrame = 29, // 30fps, drop-frame
 // HH;MM;SS;FF
 ffFeetFrame16 = 16, // Feet.Frames (35mm) ffff.FF
 ffFeetFrame40 = 40, // Feet.Frames (16mm) ffff.FF
 ffDelta = 0x4000, // delta symbol in front of
 // time
 ffTensHours = 0x3000, // show tens of hours
 ffTensFlag = 0x2000, // test bit for tens of hours
 ffHours = 0x1000, // show hours
 ffOffset = 0x0800 // +SS:FF or -SS:FF
};
```

Here's an example. For the following TEXT resource:

```
resource 'TEXT' (10000, "BuildString example 1") {
 "There are ^0 /1s at time %2, (yes, ^0)."
```

And the following code:

```
{
 Handle h;

 h = BuildString(10000, ffNonDrop30 | ffHours, 10,
 "\pwidget", 45);
 DrawText(*h, 0, GetHandleSize(h));
 DisposeHandle(h);
}
```

Would display the following output:

```
There are 10 widgets at time 0:00:01:15, (yes, 10).
```

Note that because the identifiers tell which parameter they get their value from, the `^0` at the end of the TEXT resource references the first parameter again. Also, since we used a time-code identifier in our TEXT resource (`%2`), we had to specify the time code format in the *listID* parameter. This means that you cannot mix `&n` identifiers with `'}n'` and `'%n'` identifiers in the same template.

Here's an example of how to use the string lookup feature. For the following TEXT resource and STR# resource:

```
resource 'TEXT' (10001, "BuildString example 2") {
 "Word #^0: &0. "
};

resource 'STR#' (10001, "BuildString example 2") {
 "Good",
 "Better",
 "Best",
};
```

And the following code:

```
{
 short i;
 Handle h;

 for (i = 1; i <= 3; i++) {
 h = BuildString(10001, 10001, i);
 DrawText(*h, 0, GetHandleSize(h));
 DisposeHandle(h);
 }
}
```

Would display the following output:

```
Word #1: Good. Word #2: Better. Word #3 Best.
```

Here the TEXT template used the *i* parameter twice, the second time using it as the index into the STR# resource that we specified in the *listID* parameter of BuildString.

### StringTrunc:

Copies a string then truncates it to a particular screen width.

```
void StringTrunc (
 StringPtr instr,
 StringPtr outstr,
 short width);
```

This routine first copies *instr* to *outstr* then truncates the string at the end so that it will fit in *width* pixels on the screen at the current port's font, size, and style. The

truncation is done in an internationally compatible way (that is, with the Script Manager TruncString routine).

**TruncLength:** Truncates a string in place to be no more than a certain number of bytes.

```
void TruncLength (
 StringPtr instr,
 short width);
```

This routine truncates (by changing the length byte) the string `instr` to be no longer than `width` bytes. It is smart enough to not chop a two-byte character in half.

**NumToQuan:** Converts a memory- or disk-size quantity to an ASCII representation.

```
void NumToQuan (
 long num,
 StringPtr str);
```

This routine converts `num`, which is assumed to be a size in bytes, into a string with the usual abbreviations (K or M). It stores the text representation in the buffer pointed to by `str`. Here are some examples:

| Value   | Output |
|---------|--------|
| 512     | 512    |
| 753049  | 735.3K |
| 3672415 | 3.5M   |

**ExtraNewHandle:** Try to allocate a handle in temp memory, then in local heap.

```
Handle ExtraNewHandle (long size);
```

This routine is like `NewHandle` except that it tries to allocate a handle of size `size` from temporary (MultiFinder) memory first. If that fails, it tries again in the local heap. If both tries fail `ExtraNewHandle` returns `nil`—check `MemError()` for the error code.

## Clip Routines

The routines in this category are used to get information about clips in Premiere. These routines will be of most use in Export ('ExpD') modules, but can be used in other types of plug-ins as well. For more information on clips in Premiere see the section [Premiere Terminology](#) in the [Introduction](#).

**ClipStart:** Returns the in-point for a clip for a given frame rate.

```
long ClipStart (
 short clipID,
 short fps);
```

This routine returns the in-point for the clip specified by `clipID` at the frame rate `fps`.

**ClipRate:** Returns the playback rate (speed) for a clip.

```
short ClipRate (short clipID);
```

This routine returns the playback rate (speed) for the clip specified by clipID. Normal playback speed is 100. Note that ClipRate may return a negative value if the speed is negative (backwards).

**GetClipTitle:** Returns the title for a clip that appears in the Project window.

```
void GetClipTitle (
 short clipID,
 StringPtr str,
 char *alias);
```

This routine returns the title for the clip specified by clipID into the buffer str. If the clip's name has been changed by the user with the Name Alias command, the char pointed to by alias will be set to non-zero, otherwise zero.

**ClipFile:** Returns the file ID of the file associated with a clip.

```
short ClipFile (short clipID);
```

This routine returns the file ID associated with the clip specified by clipID. Note that this is a Premiere file ID, not a file system reference number. Once you have the file ID for a clip, you can use GetFSSpec to get the complete file specification for the file. See GetFSSpec in the File category of this section.

**ClipSize:** Returns the total media length of a clip in frames at a given frame rate.

```
long ClipSize (
 short clipID,
 short fps);
```

This routine returns the total media length (ignoring the in-point and out-point) for the clip specified by clipID at the frame rate fps. To get the length in frames between the in-point and the out-point, use ClipWidth, documented below.

**ClipWidth:** Returns the length from in- to out-point of a clip in frames at a given frame rate.

```
long ClipWidth (
 short clipID,
 short fps);
```

This routine returns the length in frames between the in-point and the out-point for the clip specified by clipID at the frame rate fps. To get the total media length (ignoring the in-point and out-point), use ClipSize, documented above.

**ClipAspect:** Returns the state of a clip's "maintain aspect" attribute.

```
short ClipAspect (short clipID);
```

This routine returns the state of the "maintain aspect ratio" attribute of the clip specified by clipID. Non-zero says to maintain the aspect ratio, zero indicates not.

- GetClipBackwards:** Returns the state of a clip's "backwards" attribute.
- ```
short GetClipBackwards (short clipID);
```
- This routine returns the state of the "backwards" attribute of the clip specified by clipID. Non-zero means the clip is to play backwards, zero indicates not.
- CountClipMarkers:** Returns the number of markers for a clip.
- ```
short CountClipMarkers (short clipID);
```
- This routine returns the number of markers for the clip specified by clipID, which will always be at least 12.
- GetClipMarker:** Returns the location in frames of a given marker at a specified frame rate.
- ```
long GetClipMarker (
    short clipID,
    short marker,
    short fps );
```
- This routine returns the location of the marker with index marker in the clip specified by clipID at the frame rate fps. The returned value is in frames at fps. The marker parameter ranges from 0 to CountClipMarkers() - 1. Marker 0 is the in-point, marker 1 is the out-point, markers 2-11 are the numbered markers 0-9, and markers with an index > 11 are unnumbered markers. See also CountClipMarkers, FindClipMarker, PreviousClipMarker, and NextClipMarker below.
- FindClipMarker:** Returns the number of the marker at a specific time, if any.
- ```
short FindClipMarker (
 short clipID,
 long value,
 short fps);
```
- This routine returns the number of the marker in the clip specified by clipID that is exactly at the frame value at the frame rate fps. If no such marker exists, FindClipMarker returns -1.
- PreviousClipMarker:** Returns the first marker to the left of the given frame for a clip.
- ```
short PreviousClipMarker (
    short clipID,
    long value,
    short fps );
```
- This routine returns the number of the marker in the clip specified by clipID that to the left (that is, earlier in time) of the frame value at frame rate fps. If there is no previous marker, PreviousClipMarker returns -1.
- NextClipMarker:** Returns the first marker to the right of the given frame for a clip.
- ```
short NextClipMarker (
 short clipID,
 long value,
 short fps);
```

This routine returns the number of the marker in the clip specified by clipID that to the right (that is, later in time) of the frame value at frame rate fps. If there is no next marker, NextClipMarker returns -1.

## File Routines

The routines in this category are used to get information about files in Premiere. These routines will be of most use in Export ('ExpD') modules, but can be used in other types of plug-ins as well. For more information on files in Premiere see the section Premiere Terminology in the [Introduction](#).

**GetFSSpec:** Get the FSSpec corresponding to a Premiere file ID.

```
void GetFSSpec (
 short fileID,
 FSSpec *thespec);
```

This routine returns via reference parameter thespec a Macintosh FSSpec for the file specified by the Premiere file ID fileID. You can use this information to access the file directly if you wish.

**FindFType:** Find the file type for a given file.

```
short FindFType (
 StringPtr name,
 long dirID,
 short vRefNum,
 long *ftype);
```

This routine returns via reference parameter ftype the type of the file specified by name, dirID, and vRefNum. The GetFSSpec routine returns the result of its internal HGetInfo call as the function result.

**exists:** Tells whether a particular disk file exists.

```
char exists (
 StringPtr name,
 short vrefnum,
 long dirID);
```

This routine returns true if the file specified by name, vrefnum and dirID exists, false if not.

**SuperFileInit:** Initialize a SuperFileRec for buffered file reads.

```
OSErr SuperFileInit (
 short ref,
 long bufferSize,
 SuperFilePtr superData);
```

This routine is the first of four routines that make up a buffered file reader, which allows high speed read access to a file, even if only small numbers of bytes are read at a time. SuperFileInit initializes a SuperFileRec for buffered file reads.

**Important!** The ref parameter is the Macintosh file reference number of an **already open-for-reading file**. The **bufferSize** parameter is the chunk size you'd like the SuperFile package to read when it has to actually read from the file—32768 is a good choice. The **superData** parameter is a pointer to the SuperFileRec you're initializing. The

*SuperFile* package will use this record for state information. *SuperFileInit* may return *paramErr* if **ref** or **superData** is nil, or **memFullErr** if it was unable to allocate its internal file buffer.

Following is an example of how you might use the *SuperFile* package (using all four *SuperFile* routines). The example is a reader for an imaginary data file that has a 512 byte reserved section at the beginning, a 512-entry interleaved list containing a 256-byte string followed by a 26-byte record, a 1024-byte reserved area, and a 256K image. Given three pre-allocated pointers, the following routine will unroll the interleaved file list into two pointers, one for the strings and another for the 26-byte records, skip the reserved area, and read the image at the end.

```
OSErr MyFileReader (Ptr dest1, Ptr dest2, Ptr dest3,
 FSSpec *mySpec)
{
 SuperFileRec sf; //The SuperFile state information
 long count; //The byte count for SuperFileRead
 long pos; //File position for SuperFileSeek
 OSErr err = noErr; //Error code from various calls
 short ref; //The Mac refNum of the data file
 short i; //The iterator for reading our list
 Boolean fileOpen; //Tells if we've opened the file
 Boolean superFileUp; //Tells if we've done a
 // SuperFileInit

 // First, try to open the file.
 fileOpen = superFileUp = false;
 err = FSpOpenDF(mySpec, fsRdPerm, &ref);

 // Skip the reserved 512 bytes at the beginning my
 // data file.
 if (err == noErr)
 {
 fileOpen = true;
 err = SetFPos(ref, fsFromStart, 512);
 }

 // Fire up a SuperFile starting right here.
 if (err == noErr)
 {
 superFileUp = true;
 err = SuperFileInit(ref, 32768, &sf);
 }

 // Read the interleaved 512-entry list out of my data
 // file into the two array pointers dest1 and dest2.
 if (err == noErr)
 {
 for (i = 0; i < 512 && err == noErr; i++)
 {
 // Read the string into dest1
 count = 256;
 err = SuperFileRead(&sf, &count, dest1);
 dest1 += 256;

 // Read the MyListRec record into dest2
 if (err == noErr)
 {
 count = sizeof(MyListRec);
 err = SuperFileRead(&sf, &count, dest2);
 dest2 += sizeof(MyListRec);
 }
 }
 }

 // Skip the 1K reserved area that follows the list.
```



```

if (err == noErr)
{
 pos = 512 + (512 * (256 + sizeof(MyListRec)))
 + 1024;
 err = SuperFileSeek(&sf, pos);
}

// Read the big image at the end of the file
// into dest3.
if (err == noErr)
{
 count = 256 * 1024;
 err = SuperFileRead(&sf, &count, dest3);
}

// Dispose the SuperFile private storage and close
// the file
if (superFileUp)
 SuperFileDispose(&sf);
if (fileOpen)
 FSClose(ref);

return(err);
}

```

**SuperFileRead:** Read some bytes from a SuperFile.

```

OSErr SuperFileRead (
 SuperFilePtr superData,
 long *userCount,
 Ptr outBuf);

```

This routine is like FSRead, except that it reads data from a SuperFile that has been previously initialized with SuperFileInit. The superData parameter is a pointer to an initialized SuperFileRec. The userCount parameter points to a long that specifies the number of bytes to read. Also, the actual number of bytes transferred is returned via the reference parameter userCount. The outBuf parameter is the buffer into which the bytes are transferred. SuperFileRead is very efficient even if the number of bytes requested is small—the file is buffered by the SuperFile package. SuperFileRead will return the results of FSRead calls, but remember—the point of the SuperFile package is to do large FSReads and buffer the data, so there probably won't be an FSRead call every time you call SuperFileRead.

**Important!** SuperFileRead does not consider an eofErr from the file system to be an error. SuperFileRead will **never return eofErr**, so if you don't already know how much data is in the file, you'll have to check for end-of-file yourself. See the example in the description of SuperFileInit for more information.

**SuperFileSeek:** Seek a SuperFile to a particular location.

```

OSErr SuperFileSeek (
 SuperFilePtr superData,
 long position);

```

This routine seeks the SuperFile represented by the SuperFileRec pointed to by superData to the position position, in bytes, from the beginning of the file.

**Important!** The SuperFile package is most efficient when it is used to read linearly—seeking may reduce SuperFile performance. See the example in the description of

*SuperFileInit* for more information.

**SuperFileDispose:** Dispose a SuperFile's private storage.

```
OSErr SuperFileDispose (
 SuperFilePtr superData);
```

This routine disposes the private storage allocated by the SuperFile package at SuperFileInit time.

**Important!** *SuperFileDispose* does **not** close the file. See the example in the description of *SuperFileInit* for more information.

**SafeSetupAIFFHeader:** Bug-free version of SetupAIFFHeader.

```
short SafeSetupAIFFHeader (
 short ref,
 short channels,
 unsigned long speed,
 short bits,
 long compression,
 long numBytes,
 long numFrames);
```

This routine is just like SetupAIFFHeader except that it properly handles sample rates over 32767.0. The ref parameter is the Macintosh file reference number of the file to which you're writing. The speed parameter is an unsigned Fixed number in 16.16 format, bits is the number of bits per sample, compression is the compression type which in SafeSetupAIFFHeader is ignored. The numBytes parameter is the sound data chunk data length, and the numFrames parameter is ignored by SafeSetupAIFFHeader (instead, the number of sample frames is calculated using the formula  $\text{numBytes} * (\text{bits} / 8)$ ).

## Debugging Routines

**PrDebug:** Send a formatted string to the Premiere Debug window.

```
void PrDebug (
 short panel,
 char *str,
 ...);
```

This routine allows you to easily send debugging information to Premiere's Debug window, which is built into every version of Premiere. To display the debug window, hold Control and Option and press "0" (zero). The panel parameter is 0 for the top (large) part of the window, or 1 for the smaller lower panel. The str parameter is a formatting string similar to printf, and

the values for the identifiers follow str. The table below shows the valid identifiers and what they display.

**Table 0–3: PrDebug Identifiers**

| Identifier | Parameter   | Description                                                            |
|------------|-------------|------------------------------------------------------------------------|
| %d         | long        | Replace with ASCII decimal value of parameter.                         |
| %f         | Fixed       | Replace with ASCII decimal fixed-point value of parameter.             |
| %s         | StringPtr   | Replace with Pascal string pointed to by parameter.                    |
| %t         | OStype      | Replace with four-character OStype parameter.                          |
| %r         | Rect *      | Replace with text representation of Rect pointed to by parameter.      |
| %p         | Point *     | Replace with text representation of Point pointed to by parameter.     |
| %R         | LongRect *  | Replace with text representation of LongRect pointed to by parameter.  |
| %P         | LongPoint * | Replace with text representation of LongPoint pointed to by parameter. |

Here's an example of a call to PrDebug.

```
void MyTestFunction (void)
{
 Point pt = { 10, 5 };
 unsigned char *str = "\pCarpe diem";
 OStype cook = 'COOK';
 short value = 42;

 PrDebug(0, "\pToday's saying: \"%s\", pt = %p,
 type = %t, value = %d.", str, &pt, cook,
 value);
}
```

The output you'd see in Premiere's debug window would be:

```
Today's saying: "Carpe diem", pt = [h=5,v=10], type =
COOK, value = 42.
```

## Cursor Control Routines

### SpinCurs:

Starts the cursor spinning after a delay.

```
void SpinCurs (short delay);
```

This routine causes the cursor to start spinning after delay ticks have elapsed. The spinning is handled by a VBL task, so you don't have to do anything further. To start the cursor spinning immediately, do a SpinCurs(0). To stop cursor spinning, see StopCurs documented below.

### StopCurs:

Stop the cursor spinning.

```
void StopCurs (void);
```

This routine causes the cursor to stop spinning immediately.

**MySetCursor:** Set the cursor to one of Premiere's color cursors.

```
void MySetCursor (short theID);
```

This routine sets the cursor to one of Premiere's color cursors.

**Important!** *This routine is not a general routine. It looks for **theID** in a list of preloaded color cursors. If you can find a Premiere color cursor that fits your application among the 58 that are listed in the Premiere headers, you can use `MySetCursor` to set that cursor. For your own custom cursors, stick to `GetCCursor` and `SetCCursor`.*

**Important!** *Do not call this routine at interrupt time—it calls `SetCCursor`, which moves memory.*

## Event Routines

**CheckStop:** Checks whether the user has pressed Command-".".

```
char CheckStop (void);
```

This routine uses `GetOSEvent` to determine if the user has pressed Command-".", and returns true if so, false if not. `CheckStop` uses `GetOSEvent` to prevent a context switch.

**CheckStopUpdate:** Checks whether the user has pressed Command-".", and updates background windows.

```
char CheckStopUpdate (void);
```

This routine uses `GetNextEvent` to determine if the user has pressed Command-".", and returns true if so, false if not. Also `CheckStopUpdate` will process any pending update events, which allows other Premiere windows to update. Note that because `CheckStopUpdate` uses `GetNextEvent`, there is the possibility of a context switch whenever this routine is called. To prevent the context switch, use `UpdateAllWindows` (described in the Window category of this section) to manually update background windows and use `CheckStop` (described above) instead.

**GetModifiers:** Gets the modifiers of the last event or the current modifiers.

```
short GetModifiers (EventRecord *theEvent);
```

This routine returns a set of modifier flags based on the modifiers field of the event record pointed to by `theEvent`. If `theEvent` is nil, the `GetModifiers` does a `GetKeys` and the output is based on that. Note that in Premiere the space bar is considered a modifier in some instances, so there's a flag for that. Below is a list of the bits that can be set in the output short (these are provided in the Premiere headers):

```
#define bCmd 0x0100
#define bShift 0x0200
#define bCapsLock 0x0400
#define bOption 0x0800
#define bControl 0x1000
#define bSpace 0x2000
```

## Standard File Routines

**SpecialGetFile:** Provides an easy interface to CustomGetFile with Premiere features.

```
char SpecialGetFile (
 FSSpec *thespec,
 short numTypes,
 OSType *types,
 StringPtr origName,
 short dlgID,
 long *type);
```

This routine provides an easy interface to CustomGetFile and provides standard Premiere features like the Find buttons. The file spec of the returned file is returned via the reference parameter thespec. The numTypes parameter tells CustomGetFile how many file types are in the list pointed to by types. The origName string is used as the default text in the Find dialog box, if the user clicks the Find button. The dlgID parameter is the ID of the custom get file dialog resource. The type of the returned file is returned via the reference parameter type, if you don't want this information, pass nil for type. The function returns true if the user chose a file, false if he canceled.

**SpecialGetFilePreview:** Provides an easy interface to CustomGetFilePreview with Premiere features.

```
char SpecialGetFilePreview (
 FSSpec *thespec,
 short numTypes,
 OSType *types,
 StringPtr origName,
 short dlgID,
 long *type,
 WindowPtr theWindow);
```

This routine provides an easy interface to CustomGetFilePreview and provides standard Premiere features like the Find buttons, as well as giving the user the ability to preview sounds and see movie and still picture previews. The thespec, numTypes, types, origName, dlgID, and type parameters are all the same as in SpecialGetFile (described above). The theWindow parameter is used internally by Premiere—you should always use -1.

**MyPutFile:** Provides an easy interface to StandardPutFile with Premiere features.

```
void MyPutFile (
 StringPtr prompt,
 StringPtr origname,
 StandardFileReply *reply);
```

This routine provides an easy interface to StandardPutFile and provides standard Premiere features like the volume free-space box. The prompt string is the prompt for the StandardPutFile dialog. The origname string is the default save name for the StandardPutFile dialog. The reply parameter is a pointer

to a StandardFileReply record into which the results of the StandardPutFile are placed.

**freehook:** A standard file hook for displaying the free space on the current volume.

```
pascal short freeHook (
 short item,
 DialogPtr thedlg,
 void *data);
```

This routine, when passed as a hook to Standard File, displays the volume free space at the bottom of the dialog. You can use this hook to make your standard file dialogs look more like Premiere's. The item, thedlg, and data parameters correspond to the usual Standard File hook parameters.

## Dialog Routines

**ShowModal:** Does a ShowWindow on a modal dialog with proper floating window handling.

```
void ShowModal (WindowPtr theWindow);
```

This routine should be called instead of ShowWindow to show a modal dialog. It performs the correct floating-window maintenance. Remember to make your dialogs "not visible" in their DLOG resource.

**DisposeModal:** Does a DisposeDialog and deallocates UPPs on the Power Macintosh.

```
void DisposeModal (WindowPtr theWindow);
```

This routine should be called instead of DisposeDialog to dispose a modal dialog. It performs the correct floating-window maintenance after disposing the dialog. Also, on PowerPC-based Macintosh models it disposes ControlActionUPPs and UserItemUPPs that were allocated by calls to UserItem and SetCAction (detailed in the Dialog category of the General Macintosh Routines section in this chapter). It is important to do this, otherwise your modal dialogs will leak memory.

**CenterModal:** Puts up a standard modal dialog on the main screen.

```
short CenterModal (short theID);
```

This routine is convenient for putting up message alerts. It loads the DLOG/DITL specified by the theID parameter, centers it on the main screen, then calls PrModalDialog (filtering with modalfilter) until one of the first three buttons are hit. It then disposes the dialog and returns the number of the button that was hit. PrModalDialog is described in the Interface Compatibility category of this section and modalfilter is described below.

**CenterModalKeys:** Puts up a standard modal dialog on the main screen with keyboard equivalents.

```
short CenterModalKeys (short theID);
```

This routine is just like CenterModal except that if the user presses keys CenterModalKeys will find and click on the first button it finds that starts with the key the user pressed. This is the routine that Premiere uses for its "Do you wish to save Untitled-1 before closing" Don't Save/Save/Cancel dialog box.

**modalfilter:** Standard movable modal dialog filter.

```
pascal char modalfilter (
 DialogPtr thedialog,
 EventRecord *event,
 short *itemhit);
```

This routine is a convenient modal dialog filter procedure that allows you to make your modal dialogs behave like movable modal dialogs. This routine handles moving the dialog, update events for background windows, floating-window details, handling of the return, enter, escape, and Command-". keys, and the Edit menu commands Cut, Copy, Paste, and Clear. The parameters thedialog, event, and itemhit) are the standard parameters for a modal dialog filter procedure.

**notextfilter:** Movable modal dialog filter for dialogs with numeric entry fields.

```
pascal char notextfilter (
 DialogPtr thedialog,
 EventRecord *event,
 short *itemhit);
```

This routine is just like modalfilter (described above) except that it only allows the numbers '0' through '9', '-', ', delete, tab, left-arrow, and right-arrow keys through. This is good for dialogs that have numeric edit-fields only. The parameters thedialog, event, and itemhit are the standard parameters for a modal dialog filter procedure.

**hoursfilter:** Movable modal dialog filter for dialogs with time code entry fields.

```
pascal char hoursfilter (
 DialogPtr thedialog,
 EventRecord *event,
 short *itemhit);
```

This routine is just like notextfilter (described above) except that it allows the additional characters ".", ":", and ";" to be typed. This is good for dialogs that have time code edit-fields only. The parameters thedialog, event, and itemhit are the standard parameters for a modal dialog filter procedure.

**decimalfilter:** Movable modal dialog filter for dialogs with decimal entry fields.

```
pascal char decimalfilter (
 DialogPtr thedialog,
 EventRecord *event,
 short *itemhit);
```

This routine is just like `notextfilter` (described above) except that it allows the additional character "." to be typed. This is good for dialogs that have fixed-point or floating-point edit-fields only. The parameters `thediolog`, `event`, and `itemhit` are the standard parameters for a modal dialog filter procedure.

**AlertSystem:** Error alert system.

```
short AlertSystem (
 short type,
 char hascancel,
 short str1x,
 short str1num,
 short str2x,
 short str2num);
```

This routine is a sophisticated error alert facility that is used throughout Premiere. The window that is displayed is a movable modal with the title "Note:". The `type` parameter specifies the icon that is to be displayed in the dialog, typically `stopIcon`, `noteIcon`, or `cautionIcon` (which are defined in `Dialogs.h`). The `hascancel` parameter tells `AlertSystem` whether the dialog should have a Cancel button or not. If `hascancel` is false, `AlertSystem` will always return 1; if `hascancel` is true it may return 1 (OK) or 2 (Cancel). The `str1x` and `str1num` parameters specify the ID of a `STR#` and the index into that string list of the main alert text. The `str2x` and `str2num` fields are used internally to look up error codes—just pass 0 for these two parameters. Note that you don't have to worry about whether the dialog will be big enough for your text—it resizes itself based on the text size.

## Window Routines

**CenterWindowOnMain:** Center the given (hidden) window on the main screen.

```
void CenterWindowOnMain (WindowPtr theWindow);
```

This routine centers the given window on the main screen. It is equivalent to the call:

```
CenterWindow(theWindow, &(*GetMainDevice())->gdRect);
```

**UpdateAllWindows:** Process any needed update events in other Premiere windows.

```
void UpdateAllWindows (void);
```

This routine causes Premiere windows that need it to be updated. Use this when you're doing a custom dialog filter for a movable modal dialog so that when the window is moved, Premiere windows that are behind your dialog will be updated.

**MakeWindowForFile:** Spawn a new window for the given file and file type.

```
void MakeWindowForFile (
 OSType fileType,
 long reserved,
 FSSpec *theFile);
```



This routine spawns a new window for the file specified in theFile, given it's file type fileType. The parameter reserved should be set to nil. Premiere looks up the given file type in its cross-reference list and creates a window of the appropriate type for the file. Use this routine to make your plug-ins act like Premiere's—if Premiere spawns a window after a certain operation, you should too.

**MakeWindowForTextFile:** Spawn a new window for the given TEXT file.

```
void MakeWindowForTextFile (FSSpec *theFile);
```

This routine spawns a new TEXT editor window. The file specified by theFile must be a TEXT file and be less than 32K in length. It is appropriate, for instance, for EDL export modules to open an EDL text editor window showing the EDL that has just been generated.

## Graphics Routines

**SafeNewGWorld:** Bug-free version of NewGWorld.

```
short SafeNewGWorld (
 GWorldPtr *theworld,
 short depth,
 Rect *box,
 CTabHandle ctab,
 GDHandle gdev,
 GWorldFlags flags);
```

This routine is a replacement for NewGWorld that takes the exact same parameters but works around a 32-bit QuickDraw bug involving inverse tables. Use it instead of NewGWorld.

**BetterNewGWorld:** Simplified version of NewGWorld.

```
short BetterNewGWorld (
 GWorldPtr *theworld,
 short depth,
 Rect *box,
 short flags);
```

This routine is a replacement for NewGWorld that takes fewer parameters. It is useful when you are going to create either 24/32-bit GWorlds or GWorlds with default color tables. The theworld, depth, box parameters correspond to NewGWorld parameters. The flags parameter, however, is Premiere-specific. The values that can be added together to form flags are shown below. They are provided in the Premiere headers.

```
enum {
 gwFourPlanes = 1, // Set pixmap cmpCount to 4
 gwKeepLocal = 2, // Allocate in local memory
 gwLockPixels = 4, // Lock down the pixels
 gwExtraRAM = 8 // Leave extra memory around
};
```

The gwFourPlanes flag will make a four-component (ARGB) GWorld. The gwKeepLocal flag keeps the GWorld in main memory (rather than on some video card's GWorld memory). The gwLockPixels flag tells

BetterNewGWorld to go ahead and do a LockPixels call on the GWorld before returning it. The gwExtraRAM flag tries to allow for an amount of extra memory after the allocation of the GWorld equal to that GWorld's size. That is, allocating a 640 x 480 x 32-bit GWorld with the gwExtraRAM flag set, BetterNewGWorld will only allocate the 1.2 megabyte GWorld if there will be 1.2 megabytes free afterwards. Note that for best performance in the average case, you'll probably benefit from setting the gwKeepLocal flag when you call BetterNewGWorld.

BetterNewGWorld calls SafeNewGWorld to create the GWorld and returns SafeNewGWorld's error.

**SetFont:**

Set up the current port's text parameters from a standard template.

```
void SetFont (long which);
```

This routine provides a way to get to some common text-parameter combinations easily and in an internationally localizable way. SetFont looks up the font ID and size associated with the value *which* and makes TextFont and TextSize calls. The text face and mode are left unchanged. Use this routine instead of hard-coding font names or font IDs for localizability. Below is a table of available font/size combinations and their associated constants. These constants are provided in the Premiere header files.

**Table 0-4: SetFont Constants**

| Constant      | Description                     |
|---------------|---------------------------------|
| fontGeneva9   | Geneva 9-point                  |
| fontGeneva12  | Geneva 12-point                 |
| fontChicago12 | Chicago 12-point                |
| fontAbout     | Built-in font used in about box |

**OffscreenBox:**

An offscreen version of TextBox.

```
void OffscreenBox (
 StringPtr str,
 long size,
 Rect *box,
 short style);
```

This routine is just like TextBox except that it draws into an offscreen bitmap and blits to the screen. The parameters *str*, *size*, *box*, and *style* correspond to the parameters of TextBox. The only restriction is that the rectangle pointer *box* must be  $\leq 512$  pixels wide and  $\leq 256$  high or it will be clipped. The main benefit of OffscreenBox is that it prevents flicker when drawing text.

**Color82RGB:**

Converts a Color8 structure to a RGBColor.

```
void Color82RGB (
 Color8 *incolor,
 RGBColor *outcolor);
```

This routine converts the Color8 structure pointed to by `incolor` to the RGBColor structure pointed to by `outcolor`. Color8 structures are useful because they match the way RGB data is actually stored in a 32-bit GWorld.

**RGB2Color:**

Converts a Color8 structure to a RGBColor.

```
void RGB2Color8 (
 RGBColor *incolor,
 Color8 *outcolor);
```

This routine converts the RGBColor structure pointed to by `incolor` to the Color8 structure pointed to by `outcolor`. Color8 structures are useful because they match the way RGB data is actually stored in a 32-bit GWorld.

**DitherBox:**

Converts a Color8 structure to a RGBColor.

```
void DitherBox (
 DialogPtr thedlg,
 Rect *box,
 RGBColor *thecolor);
```

This routine draws a rectangle of size `box` in the color specified by `thecolor` into a preallocated 32-bit GWorld, then blits it into the rectangle `box` in the port specified by `thedlg` in `ditherCopy` mode. This routine is useful for drawing color swatches (color samples in a dialog box, for instance), because the color is accurately represented even on screens with less than 24-bit color.

**Time Code Routines****Time2Str:**

Convert a frame number to a formatted time code string for a given frame rate.

```
void Time2Str (
 long frame,
 StringPtr str,
 short fps,
 short flags);
```

This routine converts the frame number `frame` into a formatted time code string, storing it into the buffer `str` as a Pascal string. The `fps` parameter indicates the frame rate in frames per second. For drop-frame time code, set the high bit of the `fps` parameter. The `flags` parameter is a combination of the following flags (which are defined in the Premiere headers):

```
enum
{
 tsDelta = 0x01,
 tsHours = 0x02,
 tsOneHour = 0x04
};
```

Table 0-5: Time2Str Flags

| Flag      | Value | Description                                   |
|-----------|-------|-----------------------------------------------|
| tsDelta   | 0x01  | Prepend a “Δ” character to the output string. |
| tsHours   | 0x02  | Format: HH:MM:SS:FF                           |
| tsOneHour | 0x04  | Format: H:MM:SS:FF                            |

Time2Str doesn’t use “;” characters to denote drop-frame time code—it always uses colons. If you want semicolons, use FormatTimeCode, described below.

**Str2Time:**

Turn a formatted time code string into a frame number.

```
long Str2Time (
 StringPtr str,
 short len,
 short fps);
```

This routine converts the formatted time code text pointed to by str of length len to a frame number at the frame rate specified by fps. For drop-frame time code, set the high bit of the fps parameter. Note that str is **not** a Pascal string, but a buffer pointer, and the len parameter specifies its length.

**ParseTimecode:**

Turn a formatted time code string into a BIN\_TC structure.

```
void ParseTimecode (
 StringPtr src,
 BIN_TC *timecode);
```

This routine parses the formatted time code string str and writes the corresponding BIN\_TC structure into the BIN\_TC pointed to by timecode. The BIN\_TC structure is provided in the Premiere header files. ParseTimeCode knows that time code strings that use semicolon characters to delimit the fields are in drop-frame format, and allows an optional frame rate specifier in square brackets (if no frame rate is specified, 30 is assumed).

**FormatTimecode:**

Turn a formatted time code string into a frame number.

```
void FormatTimecode (
 BIN_TC *timecode,
 StringPtr str);
```

This routine formats the BIN\_TC structure pointed to by timecode into a Pascal string and places it in str. If the BIN\_TC structure specifies drop frame time code, then the output string will have semicolon delimiters. If the BIN\_TC specifies a frame rate other than 30, the output string will have a “[nn]” frame rate specifier at the end.

**StrToBin:** Turn a formatted time code string into a BIN\_TC structure.

```
void StrToBin (
 StringPtr str,
 BIN_TC *b,
 short df,
 short ntsc);
```

This routine parses the formatted time code string *str* and writes the corresponding BIN\_TC structure into the BIN\_TC pointed to by *b*. The *df* parameter should be DROP\_FRAME for drop-frame, NON\_DROP\_FRAME for non-drop-frame. The *ntsc* field can have the values TC\_PAL for 25 frames-per-second, TC\_NTSC for 30 frames-per-second, or TC\_FILM for 24 frames-per-second. These constants and the BIN\_TC structure are provided in the Premiere header files.

**BinToStr:** Turn a BIN\_TC structure into a formatted time code string.

```
void BinToStr (
 BIN_TC *b,
 StringPtr str);
```

This routine converts the BIN\_TC structure pointed to by *b* into a formatted time code string and stores it as a Pascal string in the buffer *str*.

**StrToBcd:** Turn a formatted time code string into a SMPTE structure.

```
void StrToBcd (
 StringPtr str,
 SMPTE *bcd,
 short frameRate);
```

This routine parses the formatted time code string *str* and writes the corresponding SMPTE structure into the SMPTE pointed to by *bcd*. The *frameRate* field can have the values TC\_PAL for 25 frames-per-second, TC\_NTSC for 30 frames-per-second, or TC\_FILM for 24 frames-per-second. These constants and the SMPTE structure are provided in the Premiere header files.

**BcdToStr:** Turn a SMPTE structure into a formatted time code string.

```
void BcdToStr (
 StringPtr str,
 SMPTE *bcd);
```

This routine converts the SMPTE structure pointed to by *bcd* into a formatted time code string and stores it as a Pascal string in the buffer *str*.

**BcdToBin:** Turn a SMPTE structure into a BIN\_TC structure.

```
void BcdToBin (
 SMPTE *t,
 BIN_TC *b,
 short frameRate);
```

This routine converts the SMPTE structure pointed to by *t* into a BIN\_TC structure and stores it in the BIN\_TC

pointed to by *b*. The *frameRate* field can have the values TC\_PAL for 25 frames-per-second, TC\_NTSC for 30 frames-per-second, or TC\_FILM for 24 frames-per-second. These constants and structures are provided in the Premiere header files.

**BinToBcd:** Turn a BIN\_TC structure into a SMPTE structure.

```
void BinToBcd (
 BIN_TC *b,
 SMPTE *t,
 short dur);
```

This routine converts the BIN\_TC structure pointed to by *b* into a SMPTE structure and stores it in the SMPTE pointed to by *t*. The *dur* parameter, if true, tells BinToBcd that *b* represents a duration, rather than a time. This only matters for NTSC drop-frame. If *dur* is true, no validation is performed, but if *dur* is false (that is, *b* represents a location in time), BinToBcd makes that the output SMPTE doesn't land on a drop frame.

## Data Export Module Utilities

The routines in this category are only valid in a data export ('ExpD') plug-in.

**GetExportMovie:** Return the QuickTime movie handle for the export clip.

```
Movie GetExportMovie (
 DataExportHandle exportData);
```

This routine examines the DataExportRec record referenced by parameter *exportData* and extracts a QuickTime movie handle, if available. If the clip is not a QuickTime movie, GetExportMovie returns nil.

**GetExportFSSpec:** Returns the FSSpec of the clip being exported.

```
void GetExportFSSpec (
 DataExportHandle exportData,
 FSSpec *theSpec);
```

This routine returns, via the reference parameter *theSpec*, the FSSpec of the clip being exported by examining the DataExportRec passed to it through parameter *exportData*. You may use this FSSpec to directly access the file that is associated with the clip being exported, or use it to get the clip's name.

**GetExportClipID:** Returns the clipID of the clip being exported.

```
short GetExportClipID (
 DataExportHandle exportData);
```

This routine returns the clip ID of the clip being exported by examining the DataExportRec passed to it through parameter *exportData*.

**GetExportDivisor:** Returns the "clicks divisor" of the clip being exported.

```
short GetExportDivisor (
 DataExportHandle exportData);
```

This routine returns the clicks (600ths of a second) divisor of the clip being exported by examining the

DataExportRec passed to it through parameter exportData. To determine the frame rate for the clip you may perform the following calculation:

```
framesPerSecond = CLICKS / GetExportDivisor(theData);
```

Or, better yet, simply call GetExportFPS, described below.

**GetExportFPS:** Returns the frame rate of the clip being exported.

```
short GetExportFPS (DataExportHandle
 exportData)
```

This routine gives the frame rate in frames/second of the clip being exported by examining the DataExportRec referenced by parameter exportData.

**GetRate:** Given some audio flags, returns some common sample rates.

```
long GetRate (short flags)
```

This routine given some Premiere audio flags as the flags parameter, returns a sample rate in integer samples per second. GetRate knows how to handle the gaDropFrame flag. Following is a table of most of the possible results from GetRate:

**Table 0-6: GetRate Flags**

| Flags                 | Output |
|-----------------------|--------|
| ga5kHz                | 5563   |
| ga5kHz + gaDropFrame  | 5558   |
| ga11kHz               | 11127  |
| ga11kHz + gaDropFrame | 11116  |
| ga22kHz               | 22254  |
| ga22kHz + gaDropFrame | 22232  |
| ga44kHz               | 44100  |
| ga44kHz + gaDropFrame | 44056  |
| ga48kHz               | 48000  |
| ga48kHz + gaDropFrame | 47952  |

Notice that the drop-frame versions of the sample rates are simply 0.1% smaller than their non-drop counterparts. Since Premiere 4.2 can handle any sample rate, you may wish to simply perform this calculation yourself. GetRate only handles the common rates.

## EDL Export Module Utilities

The routines in this category are used by EDL export ('ExpM') plug-ins. The block routines pertain to the project data block list passed to EDL export modules. For more information on the format of this data, see the chapter [EDL Export Modules](#).

**Important!** The block routines below (*NextBlock*, *CountTypeBlocks*, *FindBlock*, *GetBlock*, and *ExtractBlockData*) are not exported by Premiere. The source code for these routines, however, is provided in "Generic EDL.c".

**GetWipeCodes:** Return the current set of user wipe code identifiers.

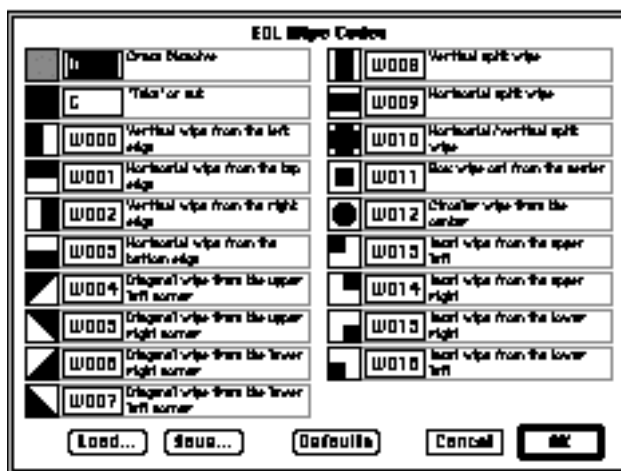
```
void GetWipeCodes (long *codes);
```

This routine returns, via the reference parameter `codes`, Premiere's current set of wipe codes as saved in the preference file. The user may modify these codes through the EDL Wipe Codes dialog (see `EditWipeCodes` below). The `codes` parameter must point to an array of 20 longs.

**EditWipeCodes:** Return the current set of user wipe code identifiers.

```
void EditWipeCodes (void);
```

This routine puts up the EDL Wipe Codes dialog, shown below. The wipe code values are saved internally—you access them by calling `GetWipeCodes`, described above.



**NextBlock:** Advance the given `BlockRec` pointer to the next block.

```
void NextBlock (BlockRec **srcBlock);
```

This routine takes the address of a `BlockRec` pointer as the parameter `srcBlock`. It adds the size of the block `srcBlock` currently points to `srcBlock` and stores the result back in `*srcBlock`.

**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. The following table details the routine's behavior with different combinations of parameters:

**Table 0-7: FindBlock Parameters**

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

**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.

# 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 {
 shortcount; // Number of routines
 shortreserved[14];

 StretchBitsPtrStretchBits;
 DistortPolygonPtrDistortPolygon;
 PolyToPolyPtrMapPolygon;
 AudStretchPtrAudioStretch;
 AudMixPtrAudioMix;
 AudSumPtrAudioSum;
 AudLimitPtrAudioLimit;
 DistortFixedPolygonPtrDistortFixed;
 FixedToFixedPtrFixedToFixed;
 longImageKey;
 ResamplePtrResample;
 AudioMungePtrAudioMunge;

 longunused[1];
} BottleRec;
```

The count field specifies how many bottleneck routines follow the reserved field. As of Adobe Premiere 4.2, count is 12. The reserved and unused fields are reserved for future use by Adobe Systems and are currently 0.

For backwards-compatibility, the Power Macintosh version of Adobe Premiere also keeps a parallel “UPP version” of the bottleneck record around. It is only passed to 68K plug-ins running under Power Macintosh Premiere (the alternate record does not exist under 68K Premiere).

```
typedef struct {
 shortcount; // number of routines
 shortreserved[14];

 UniversalProcPtrStretchBits;
 UniversalProcPtrDistortPolygon;
 UniversalProcPtrMapPolygon;
 UniversalProcPtrAudioStretch;
 UniversalProcPtrAudioMix;
 UniversalProcPtrAudioSum;
 UniversalProcPtrAudioLimit;
 UniversalProcPtrDistortFixed;
 UniversalProcPtrFixedToFixed;
 longImageKey;
 UniversalProcPtrResample;
 UniversalProcPtrAudioMunge;

 longunused[1];
} UPPBottleRec;
```

The “proc info” macros for these routines are included in the Premiere headers. While it would be possible for a Power Mac plug-in to call bottlenecks through the UPP record (using CallUniversalProc) it only incurs extra overhead. This overhead can be avoided by simply calling through the PowerPC procedure pointers in the bottleneck record provided. The important thing to remember is that Premiere knows whether it (Premiere) is a 68K application or a Power Mac application and whether your plug-in is a 68K plug-in or a Power Mac plug-in. So Premiere always passes you the bottleneck record that is appropriate for the situation.

Most Premiere plug-ins have a standard record associated with the plug-in type. Usually an appropriate bottleneck record is provided in that record. However, you can always get the bottleneck record by making the following call:

```
{
BottleRec *bottles;

bottles = (BottleRec *)GetAGlobal(gBottleNecks);
.
.
.
}
```

## The Bottleneck Routines

### StretchBits:

The StretchBits routine works very much like CopyBits. It has two enhancements, and a few restrictions.

```
pascal void StretchBits (
 BitMap *srcBits,
 BitMap* dstBits,
 Rect *srcRect,
 Rect *dstRect,
 short mode,
 RgnHandle maskRgn);
```

StretchBits takes exactly the same parameters as CopyBits. The two restrictions are that it only works on 32-bit deep GWorlds, and that it does not handle mask regions. If either of these conditions are not met, StretchBits calls CopyBits. Its enhancements are that it properly processes the alpha channel during the copy (whereas CopyBits clears the alpha channel in any pixel

it generates), and that when the destination is larger than the source, it performs bilinear interpolation to generate the destination. The latter feature provides smoothed enlargement of source material where CopyBits would pixelate the resulting image.

**DistortPolygon:**

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

```
pascal void DistortPolygon (
 GWorldPtr src,
 GWorldPtr dest,
 Rect *srcbox,
 Point *dstpts);
```

The src and dest GWorlds must both be 32 bits deep. The srcBox parameter specifies a rectangular area within the src GWorld. The dstpts parameter should be set to point to an array of four Points which describe a four-point polygon in the destination GWorld. 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 GWorld and maps it into a four-point polygon in a destination GWorld.

```
pascal void MapPolygon (
 GWorldPtr src,
 GWorldPtr 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.

```
typedef pascal void (*AudStretchPtr) (
 Ptr src,
 long srclen,
 Ptr dest,
 long destlen,
 short 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 1.0 and 2.0 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,
 short 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`.

**AudioLimit:** The `AudioLimit` routine clips the source audio buffer while copying to a destination buffer.

```
pascal void AudioLimit (
 Ptr src,
 Ptr dest,
 long width,
 short 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.

```
pascal void DistortFixed (
 GWorldPtr src,
 GWorldPtr 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.

```
pascal void FixedToFixed (
 GWorldPtr src,
 GWorldPtr 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 Globals

Because of the modular nature of Adobe Premiere, Premiere's globals are not stored in the typical A5-relative fashion. Instead, Premiere accesses globals symbolically through two accessor routines, `GetAGlobal` and `SetAGlobal`, which are documented in the Premiere Specific Routines section of the chapter [The Utility Library](#).

## Look But Don't Touch!

The global variables listed below are the same ones Adobe Premiere depends upon for proper operation. The following are crucial: do not use `SetAGlobal` to modify the values of variables not listed in the Read/Write section below and do not depend upon the values of globals that are not specifically documented here (even though they might be listed in the Premiere headers). Adobe reserves the right to change the value or meaning of variables that are not specifically documented here without notice.

## Read/Write Globals

The variables listed below can be used by your plug-in modules on a read-write basis, within a single call to your plug-in.

**gSaveRef:** long

This global is a location that can be used to save a long word value to be read within some routine that cannot see your variables. The example below shows how to use `gSaveRef` to pass a string pointer into a user item procedure. The value you give to `gSaveRef` is completely up to you—for instance, if you need to pass more than one item of information, create a record and set `gSaveRef` to the address of the record.

```
pascal void StringUserItemProc (WindowPtr window,
 short item)
{
 Rect box;
 FontInfo info;
 StringPtr theString;

 theString = (StringPtr)GetAGlobal(gSaveRef);
 GetDRect(window, item, &box);
 GetFontInfo(&info);
 MoveTo(box.left, box.top + info.ascent);
 DrawString(theString);
}
```

```

void MyDialog (void)
{
 unsigned char myString[] = "\pZippity do da";
 DialogPtr dialog;
 short item;

 SetAGlobal(gSaveRef, (long)myString);
 dialog = GetNewDialog(kMyDLOGID, nil, (WindowPtr)-1);
 UserItem(dialog, kStringUserItem,
 StringUserItemProc);
 ShowModal(dialog);

 do {
 PrModalDialog(modalfilter, &item);
 } while (item != 1);

 DisposeModal(dialog);
}

```

- gExportRef:** long  
This global is just like gSaveRef but is specifically for use by Export ('ExpD' and 'ExpM') modules. Other (non-export) modules should avoid using this global to avoid conflicts.
- gCompileErr:** long  
New in Premiere 4.2, a plug-in can set this global if it couldn't render and wants Premiere to stop compiling. Set it with the error the plug-in got while rendering (most likely out of memory).

## Read Only Globals

The variables listed below can be read by your plug-in modules. Do *not* modify the values of these global variables.

- gSysVersion:** short  
This global contains the system software version in 8.8 BCD. For example, if the current machine is running System 7.1, GetAGlobal(gSysVersion) will return 0x0710.
- gResFileNum:** short  
This global contains the resource file reference number of Premiere's application resource file.
- gStillDefault:** long  
This global contains the default still image duration as a number of frames at the current frame rate.
- gPluginVRefNum:** short  
This global contains the vRefNum of the current plug-ins folder.



|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>gPluginDirID:</b> | long<br>This global contains the dirID of the current plug-ins folder.                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>gWorkVRefNum:</b> | short<br>This global contains the vRefNum of the location for temporary files and newly captured movies.                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>gWorkDirID:</b>   | long<br>This global contains the dirID of the location for temporary files and newly captured movies.                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>gTempVRefNum:</b> | short<br>This global contains the vRefNum of the Premiere application.                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>gTempDirID:</b>   | long<br>This global contains the dirID of the Premiere application.                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>gPrefVRefNum:</b> | short<br>This global contains the vRefNum of the preferences file location.                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>gPrefDirID:</b>   | long<br>This global contains the dirID of the preferences file location.                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>gPrintRec:</b>    | THPrint<br>This global contains a handle to Premiere's print record. You can use this print record if your plug-in does any printing.                                                                                                                                                                                                                                                                                                                                                                           |
| <b>gWStrip:</b>      | GWorldPtr<br>This global contains a pointer to a locked, 32-bit deep "strip buffer" GWorld. The width of this GWorld is determined by the current "maximum image size" preference which is available from the global gMaxWidth described below. The height of the strip GWorld is fixed at kBandHeight, a constant that is defined in the Premiere headers. You can use this buffer for banding images.<br><i><b>Important!</b> This GWorld is created and locked at initialization time. Do not unlock it!</i> |
| <b>gOneBitWorld:</b> | GWorldPtr<br>This global contains a pointer to a locked, 1-bit deep 512 x 256 utility GWorld. You can use this buffer for offscreen black-and-white drawing.<br><i><b>Important!</b> This GWorld is created and locked at initialization time. Do not unlock it!</i>                                                                                                                                                                                                                                            |

**gRGB2Y:** unsigned char \*

This global contains three consecutive Y (luminance) tables as follows:

**Table 4–1: gRGB2Y Luminance Tables**

| Indices   | Description                             |
|-----------|-----------------------------------------|
| 0...255   | Luminance value of red values 0...255   |
| 256...511 | Luminance value of green values 0...255 |
| 512...767 | Luminance value of blue values 0...255  |

You can use this table to determine the Y (luminance) of a given RGB pixel value. Here's an example of how you might use this table (although this would be an inefficient approach for a lot of pixels):

```
unsigned char Color8Y (Color8 *pixel)
{
 unsigned char *lumaTable;
 unsigned char luma;

 lumaTable = (unsigned char *)GetAGlobal(gRGB2Y);
 luma = lumaTable[pixel->red];
 luma += lumaTable[pixel->green + 256];
 luma += lumaTable[pixel->blue + 512];

 return(luma);
}
```

**gRGB2UV:** short \*

This global contains three consecutive U tables followed by three consecutive V tables (which together describe chrominance) as follows:

**Table 0–2: gRGB2UV Chrominance Tables**

| Indices     | Description                     |
|-------------|---------------------------------|
| 0...255     | U value of red values 0...255   |
| 256...511   | U value of green values 0...255 |
| 512...767   | U value of blue values 0...255  |
| 768...1023  | V value of red values 0...255   |
| 1024...1279 | V value of green values 0...255 |
| 1280...1535 | V value of blue values 0...255  |

You can use this table to determine the U and V of a given RGB pixel value. Here's an example of how you might use this table (although this would be an inefficient approach for a lot of pixels):

```

void Color8UV (Color8 *pixel, short *outU, short *outV)
{
 short *uvTable;
 short u, v;

 uvTable = (short *)GetAGlobal(gRGB2UV);
 u = uvTable[pixel->red];
 u += uvTable[pixel->green + 256];
 u += uvTable[pixel->blue + 512];
 v = uvTable[pixel->red + 768];
 v += uvTable[pixel->green + 1024];
 v += uvTable[pixel->blue + 1280];

 *outU = u;
 *outV = v;
}

```

**gDecimalPt:**

char

This global contains the decimal separator from the record returned from IUGetIntI(0). When you have the urge to place a literal '.' in a string, use the value of gDecimalPt instead.

**gMultiply:**

unsigned char \*

This global contains a pointer to a 64K 8 x 8 multiply table. You can use this table to perform fast 8-bit multiplication operations. To multiply two 8-bit values, place the first byte in the high 8-bits of an unsigned short, place the second byte in the low 8-bits of the unsigned short. Use the unsigned short as an index into the gMultiply table. The result is the unsigned char at that location. Here's an example (but note that this table is often more useful from assembly):

```

{
 unsigned char *multiply;
 unsigned char result, byte1, byte2;
 unsigned short index;

 multiply = GetAGlobal(gMultiply);

 byte1 = 6;
 byte2 = 20;
 index = (byte1 << 8) | byte2;
 result = multiply[index]; // result gets 120
}

```

**gHasOutline:**

long

This global contains true if TrueType fonts are available, false if not.

**gBottleNecks:**

BottleRec \*

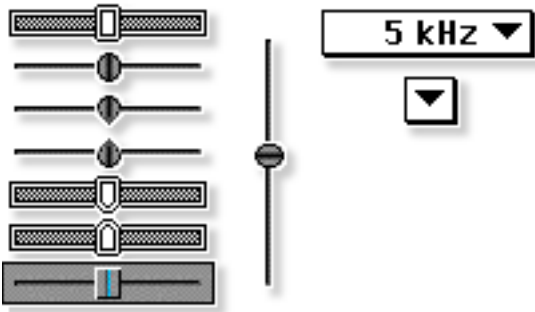
This global contains a pointer to Premiere's bottleneck record. With Power Mac Premiere this record will contain function pointers suitable for direct use by native Power Mac plug-ins. With 68K Premiere this record will contain function pointers suitable for direct use by 68K plug-ins. Most plug-ins are passed an appropriate bottleneck record in their respective plug-in data records, but routine that don't have access to that data record can access the bottlenecks via the global. See also gUPPBottleNecks below.

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>gUPPBottleNecks:</b>  | UPPBottleRec *                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|                          | This global, like gBottleNecks contains a pointer to Premiere's bottleneck record except that it is filled with UPPs rather than Power Mac function pointers. This is suitable for use by 68K plug-ins running under Power Mac Premiere. Most plug-ins are passed an appropriate bottleneck record in their respective plug-in data records, but routine that don't have access to that data record can access the bottlenecks via the global. See also gBottleNecks above. |
|                          | <i><b>Important!</b> This global is nil under 68K Premiere.</i>                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>gMaxWidth:</b>        | long                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                          | This global contains the maximum frame width set in the current preferences.                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>gMaxHeight:</b>       | long                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                          | This global contains the maximum frame height set in the current preferences.                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>gLockStillAspect:</b> | long                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                          | This global contains the default state of the "lock aspect" attribute for still images set in the current preferences.                                                                                                                                                                                                                                                                                                                                                      |
| <b>gBit16OK:</b>         | long                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                          | This global contains true if Sound Manager 3.0 or later is installed.                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>gSysWindow:</b>       | WindowPtr                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|                          | This global contains a pointer to the "HIDE window," which provides a variety of services to 'HDLR' and 'Draw' plug-in modules. See the <a href="#">Other Plug-Ins</a> chapter for more information.                                                                                                                                                                                                                                                                        |
| <b>gTicks:</b>           | long                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                          | This global is equivalent to a call to TickCount() (in fact, doing GetAGlobal(gTicks) does exactly that). See also gTickNumer and gTickDenom below.                                                                                                                                                                                                                                                                                                                         |
| <b>gTickNumer:</b>       | long                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                          | This global contains the numerator of a fraction that exactly describes the duration of a Macintosh "tick." A tick is <i>not</i> exactly 1/60th of a second, but is in fact 1/60.14th of a second. Thus GetAGlobal(gTickNumer) returns 6014. See also gTickDenom.                                                                                                                                                                                                           |
| <b>gTickDenom:</b>       | long                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                          | This global contains the denominator of a fraction that exactly describes the duration of a Macintosh "tick." A tick is <i>not</i> exactly 1/60th of a second, but is in fact 1/60.14th of a second. Thus GetAGlobal(gTickDenom) returns 100. See also gTickNumer.                                                                                                                                                                                                          |

- gqd:** QDGlobals \*
- This global contains a pointer to Premiere's QuickDraw globals. You can use this record to access the handy arrow cursor and patterns therein.
- gQTVers:** short
- This global contains the QuickTime version in 8.8 BCD. For example, if the current machine is running QuickTime 1.6.1, GetAGlobal(gQTVers) will return 0x0161.
- gHaveDragAndDrop:** long
- This global contains true if the Macintosh Drag & Drop extension is present or false if not.

# 5 CDEFs

Premiere has several built-in control definition procedures (CDEFs) that you can use in your dialogs to help your plug-in have a consistent “look-and-feel” with the rest of Premiere. Here’s a picture to give you an idea of what’s available:



## Control Hits

Because of a “feature” of the Dialog Manager, controls that do their own tracking (as do all of Premiere’s controls) do not cause ModalDialog to return a “hit” for dialog control manipulations. For instance, if you had a popup menu in your dialog using Premiere’s popup CDEF as item number 5, and the user changes the value of the popup, you will *not* get item 5 back from ModalDialog—no hit is registered. If you need to know immediately when a control’s value has been modified, use a dialog filter procedure to manually track your controls. An example is shown below:

```
pascal char MyFilter (DialogPtr dialog, EventRecord *event, short *hit)
{
 Point where;
 ControlHandle control;
 short which, oldValue, part;
 char result = false;

 SetPort(dialog);
 result = modalfilter(dialog, event, hit);
 if (!result && event->what == mouseDown)
 {
 where = event->where;
 GlobalToLocal(&where);
 which = FindDItem(dialog, where) + 1;
 if (which != 0 && FindControl(where, dialog, &control))
 {
 oldValue = GetCtlValue(control);
 part = TrackControl(control, where, nil);
 if (oldValue != GetCtlValue(control))
 {
 *hit = which;
 result = true;
 }
 else if (part != 0)
 {
 *hit = which;
 result = true;
 }
 }
 }
}
```

```

else
{
 *hit = 0;
 result = true;
}
}
}
return(result);
}

```








The modal filter routine used above is documented in the chapter [The Utility Library](#).

## Control Types

### Horizontal Sliders

You can create a variety of horizontal sliders with CDEF 136. The base ProclD for this CDEF is 2176.


Table 0-3: Horizontal Sliders

| Variation | ProclD | Description                                                                                                                                                                                                                                            | Slider                                                                                |
|-----------|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| 0         | 2176   | This variation provides a slider with a rectangular thumb. This is the most common type of horizontal slider used in Premiere. Adobe encourages you to use ProclD 2176 rather than 2177 unless your specific situation warrants otherwise.             |  |
| 1         | 2177   | This variation provides a slider with an oval thumb and a thin slide bar. This slider is no longer used in Premiere or its plug-ins, but you can use it if it fits your application.                                                                   |  |
| 2         | 2178   | This variation provides a slider with an oval down-pointing thumb and a thin slide bar. This slider is no longer used in Premiere or its plug-ins, but you can use it if it fits your application.                                                     |  |
| 3         | 2179   | This variation provides a slider with an oval up-pointing thumb and a thin slide bar. This slider is no longer used in Premiere or its plug-ins, but you can use it if it fits your application.                                                       |  |
| 4         | 2180   | This variation provides a slider with a rectangular down-pointing thumb. This slider is used by Premiere in the Transparency dialog. Adobe encourages you to use ProclD 2180 rather than 2178 above unless your specific situation warrants otherwise. |  |
| 5         | 2181   | This variation provides a slider with a rectangular up-pointing thumb. This slider is used by Premiere in the Transparency dialog. Adobe encourages you to use ProclD 2181 rather than 2179 above unless your specific situation warrants otherwise.   |  |
| 8         | 2188   | This variation provides a slider with a full frame and rectangular thumb. This slider is used by Premiere in clip windows for "movie location."                                                                                                        |  |

## Vertical Sliders

You can create vertical sliders with CDEF 134. The base ProclD for this CDEF is 2144 with a single variation.


**Table 0–4: Vertical Sliders**

| Variation | ProclD | Description                                                                                                                                                                                   | Slider                                                                              |
|-----------|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| 1         | 2145   | This variation provides a vertical slider with an oval thumb and a thin slide bar. This slider is no longer used in Premiere or its plug-ins, but you can use it if it fits your application. |  |


## Popup Menus

You can create popup menus with CDEF 137. The base ProclD for this CDEF is 2192. Premiere’s popup CDEF, while somewhat less flexible, is much easier to use than the Apple Communications Toolbox-style CDEF.

**Table 0–5: Popup Menus**

| Variation | ProclD | Description                                   | Slider                                                                                |
|-----------|--------|-----------------------------------------------|---------------------------------------------------------------------------------------|
| 0         | 2192   | This variation provides a a basic popup menu. |  |

The MENU ID is taken from the control’s refCon field, and the initially checked item is taken from the control’s value field. To determine which item is chosen, simply call GetCtlValue on the control.

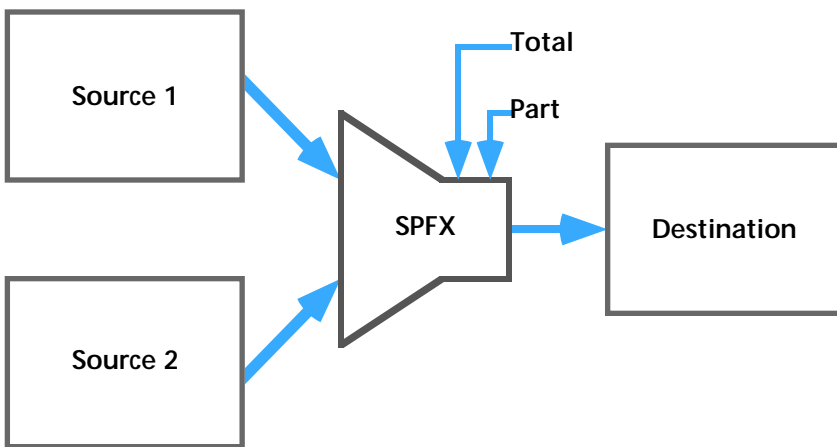
If the width of the control bounds rectangle is less than 25 pixels, a “popup bud” is created. 

To get the correct look for a popup bud, make the control bounding box 20 pixels wide.



# 6 Transitions

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



All three GWorlds 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, then storing them in compressed 'ICNc' resources in your transition's resource fork. Consequently, if the code for a transition is changed, any ICNc resources that Premiere may have stored in your transition should be deleted (see the example transition modules).

Transition modules are stored in files of type 'SPFX', with creator 'PrMr'. The name of the file is the name that Premiere will display in the Transitions window to refer to the transition. 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 0-6: Transition Resources**

| Type & ID | Description                                                                          |
|-----------|--------------------------------------------------------------------------------------|
| FXvs 1000 | A two-byte version number stored as a short integer.                                 |
| TEXT 1000 | A textual description of what the transition does.                                   |
| Fopt 1000 | A resource that describes what options the transition supports.                      |
| FXDF -1   | An OSType that gives a mapping to one of the standard SMPTE wipes.                   |
| SPFX 1000 | The transition code itself (68K). The entry point is the first byte of the resource. |
| SPFx 1000 | The transition code itself (PowerPC). The entry point is specified by the PEF.       |

## FXvs 1000

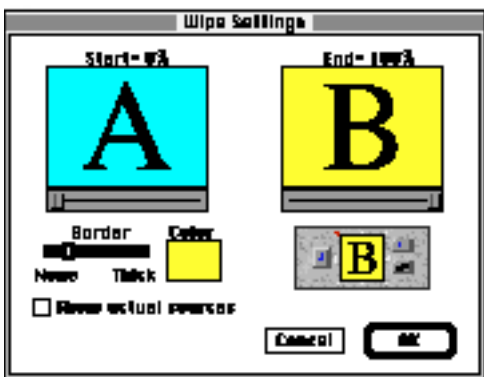
This resource is two bytes in length and gives the version of the module interface. The current version is \$0001.

## TEXT 1000

This resource contains a plain text description of the transition and is displayed beside the animation 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 in Rez format, along with some bit values you'll use with the definition:

```

type 'Fopt'
{
 byte; // Valid corners mask
 byte; // Initial corners
 byte; // Has custom, in pairs, time invariant
 byte No = 0, Yes = 1; // Exclusive?
 byte No = 0, Yes = 1; // Reversible?
 byte No = 0, Yes = 1; // Has edges?
 byte No = 0, Yes = 1; // Has start point?
 byte No = 0, Yes = 1; // Has end point?
};

#define bitTop 0x01 // Corner bits
#define bitRight 0x02
#define bitBottom 0x04
#define bitLeft 0x08
#define bitUpperRight 0x10
#define bitLowerRight 0x20
#define bitLowerLeft 0x40
#define bitUpperLeft 0x80

#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—first 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—second 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.

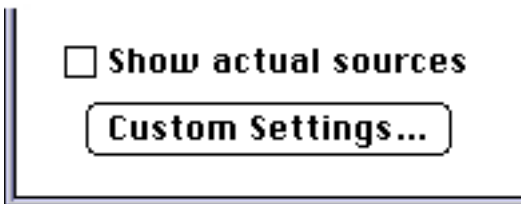
### Fopt—third byte: Bit flags

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—fourth 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.

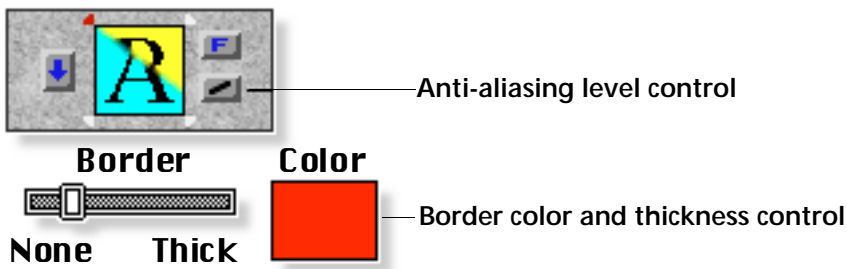
### Fopt—fifth byte: Reversible

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.



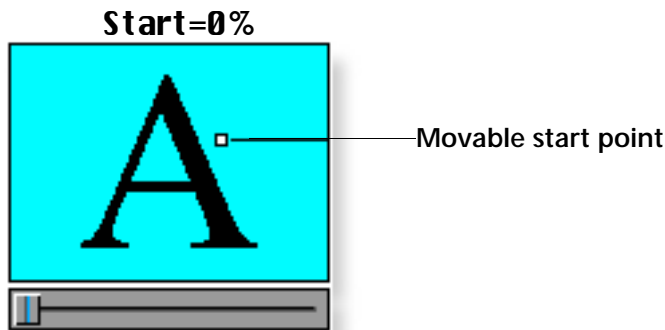
### Fopt—sixth byte: Has edges flag

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.



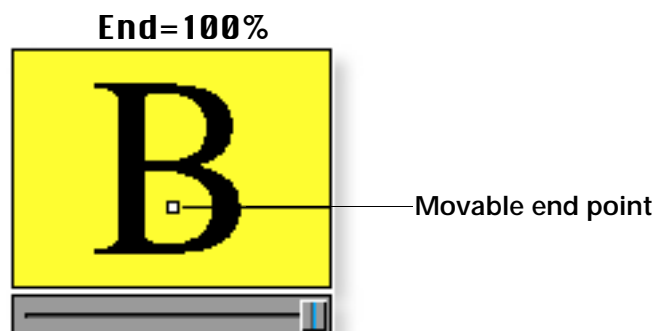
**Fopt—seventh byte: Movable start point flag**

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—eighth byte: Movable end point flag**

The eight 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 0-7: SMPTE Wipe Tags**




| Tag  | SMPTE wipe description            |
|------|-----------------------------------|
| DISS | Cross dissolve                    |
| TAKE | “Take” or cut                     |
| WI00 | Vertical wipe from the left edge  |
| WI01 | Horizontal wipe from the top edge |
| WI02 | Vertical wipe from the right edge |

Table 0-7: SMPTE Wipe Tags

| Tag  | SMPTE wipe description                |
|------|---------------------------------------|
| 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" |

## SPFX/SPFx 1000

These resources contain the code for your transition, SPFX and SPFx containing 68K and PowerPC code respectively. The entry point should be declared like this:

```
pascal short Transition (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 0–8: 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.

**Important!** The destination GWorld must be left with its foreground color set to black and its background color set to white, otherwise subsequent operations with in Premiere will draw incorrectly. At the end of your esExecute selector use this code:

```
.
.
.
SetGWorld((*theData)->destination, nil);
ForeColor(blackColor);
BackColor(whiteColor);

SetGWorld(oldWorld, oldDevice);
break;
.
.
.
```

## 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, 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
 GWorldPtr source1; // Source GWorld 1 (video track A)
 GWorldPtr source2; // Source GWorld 2 (video track B)
 GWorldPtr destination; // Destination GWorld
 long part; // part / total = % complete
 long total;
 char previewing; // In preview mode?
 char arrowFlags; // Flags for direction arrows
 char reverse; // Is transition being reversed?
 char source; // Are sources swapped?
 Point start; // Spatial starting point
 Point end; // Spatial ending point
 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.

### source1

The source1 field is the GWorld 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 GWorld 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 GWorld 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 GWorlds (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.

## 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,
 CGrafPtr 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 `GWorld`. The `theBox` parameter is the destination rectangle in thePort. Finally, the `privateData` parameter is `(*theData)->privateData`.

Note that when a 68K plug-in is called from Power Mac Premiere, the callback field actually contains a UPP rather than a PowerPC procedure pointer.

## 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 0-9: sizeFlags**

| Flag                      | Description                                                  |
|---------------------------|--------------------------------------------------------------|
| <code>gvHalfV</code>      | User has specified half-vertical processing                  |
| <code>gvHalfH</code>      | User has specified half-horizontal processing                |
| <code>gvFieldsEven</code> | User has specified field-based processing, even fields first |
| <code>gvFieldsOdd</code>  | 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 Toolkit comes with source code for three example SPFX modules.

**Additive Dissolve**

Similar to a cross dissolve, this transition adds the two images together using a special QuickDraw transfer mode gradually through the duration of the transition. This is a good example of a basic transition.

**Cross Zoom**

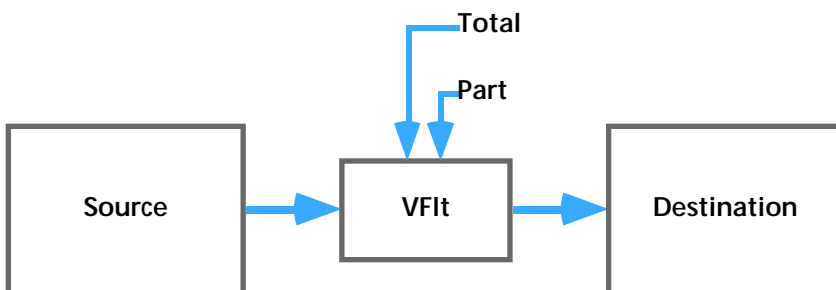
This is the source code for the Cross Zoom transition that ships with Adobe Premiere. It supports movable start- and end-points and demonstrates the use of the StretchBits bottleneck.

**Wipe**

This is a basic wipe transition that works from any of the eight directions. It is a good basis for many region-based shaped wipes.

# 7 Video Filters

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



Both GWorlds 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 are stored in files of type 'VFlt', with creator 'PrMr'. The name of the file is the name that Premiere will display in the filters dialog list. Video filter files contain two kinds of resources which are listed below. Following the table is a detailed description of each resource.

**Table 0-10: Video Filter Resources**

| Type & ID | Description                                                                           |
|-----------|---------------------------------------------------------------------------------------|
| FXvs 1000 | A two-byte version number stored as a short integer.                                  |
| FltD 1000 | An optional time-animation resource describing the format of your settings data blob. |
| VFlt 1000 | The video filter code itself (68K). The entry point is the resource's first byte.     |
| VFIT 1000 | The video filter code itself (PowerPC). The entry point is specified by the PEF.      |

## FXvs 1000

This resource is two bytes in length and gives the version of the module interface. The current version is \$0001.

## FltD 1000

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. 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 our FltD resource would look like:

```
resource 'FltD' (1000, "")
{
 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
};
```

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
#define pdDouble 0x0008 // Interpolate as a double
#define pdFloat 0x0009 // Interpolate as a float
```

## VFit /VFIT 1000

These resources contain the code for your video filter, VFit and VFIT containing 68K and PowerPC code respectively. The entry point should be declared like this:

```
pascal short VideoFilter (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 0–11: Video Filter Selectors**

| 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.

**Important!** The destination GWorld must be left with its foreground color set to black and its background color set to white, otherwise subsequent operations with in Premiere will draw incorrectly. At the end of your esExecute selector use this code:

```
.
.
.
SetGWorld((*theData)->destination, nil);
ForeColor(blackColor);
BackColor(whiteColor);

SetGWorld(oldWorld, oldDevice);
break;
.
.
.
```

### 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, 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 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
 GWorldPtr source; // The source GWorld
 GWorldPtr destination; // The destination GWorld
 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.

### source

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

### destination

The destination field is the GWorld 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 pascal short (*VFilterCallBackProcPtr) (
 long frame,
 CGrafPtr thePort,
 Rect *theBox,
 Handle privateData);
```

The frame parameter is the desired frame, from 0 to total inclusive. The thePort parameter is the destination for the frame, often a locally allocated GWorld. The theBox parameter is the destination rectangle in thePort. Finally, the privateData parameter is (\*theData)->privateData.

Note that when a 68K plug-in is called from Power Mac Premiere, the callback field actually contains a UPP rather than a PowerPC procedure pointer.

## 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. For Premiere 4.2, this field is 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 0-12: sizeFlags**

| 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 deallocate 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 Toolkit comes with source code for a video filter module that you can use as an example of how to write your own.

**Video Noise**

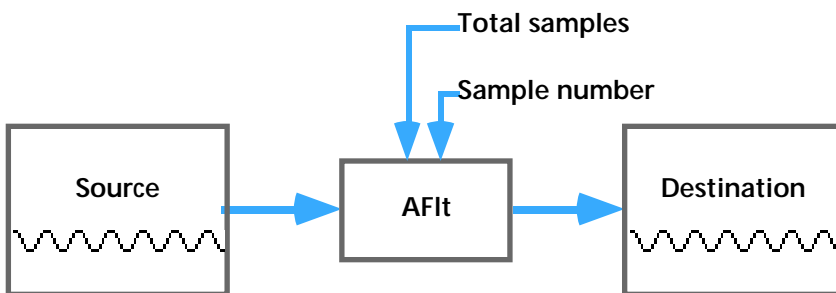
This is the source code for the Video Noise filter that ships with Adobe Premiere. It is a good example of a basic video filter with no settings dialog.

**Burn Time Code**

This is very similar to the source code for the Burn Time Code filter that ships with Adobe Premiere. It gives a good example of a fairly sophisticated settings dialog and uses some of the simpler timecode routines from `UtilLib`.

# 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 are stored in files of type 'AFIt', with creator 'PrMr'. The name of the file is the name that Premiere will display in the filters dialog list. Audio filter files contain two kinds of resources which are listed below. Following the table is a detailed description of each resource.

**Table 0-13: Audio Filter Resources**

| Type & ID | Description                                                                           |
|-----------|---------------------------------------------------------------------------------------|
| FXvs 1000 | A two-byte version number stored as a short integer.                                  |
| FltD 1000 | An optional time-animation resource describing the format of your settings data blob. |
| AFIt 1000 | The audio filter code itself (68K). The entry point is the resource's first byte.     |
| AFIT 1000 | The audio filter code itself (PowerPC). The entry point is specified by the PEF.      |

## FXvs 1000

This resource is two bytes in length and gives the version of the module interface. The current version is \$0001.

## FltD 1000

This optional resource is a variable-length description of your filter's data blob (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 “Start” and “End” settings buttons in the Filter dialog box. When it comes time to filter audio using your filter, Premiere uses the information in your FltD to interpolate the values in your settings structure over time.

For more information about FltD resources, see the description in the chapter [Video Filters](#).

## AFIt/AFIT 1000

These resources contain the code for your audio filter, AFIt and AFIT containing 68K and PowerPC code respectively. The entry point should be declared like this:

```
pascal short AudioFilter (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 black.

The selector can take the following values:

**Table 0–14: Audio Filter Selectors**

| 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, 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 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.

### 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 buffer is preloaded with a certain amount of the clip's audio data (specified by a user preference), already filtered by any previous audio filters in the chain. The default preview duration is 3 seconds. You can process this sound into the destination buffer and use the Sound Manager to loop the destination buffer. This allows your audio filter to perform a "preview audio" function. See the Pan filter example for details about how to do this. 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 also allocated and you can use it as a processing buffer

### 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 a "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 0–15: sampleNum**

| 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 pascal short (*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` parameter is `(*theData)->privateData`.

Note that when a 68K plug-in is called from Power Mac Premiere, the `callback` field actually contains a UPP rather than a PowerPC procedure pointer.

### 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 Macintosh sample rate, rate would contain 22254. This is for informational purposes in case your filter needs 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 audio filter.

### **version**

The version field tells the version of this VideoRecord. For Premiere 4.2, this value should be 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 Toolkit comes with source code for an audio filter module that you can use as an example of how to write your own.

### **Backwards [Audio]**

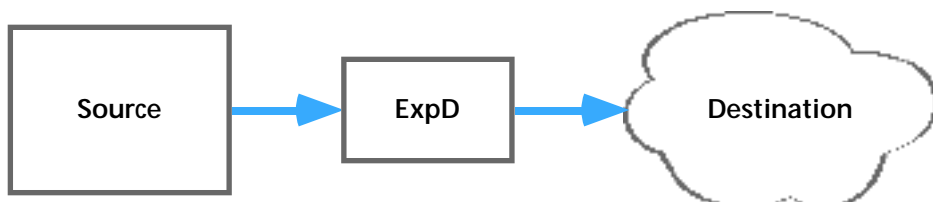
This is the source code for the Backwards [Audio] filter that ships with Adobe Premiere. It is a good example of a basic audio filter with no settings dialog, and shows how to use the callBack procedure to retrieve audio from a different point in the clip.

### **Pan**

This is the source code for the Pan filter that ships with Adobe Premiere. It gives an example of how to implement a "Preview Sound" checkbox in your settings dialog and how to specify an 'FltD' animation resource.

# 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 standard 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.

Export modules are stored in files of type 'ExpD', with creator 'PrMr'. The name of the file is the name that Premiere will display in the Export submenu of the File menu. Data export files contain two kinds of resources which are listed below. Following the table is a detailed description of each resource.

**Table 0-16: Data Export Module Resource IDs**

| Type & ID | Description                                                                      |
|-----------|----------------------------------------------------------------------------------|
| FXvs 1000 | A two-byte version number stored as a short integer.                             |
| FLAG 1000 | A two-byte flags word that tells the capabilities of the data export module.     |
| ExpD 1000 | The data export code itself (68K). The entry point is the resource's first byte. |
| Expd 1000 | The data export code itself (PowerPC). The entry point is specified by the PEF.  |

## FXvs 1000

This resource is two bytes in length and gives the version of the module interface. The current version is \$0001.

## 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 in Rez format, along with some bit values you'll use with the definition:

```
type 'FLAG' {
 integer; // Capability flags
};

#define bitCanDoVideo 0x8000 // This module can export video clips
#define bitCanDoAudio 0x4000 // This module can export audio clips
```

For example, if your export module can export both audio and video, you'd make a FLAG resource containing bitCanDoVideo + bitCanDoAudio.

## ExpD/Expd 1000

These resources contain the code for your data export module, ExpD and Expd containing 68K and PowerPC code respectively. The entry point should be declared like this:

```
pascal short DataExportModule (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 0-17: Data Export Module Selectors**

| 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 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: the markers array is vestigial. Now Premiere supports unnumbered markers. Use the UtilLib.o functions CountClipMarkers and GetClipMarker to access the markers from index 2 on. See the Storyboard Image example source code for details.

### 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 Macintosh sample rate, audrate would contain 22254. 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:

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

The frame parameter is the desired frame in the units defined by the framerate field. For example, you would use (\*theData)->markers[0] to retrieve the frame at the in-point. The thePort parameter is the destination for the frame, usually a locally allocated GWorld, which must be 32-bits deep. The theBox parameter is the destination rectangle in thePort. Finally, the privateData parameter is (\*theData)->privateData.

Note that when a 68K plug-in is called from Power Mac Premiere, the getVideo field actually contains a UPP rather than a PowerPC procedure pointer.

## 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:

```
typedef pascal short (*GetAudCallBack) (
 long second,
 short formatFlags,
 Ptr buffer,
 Handle privateData);
```

The second parameter is the desired second (like second #0, second #1, etc.) The formatFlags parameter specifies the sample format for the retrieved samples. Premiere will perform rate- and format-conversion for you. The following constants may be used in the formation of your formatFlags value:

```
#define aflag5KHz 0x0001 // Predefined sample rates
#define aflag11KHz 0x0002
#define aflag22KHz 0x0004
#define aflag44KHz 0x0008
#define aflagSpecial 0x0040 // Set to get audio at an arbit. rate
#define aflagStereo 0x0100 // Set if you want stereo
#define aflag16Bit 0x0200 // Set if you want 16-bit
#define aflagDropFrame 0x0400 // Set if you want drop-frame
```

The aflagSpecial flag allows you to request audio at an arbitrary sample rate. Set the specialRate field of the DataExportRec to a sample rate in Hz, then call the getAudio callback with the aflagSpecial flag set to have Premiere resample the clip's audio to your specified rate. For instance, if you wanted Premiere to give you audio sampled at 22050 samples per second, 8-bit stereo, you'd make a call like this:

```
(*theData)->specialRate = 22050;
>(*theData)->getAudio)(curSecond, aflagSpecial + aflagStereo, audioBuf,
 (*theData)->privateData);
```

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 0-18: Audio Buffer Sizes**

| Sample rate flags          | 8-bit mono | 8-bit stereo | 16-bit mono | 16-bit stereo |
|----------------------------|------------|--------------|-------------|---------------|
| aflag5KHz                  | 5563       | 11126        | 11126       | 22252         |
| aflag5KHz + aflagDropFrame | 5558       | 11116        | 11116       | 22232         |
| aflag11KHz                 | 11127      | 22254        | 22254       | 44508         |

Table 0–18: Audio Buffer Sizes

| Sample rate flags           | 8-bit mono | 8-bit stereo | 16-bit mono | 16-bit stereo |
|-----------------------------|------------|--------------|-------------|---------------|
| aflag11KHz + aflagDropFrame | 11116      | 22232        | 22232       | 44464         |
| aflag22KHz                  | 22254      | 44508        | 44508       | 89016         |
| aflag22KHz + aflagDropFrame | 22232      | 44464        | 44464       | 88928         |
| aflag44KHz                  | 44100      | 88200        | 88200       | 176400        |
| aflag44KHz + aflagDropFrame | 44056      | 88112        | 88112       | 176224        |

Finally, the `privateData` parameter is (`*theData`)->`privateData`. Note that when a 68K plug-in is called from Power Mac Premiere, the `getAudio` field actually contains a UPP rather than a PowerPC procedure pointer.

### privateData

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

### specialRate

The `specialRate` field is used when you are calling the `getAudio` callback to retrieve audio at an arbitrary sample rate. You store the desired sample rate in Hz in this field, then call `getAudio`. See the description of the `getAudio` callback above for more information.

## Relevant Routines in the Utility Library

There are several routines in `UtilLib` that will make writing a data export module easier. See the definitions of the `GetExport...` routines, the clip routines, and the marker routines in the Premiere Specific Routines section of [The Utility Library](#) chapter for details.

## Examples

The Adobe Premiere Plug-In Toolkit comes with source code for two data export modules that you can use as examples of how to write your own.

### Flattened Movie

This is the source code for the Flattened Movie data export module that ships with Adobe Premiere. Given a QuickTime movie clip, it creates a “flattened” QuickTime movie (one which contains all its data in the data fork, such that it can be used on other computer platforms). This module gives you examples of how to use the `GetExport` utility library routines and how to write a QuickTime-specific export module.

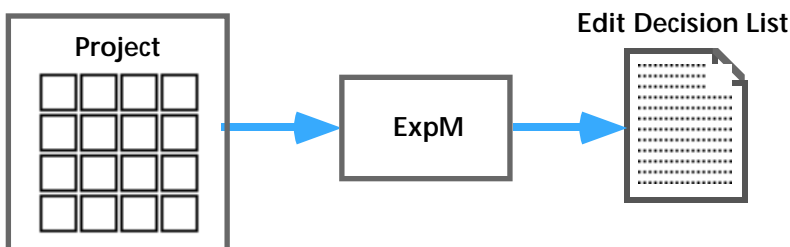
### Storyboard Image

This is the source code for the Storyboard Image data export module that ships with Adobe Premiere. Storyboard Image exports video clips. It puts up a dialog allowing the user to arrange frames in the output picture file, then it generates one or more PICT files containing the in-point frame, any

marked frames, and the out-point frame. This example shows you how to use the `getVideo` callback and shows some examples of calls to the utility library.

# 10 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 are stored in files of type 'ExpM', with creator 'PrMr'. The name of the file is the name that Premiere will display in the Export submenu of the File menu. EDL export files contain at least the resources which are listed below. Following the table is a detailed description of each resource.

**Table 0-19: EDL Export Module Resource IDs**

| Type & ID | Description                                                                     |
|-----------|---------------------------------------------------------------------------------|
| FXvs 1000 | A two-byte version number stored as a short integer.                            |
| ExpM 1000 | The EDL export code itself (68K). The entry point is the resource's first byte. |
| ExpM 1000 | The EDL export code itself (PowerPC). The entry point is specified by the PEF.  |

**NOTE:** Any other resources contained in an EDL export module's resource file should have IDs in the range 600 to 999.

## FXvs 1000

This resource is two bytes in length and gives the version of the module interface. The current version is \$0001.

## ExpM/Expm 1000

These resources contain the code for your EDL export module, ExpM and Expm containing 68K and PowerPC code respectively. The entry point should be declared like this:

```
pascal short EDLExportModule (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 0–20: EDL Export Module Selectors**

| 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 you 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:

**Table 0–21: EDL Type and ID Blocks**

| 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               |
| '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-crn timer, 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]                     |

Table 0–21: EDL Type and ID Blocks

| Type    | ID  | Parent     | Data                      | Description                                      |
|---------|-----|------------|---------------------------|--------------------------------------------------|
| '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]                 |
| '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]        |



Table 0–21: EDL Type and ID Blocks

| Type   | ID | Parent | Data                        | Description                                   |
|--------|----|--------|-----------------------------|-----------------------------------------------|
| '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;

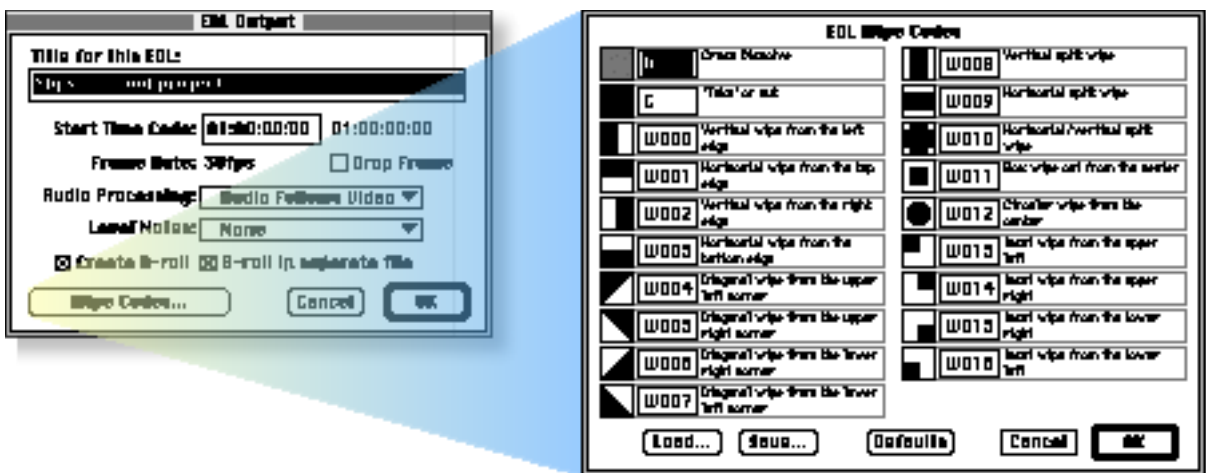
typedef struct
{
 Rect frame; // Bounding frame for video data
 short depth; // Bit depth for video data
} Rec_VIDI;

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 standard save dialog that is presented by the EDL plug ins, shown below:



You may wish to style your EDL save dialog after the one shown above. To facilitate having a “Wipe Codes...” button, Premiere provides a utility routine called `EditWipeCodes`. To put up the wipe code editor dialog (the dialog on the right), do the following:

```
switch (itemHit)
{
 case kOKButton:
 .
 .
 .
 case kWipeCodesButton:
 EditWipeCodes();
 SetPort(myDialog);
 break;
}
```

In the course of generating an EDL text output file, you should use the wipe codes that have been assigned by the user. To access these wipe codes, Premiere provides a utility routine called `GetWipeCodes`. To retrieve the user’s wipe codes, use the following code:

```
{
 long wipeCodes[20];

 GetWipeCodes(codes);
 .
 .
 .
}
```

The array will be filled with the user’s wipe codes. For example, if the wipe codes were set as shown in the picture above, `wipeCodes[0]` would equal ‘D ’, `wipeCodes[1]` would equal ‘C ’, `wipeCodes[2]` would equal ‘W000’, and so on.

## Relevant Routines in the Utility Library

There are a few routines in `UtilLib` that will make writing an EDL export module easier. See the definitions of `MakeWindowForTextFile`, `GetWipeCodes`, `EditWipeCodes`, and the block routines in the `Premiere Specific Routines` section of [The Utility Library](#) chapter for details.

## Examples

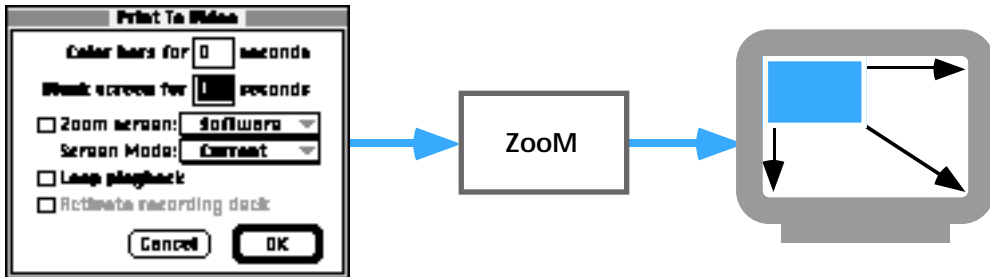
The Adobe Premiere Plug-In Toolkit comes with source code for an EDL export module that you can use as an example of how to write your own.

### Generic EDL

This is the source code for the Generic EDL export module that ships with Adobe Premiere. It contains code to recursively unroll the project data handle and parse the blocks. The source for this module contains a set of block utility routines that greatly simplify the process of generating a text EDL list.

# 11 Zoom Modules

A zoom module in Adobe Premiere is called when the user chooses the “Hardware Zoom” from the Zoom popup menu and/or chooses a mode from the “Screen Mode” menu in the Print To Video dialog. Zoom modules handle hardware-specific details of zooming and video card mode-switching.



The zoom module is provided with information about the GDevice to be zoomed.

Zoom modules are stored in files of type 'ZoomM', with creator 'PrMr'. The name of the file is never shown to the user. Zoom module files contain at least the resources which are listed below. Following the table is a detailed description of each resource.

Table 0–22: Zoom Module Resource IDs

| Type & ID  | Description                                                                      |
|------------|----------------------------------------------------------------------------------|
| ZoomM 1000 | The zoom module code itself (68K). The entry point is the resource's first byte. |
| Zoom 1000  | The zoom module code itself (PowerPC). The entry point is specified by the PEF.  |

## ZooM/Zoom 1000

These resources contain the code for your zoom module, ZooM and Zoom containing 68K and PowerPC code respectively. The entry point should be declared like this:

```
pascal short ZoomModule (short selector, ZoomHand theData);
```

The return value is checked when the selector is cmdCanZoom, cmdCanDo, and cmdGetMode. For the other selectors, the return value is ignored. In those cases you should return 0 for future compatibility.

The selector can take the following values:

**Table 0–23: Zoom Module Selector Values**

| Selector name        | Value | Description                                                   |
|----------------------|-------|---------------------------------------------------------------|
| cmdCanZoom           | 1     | Tell Premiere whether the module can zoom the specified card. |
| cmdZoomIn            | 2     | Zoom the specified card in to x2, remembering old settings.   |
| cmdZoomOut           | 3     | Zoom the specified card back out using stored settings.       |
| cmdCanDo             | 4     | Tells Premiere whether the module can mode switch this card   |
| cmdGetSupportedModes | 5     | Returns bits representing the supported modes                 |
| cmdGetMode           | 6     | Returns the current card mode                                 |
| cmdSetMode           | 7     | Changes the card's mode                                       |

Note: For zoom modules the selector values start at 1, not zero like the rest of Premiere's plug-in modules.

### cmdCanZoom

The cmdCanZoom selector is used to determine whether your zoom module can zoom a specified video card. Information about the video card in question is available in the ZoomHand (described below). Return a result of 1 if you can zoom the specified card, 0 if not.

### cmdZoomIn

The cmdZoomIn selector tells your zoom module to save the current zoom/pan state of the specified card and then zoom it to times-two zoom. Typically the zoom module allocates a handle for the current state of the card and stores it in the zoomData field of the ZoomHand. Most cards that provide hardware zoom also provide hardware pan. If this is the case for your card, you should pan the screen to the upper left of the portion of desktop provided by the card.

### cmdZoomOut

The cmdZoomOut selector tells your zoom module to restore the specified card's zoom/pan settings from the data you stored in the zoomData field of the ZoomHand. Premiere will have cleared the screen before calling your zoom module with this selector. If you allocated a handle and stored it in the zoomData field of the ZoomHand, you should dispose the handle after restoring the state of the card.

### cmdCanDo

The cmdCanDo selector is used to determine whether your zoom module can change the card mode (NTSC, PAL, etc.) Information about the video card in question is available in the ZoomRec structure (described below). Return a result of 1 if you can mode-switch the specified card, 0 if not. Note that it's okay return different responses to a cmdCanDo and cmdCanZoom. If you return 0 to this message, your zoom module will never receive the following three messages.

### cmdGetSupportedModes

The cmdGetSupportedModes selector is used to determine which the set of modes to which the video card can be switched (NTSC, PAL, etc.). The set of modes that Premiere is interested in are represented by the following bit flags:

```
enum {
 modeNTSC = 0x0001, // US NTSC
 modePAL = 0x0002, // European PAL
 modeNTSC443 = 0x0004 // Japanese NTSC
};
```

Add the modes your module (and the video card) can handle together and put them in the mode field of the ZoomRec structure (described below). For instance, if your card can handle NTSC and PAL modes, you would do the following:

```
(*theData)->mode = modeNTSC + modePAL;
```

### cmdGetMode

The cmdGetMode selector tells your zoom module to return the current card mode (one of the values in the enum above) in the mode field of the ZoomRec.

### cmdSetMode

The cmdSetMode selector tells your zoom module to mode-switch the video card to the mode specified in the mode field of the ZoomRec.

## The ZoomRec Structure

Your zoom module is passed a handle to a ZoomRec through the parameter theData. Here's the structure of a ZoomRec:

```
typedef struct
{
 GDHandle theDevice; // The GDevice of board to zoom
 short boardID; // The boardID of the video card
 Handle zoomData; // Can be used by module during zoomIn/Out
 short mode; // Screen mode is passed in/out here
} ZoomRec, **ZoomHand;
```

The fields are as follows:

### theDevice

The theDevice field gives the GDevice handle of the board being zoomed.

### boardID

The boardID field gives the board ID number of the board being zoomed.

### zoomData

The zoomData field is initially set to nil. When you get a cmdZoomIn you typically allocate a handle of state information and store it in this field of the ZoomRec. The value of the zoomData field will be retained, so that when you are later called with a cmdZoomOut selector, you may use the information stored here to restore the video card's state.

### mode

The mode field is used for mode information (in and out) for the cmdGetSupportedModes, cmdGetMode, and cmdSetMode messages.

## Other Details

When the user chooses Print To Video from the Export submenu of the File menu, Premiere calls each of the installed zoom modules with a `cmdCanZoom` selector, until one returns a value of 1. It then sends that module a `cmdCanDo` to see if the module can mode-switch the card. If the module returns 1, then it sends a `cmdGetSupportedModes` to figure out which screen modes to make available in the "Screen Modes" popup menu. If the module returns 0 in response to the `cmdCanDo` message, then the Screen Modes popup is set to "current" and dimmed out. Having gathered this information, Premiere presents the Print To Video dialog box on the screen in question. If the user chooses hardware zoom and a mode switch (for instance), Premiere clears the playback screen to black, sends the zoom module a `cmdGetMode` to save the current mode, sends a `cmdSetMode` to mode-switch the screen, sends a `cmdZoomIn` to zoom in on the top-left of the screen, plays the clip(s) on the screen, clears the screen to black again, sends a `cmdSetMode` with the old mode to reset the screen mode, and finally sends a `cmdZoomOut` to restore the screen zoom to normal. If the screen being zoomed is the main screen (that is, the one with the menu bar), Premiere takes care of hiding and restoring the menu bar.

## Examples

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

### Video - SuperMac

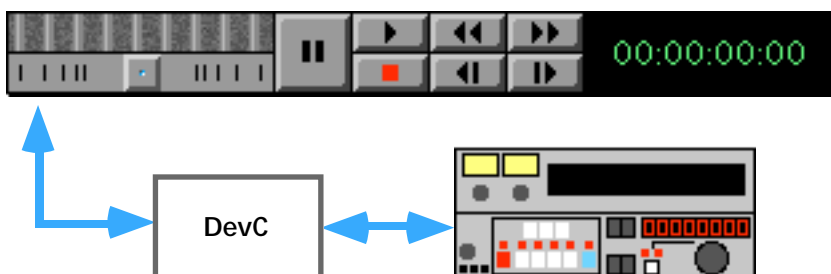
This is the source code for the zoom module used to zoom SuperMac video cards. It makes card-specific calls to the SuperMac video driver to zoom and pan the card.



# Device Control Modules

# 12

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 are stored in files of type 'DevC', with creator 'PrMr'. The name of the file is what appears in the popup menu in the Device Control dialog box. Device control module files contain at least the resources which are listed below. Following the table is a detailed description of each resource.

Table 0-24: Device Control Module Resource IDs

| Type & ID | Description                                                                        |
|-----------|------------------------------------------------------------------------------------|
| DevC 1000 | The device control module code (68K). The entry point is resource's first byte.    |
| Devc 1000 | The device control module code (PowerPC). The entry point is specified by the PEF. |

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

## DevC/Devc 1000

This resource contains the code for your device control module, DevC and Devc containing 68K and PowerPC code respectively. The entry point should be declared like this:

```
pascal short DeviceControlModule (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 take the following values:

**Table 0–25: Device Control Module 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.                            |
| dsQuiet       | 4     | New in 4.2 – Disconnect from device, but don't dispose of allocated structures yet. |

### 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 your 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.

### dsQuiet

This command is new in Premiere 4.2. The dsQuiet selector tells your device control module to disconnect from any hardware, but unlike dsCleanup, it should **not** dispose of any allocated structures yet.

## 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
} 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 the `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 Command-. for instance). The prototype for the abort callback routine is:

```
typedef pascal char (*CallBackPtr) (void);
```

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

Note that when a 68K plug-in is called from Power Mac Premiere, the callback field actually contains a UPP rather than a PowerPC procedure pointer.

## PauseProc

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

```
(*(*theData)->PauseProc)();
AlertSystem(stopIcon, false, kErrors, kMemFailure, 0, 0);
(*(*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.*

Note that when a 68K plug-in is called from Power Mac Premiere, the PauseProc field actually contains a UPP rather than a PowerPC procedure pointer.

## 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.

Note that when a 68K plug-in is called from Power Mac Premiere, the ResumeProc field actually contains a UPP rather than a PowerPC procedure pointer.

## 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 0-26: 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. |
| cmdEject       | 8     | New in 4.2 – Eject the media.                                  |

### cmdGetFeatures

The cmdGetFeatures command tells you to fill out (\*theData)->features with the features of a 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 {
 fDrvrQuiet // New in 4.2 - driver supports quiet mode
 fHasJogMode // New in 4.2 - device has jog capabilities
 fCanEject // New in 4.2 - device can eject its media
 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 fDrvrQuiet bit. If the driver sets this bit, Premiere will issue a dsRestart when a movie capture window is opened and a dsQuiet when the window is closed. A dsCleanup will still be sent at program exit time.

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

`fCanEject` is another new device control bit. Setting this will cause Premiere to send you a `cmdEject` when performing a batch capture and the current reel is known and is not the reel needed.

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

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

The `fGoto` bit indicates that a 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 a 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 a 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 a 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 a deck can play in reverse. If you set this bit, you may get 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 a 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 a 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 a 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.

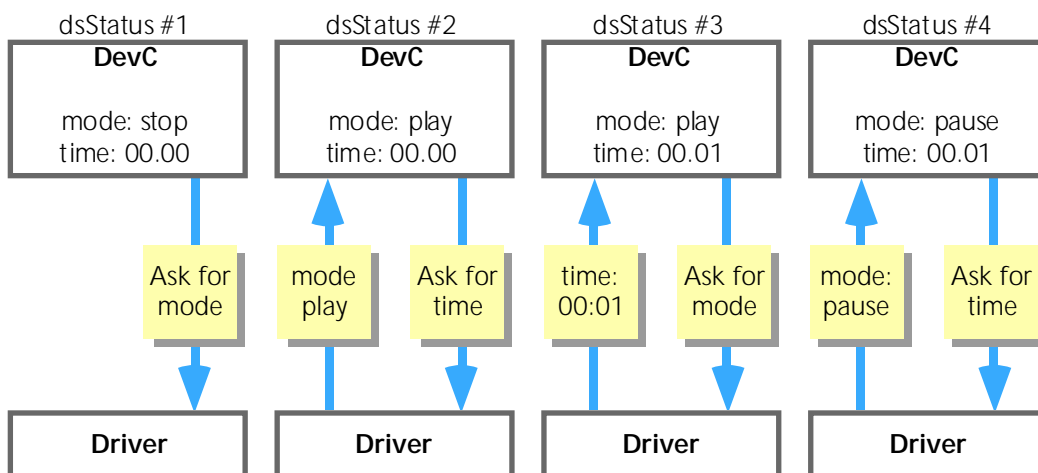
The `fCanJog` bit indicates that a device can quickly move to a nearby timecode location. If you set this bit, Premiere may make `cmdJogTo` calls to your module.

## cmdStatus

The `cmdStatus` command is Premiere's way of finding out what's going on with a 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 a 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 a device into a new operating mode, as specified in `(*theData)->mode`. The modes you may be asked to go into (depending upon your features) are as follows:

```

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 a 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).

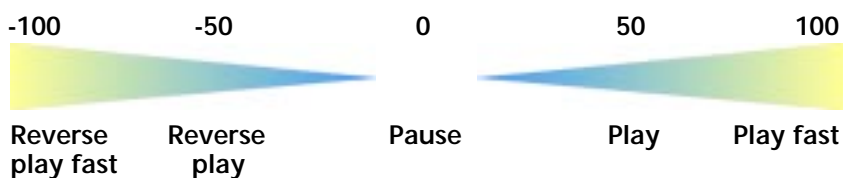
The value in `(*theData)->preroll` tells you how far before the time specified in `(*theData)->timecode` you should actually seek to. In other words, subtract `(*theData)->preroll` from `(*theData)->timecode` and seek there. The preroll value can be set by the user and is generally determined through 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 `Command-period`, 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 a deck doesn't support continuously variable speeds (which many don't), then quantize the speed. For example, here's how Premiere's Visca device control module quantizes 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());
 AlertSystem(stopIcon, false, kErrors, kMemFailure, 0, 0);
 ((*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());
 AlertSystem(stopIcon, false, kErrors, kMemFailure, 0, 0);
 ((*theData).ResumeProc());
 break;
 }

 // Open the driver
 if (result = OpenDriver((StringPtr)"\p.ViSCA",
 &((*LocalRec **)(*theData)->deviceData)->dRefNum))
 {
 .
 .
 .

```

Notice that the 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 a parameter record during development and re-run Premiere, Premiere will kindly pass you a now-invalid `deviceData` blob. 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

You can use Premiere's `AlertSystem` call to put up error alerts if you encounter errors in a device control module. Just remember to frame the error alert 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());
AlertSystem(stopIcon, false, kErrors, kMemFailure, 0, 0);
((*theData).ResumeProc());

```

## Examples

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

### Device

This is the skeleton source code for an Adobe Premiere device control module. It is heavily commented to serve as additional documentation.

# 13 Other Plug-In Types

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

## Photoshop Filters

Premiere can load and apply Adobe Photoshop filters to video clips, but there are currently several limitations to this. Premiere only supports the Photoshop 2.5 API. This has several implications, but most notably the filter must include a PiMI resource to be recognized by Premiere and must not call any Photoshop 3.0 or 4.0 callbacks. Another major implication is Premiere will only load the filters 68K code (since Photoshop 2.5 was pre-PowerPC). With some clever programming you can include an 8BFm (note the lower case m) resource in your filter to find and load your filters PPC segment.

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 ('VFit' modules). If you are writing a video-specific filter you will find it is generally easier to write a VFit 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](http://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. It includes both the PiPL and PiMI resource, and an FltD resource.

## 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.

|                              |    |
|------------------------------|----|
| AlertSystem . . . . .        | 56 |
| Append . . . . .             | 27 |
| AudioLimit . . . . .         | 69 |
| AudioMix . . . . .           | 69 |
| AudioStretch . . . . .       | 68 |
| AudioSum . . . . .           | 69 |
| BcdToBin . . . . .           | 61 |
| BcdToStr . . . . .           | 61 |
| BetterNewGWorld . . . . .    | 57 |
| BinToBcd . . . . .           | 62 |
| BinToStr . . . . .           | 61 |
| BuildString . . . . .        | 41 |
| ButtonFrame . . . . .        | 20 |
| CenterModal . . . . .        | 54 |
| CenterModalKeys . . . . .    | 54 |
| CenterWin2Win . . . . .      | 26 |
| CenterWindow . . . . .       | 26 |
| CenterWindowOnMain . . . . . | 27 |
| CenterWindowOnMain . . . . . | 56 |
| CenterWinOffset . . . . .    | 26 |
| CheckStop . . . . .          | 52 |
| CheckStopUpdate . . . . .    | 52 |
| ClipAspect . . . . .         | 45 |
| ClipFile . . . . .           | 45 |
| ClipRate . . . . .           | 44 |
| ClipSize . . . . .           | 45 |
| ClipStart . . . . .          | 44 |
| ClipWidth . . . . .          | 45 |
| Color82RGB . . . . .         | 58 |
| CountClipMarkers . . . . .   | 46 |
| CountTypeBlocks . . . . .    | 64 |
| CountVolumes . . . . .       | 36 |
| DataLookup . . . . .         | 34 |
| decimalfilter . . . . .      | 55 |
| DeleteHandItem . . . . .     | 35 |
| DepthOf . . . . .            | 30 |
| DirIDFromPath . . . . .      | 35 |
| DisableDItem . . . . .       | 17 |
| DisposeModal . . . . .       | 54 |
| DistortFixed . . . . .       | 70 |
| DistortPolygon . . . . .     | 68 |
| DitherBox . . . . .          | 59 |
| DrawAControl . . . . .       | 25 |
| DrawDItem . . . . .          | 18 |
| DrawFullCL8Hand . . . . .    | 32 |
| DrawCL8 . . . . .            | 31 |
| DrawCL8Hand . . . . .        | 32 |
| DrawItemBox . . . . .        | 22 |

|                            |    |
|----------------------------|----|
| DrawItemFrame . . . . .    | 22 |
| DrawSIC4 . . . . .         | 31 |
| DrawSTRVert . . . . .      | 31 |
| EditWipeCodes . . . . .    | 64 |
| EnableDItem . . . . .      | 17 |
| EqualColor . . . . .       | 33 |
| EqualColor8 . . . . .      | 33 |
| EraseGrow . . . . .        | 30 |
| exists . . . . .           | 47 |
| ExtractBlockData . . . . . | 65 |
| ExtraNewHandle . . . . .   | 44 |
| FillMem . . . . .          | 16 |
| FindBlock . . . . .        | 64 |
| FindClipMarker . . . . .   | 46 |
| FindFType . . . . .        | 47 |
| FindIndString . . . . .    | 28 |
| FindSelect . . . . .       | 33 |
| FixedDiv . . . . .         | 37 |
| FixedToFixed . . . . .     | 70 |
| FixMulDiv . . . . .        | 37 |
| FlashControl . . . . .     | 20 |
| FormatTimecode . . . . .   | 60 |
| FrameButton . . . . .      | 20 |
| FrameErase . . . . .       | 28 |
| FrameGrayButton . . . . .  | 20 |
| freehook . . . . .         | 54 |
| gBit16OK . . . . .         | 76 |
| gBottleNecks . . . . .     | 75 |
| gCompileErr . . . . .      | 72 |
| gDecimalPt . . . . .       | 75 |
| GetAGlobal . . . . .       | 41 |
| GetArrow . . . . .         | 33 |
| GetBlock . . . . .         | 65 |
| GetClipBackwards . . . . . | 46 |
| GetClipMarker . . . . .    | 46 |
| GetClipTitle . . . . .     | 45 |
| GetCMax . . . . .          | 23 |
| GetCRef . . . . .          | 23 |
| GetCValue . . . . .        | 17 |
| GetDHandle . . . . .       | 19 |
| GetDRect . . . . .         | 18 |
| GetDType . . . . .         | 19 |
| GetEText . . . . .         | 22 |
| GetExportClipID . . . . .  | 62 |
| GetExportDivisor . . . . . | 62 |
| GetExportFPS . . . . .     | 63 |
| GetExportFSSpec . . . . .  | 62 |
| GetExportMovie . . . . .   | 62 |
| GetFSSpec . . . . .        | 47 |

|                            |    |
|----------------------------|----|
| GetGroupVal . . . . .      | 18 |
| GetIndVolume . . . . .     | 36 |
| GetIVal . . . . .          | 21 |
| GetMenuWidth . . . . .     | 37 |
| GetModifiers . . . . .     | 52 |
| GetRate . . . . .          | 63 |
| GetVollIndex . . . . .     | 36 |
| GetWipeCodes . . . . .     | 64 |
| gExportRef . . . . .       | 72 |
| gGWStrip . . . . .         | 73 |
| gHasOutline . . . . .      | 75 |
| gHaveDragAndDrop . . . . . | 77 |
| gLockStillAspect . . . . . | 76 |
| gMaxHeight . . . . .       | 76 |
| gMaxWidth . . . . .        | 76 |
| gMultiply . . . . .        | 75 |
| gOneBitWorld . . . . .     | 73 |
| gPluginDirID . . . . .     | 73 |
| gPluginVRefNum . . . . .   | 72 |
| gPrefDirID . . . . .       | 73 |
| gPrefVRefNum . . . . .     | 73 |
| gPrintRec . . . . .        | 73 |
| gqd . . . . .              | 77 |
| gQTVers . . . . .          | 77 |
| gResFileNum . . . . .      | 72 |
| gRGB2UV . . . . .          | 74 |
| gRGB2Y . . . . .           | 74 |
| gSaveRef . . . . .         | 71 |
| gStillDefault . . . . .    | 72 |
| gSysVersion . . . . .      | 72 |
| gSysWindow . . . . .       | 76 |
| gTempDirID . . . . .       | 73 |
| gTempVRefNum . . . . .     | 73 |
| gTickDenom . . . . .       | 76 |
| gTickNumer . . . . .       | 76 |
| gTicks . . . . .           | 76 |
| gUPPBottleNecks . . . . .  | 76 |
| gWorkDirID . . . . .       | 73 |
| gWorkVRefNum . . . . .     | 73 |
| HatchBox . . . . .         | 32 |
| HiliteDControl . . . . .   | 17 |
| hoursfilter . . . . .      | 55 |
| ImageKey . . . . .         | 70 |
| InsertHandItem . . . . .   | 35 |
| inttox80 . . . . .         | 38 |
| InvalItem . . . . .        | 19 |
| LimitLong . . . . .        | 37 |
| LockHHI . . . . .          | 16 |
| longdoubletox80 . . . . .  | 38 |



|                                  |    |
|----------------------------------|----|
| MakeWindowForFile . . . . .      | 56 |
| MakeWindowForTextFile . . . . .  | 57 |
| MapPolygon . . . . .             | 68 |
| modalfilter . . . . .            | 55 |
| ModifyDItem . . . . .            | 19 |
| MyPutFile . . . . .              | 53 |
| MySetCursor . . . . .            | 52 |
| NextBlock . . . . .              | 64 |
| NextClipMarker . . . . .         | 46 |
| notextfilter . . . . .           | 55 |
| NumToQuan . . . . .              | 44 |
| OffscreenBox . . . . .           | 58 |
| OffsetControl . . . . .          | 25 |
| OffsetCSize . . . . .            | 24 |
| ParseTimecode . . . . .          | 60 |
| PathFromDirID . . . . .          | 36 |
| pie . . . . .                    | 38 |
| PinPt . . . . .                  | 31 |
| PositionDialog . . . . .         | 20 |
| PrCustomGetFile . . . . .        | 39 |
| PrCustomGetFilePreview . . . . . | 39 |
| PrCustomPutFile . . . . .        | 39 |
| PrDebug . . . . .                | 50 |
| PreviousClipMarker . . . . .     | 46 |
| PrModalDialog . . . . .          | 38 |
| PrSCSetInfo . . . . .            | 40 |
| PrSGSettingsDialog . . . . .     | 40 |
| PrSndDisposeChannel . . . . .    | 40 |
| PrSndNewChannel . . . . .        | 40 |
| PrTrackControl . . . . .         | 40 |
| pt2GDevice . . . . .             | 30 |
| pt2GDeviceRect . . . . .         | 30 |
| PtClose . . . . .                | 32 |
| QuickAbs . . . . .               | 37 |
| ReverseLookup . . . . .          | 34 |
| RGB2Color . . . . .              | 59 |
| SafeDrawControl . . . . .        | 25 |
| SafeNewGWorld . . . . .          | 57 |
| SafeSetCValue . . . . .          | 25 |
| SafeSetHandleSize . . . . .      | 16 |
| SafeSetupAIFFHeader . . . . .    | 50 |
| SetAGlobal . . . . .             | 41 |
| SetBackColor . . . . .           | 29 |
| SetBackGray . . . . .            | 28 |
| SetCAction . . . . .             | 24 |
| SetCMax . . . . .                | 23 |
| SetColor . . . . .               | 29 |
| SetColorFace . . . . .           | 29 |
| SetCRef . . . . .                | 23 |

|                                 |    |
|---------------------------------|----|
| SetCValue . . . . .             | 18 |
| SetDRect . . . . .              | 18 |
| SetEText . . . . .              | 22 |
| SetFont . . . . .               | 58 |
| SetGray . . . . .               | 28 |
| SetIVal . . . . .               | 21 |
| SetResCTitle . . . . .          | 24 |
| SetSelect . . . . .             | 33 |
| ShowModal . . . . .             | 54 |
| SlotToGD . . . . .              | 32 |
| SlotToRect . . . . .            | 32 |
| SpecialGetFile . . . . .        | 53 |
| SpecialGetFilePreview . . . . . | 53 |
| SpinCurs . . . . .              | 51 |
| StopCurs . . . . .              | 51 |
| Str2Time . . . . .              | 60 |
| StrCopy . . . . .               | 27 |
| StretchBits . . . . .           | 67 |
| StringsSame . . . . .           | 27 |
| StringTrunc . . . . .           | 43 |
| StrToBcd . . . . .              | 61 |
| StrToBin . . . . .              | 61 |
| StructsSame . . . . .           | 17 |
| SuperFileDispose . . . . .      | 50 |
| SuperFileInit . . . . .         | 47 |
| SuperFileRead . . . . .         | 49 |
| SuperFileSeek . . . . .         | 49 |
| Time2Str . . . . .              | 59 |
| TruncLength . . . . .           | 44 |
| TypeToStr . . . . .             | 28 |
| UpdateAllWindows . . . . .      | 56 |
| UserItem . . . . .              | 19 |
| Validate . . . . .              | 21 |
| VertCenter . . . . .            | 30 |
| WhichCell . . . . .             | 33 |
| WidenMenu . . . . .             | 37 |
| WidenMenu2Box . . . . .         | 36 |
| x80toint . . . . .              | 38 |
| x80tolongdouble . . . . .       | 38 |